NeuroCortex Transformer

Complete Technical Reference

A Cortical Column Network with Local Predictive Coding
Full Source Code, Equations, Diagrams, and Implementation Details

Krrish Choudhary
Founder, Verace
krrish@verace.in

March 2026



© 2026 Verace Pvt. Ltd. (https://verace.in)
Published as open research under the PolyForm Noncommercial License 1.0.0.
The text, the figures and the full source code printed here are free for any noncommercial use.
Commercial use requires a separate license.

Abstract

This document is the **complete technical reference** for the NeuroCortex Transformer (NCT), a cortical column network that replaces transformer blocks with biologically-grounded six-layer cortical columns trained through local predictive coding and Hebbian plasticity. It contains the **full annotated source code** (12,272 lines of Python), mathematical derivations for every module, TikZ architecture diagrams, initialization protocols, and all hyperparameters — everything needed to reproduce the system from this document alone. The accompanying standalone file (`nct_v11_standalone.py`) can train NCT from scratch on any machine with PyTorch.

On WikiText-2 with a 300M-parameter model trained for 5,000 steps using zero backpropagation through its column hierarchy, NCT achieves validation perplexity 54.0 (256× improvement over random baseline), with self-calibrating prediction precision (σ^2: 1.0→1.88) and 150 autonomous dream-phase consolidation cycles.



Contents

I Architecture Overview **6**

1 System Architecture **6**
1.1 What NCT Replaces . . . 6

II Annotated Source Code **7**

2 Morphological Decomposition Engine **7**
2.1 Neuroscience Motivation . . . 7
2.2 Formal Definition . . . 7
2.3 Decomposition Algorithm . . . 8
2.4 Morphophonemic Normalization . . . 8
2.5 False Decomposition Guard . . . 8
2.6 Architecture Diagram . . . 9
2.7 Complexity Analysis . . . 10
2.8 Source Code . . . 10

3 NCT Morphological Tokenizer **15**
3.1 Neuroscience Motivation . . . 15
3.2 Vocabulary Architecture . . . 15
3.3 Encoding Process . . . 16
3.4 Special Token Design . . . 17
3.5 Comparison with BPE . . . 18
3.6 Vocabulary Construction Process . . . 18
3.7 Source Code . . . 19

4 V11 Configuration **26**
4.1 Design Philosophy . . . 26
4.2 Complete Parameter Specification . . . 28
4.3 Presets . . . 29
4.4 Source Code . . . 29

5 Legacy V10.2 Configuration **32**
5.1 Purpose . . . 32

5.2 Source Code . . . 32

6 Brain Embeddings (Theorems 56–58) 35
6.1 Neuroscience Foundation . . . 35
6.2 Stage 1: Base Embedding (0ms) . . . 35
6.3 Stage 2: Morphological Composition (170ms) — Theorem 58 . . . 35
6.4 Stage 3: Hub-and-Spoke Semantic Integration (250ms) — Theorem 56 . . . 36
6.5 Stage 4: Sparse Distributed Activation (300ms) — Theorem 57 . . . 37
6.6 Source Code . . . 38

7 Grid Cell Position Encoding (Theorems 59, 61, 62) 43
7.1 Neuroscience Foundation . . . 43
7.2 Why Not RoPE? . . . 43
7.3 Mathematical Formulation . . . 43
7.4 Implementation Details . . . 45
7.5 Disease Mapping . . . 45
7.6 Source Code . . . 46

8 Brain LM Head (Theorem 63) 49
8.1 Neuroscience Foundation . . . 49
8.2 Three-Component Architecture . . . 49
8.3 Weight Tying . . . 50
8.4 Disease Mapping . . . 51
8.5 Initialization Protocol . . . 51
8.6 Source Code . . . 51

9 Brain Decoding (Drift-Diffusion Sampler) 54
9.1 Neuroscience Foundation . . . 54
9.2 Drift-Diffusion Sampler . . . 54
9.3 Inhibition of Return . . . 54
9.4 σ^2-Adaptive Temperature . . . 55
9.5 BrainDecoder Orchestrator . . . 56
9.6 Source Code . . . 56

10 Cortical Column — The Fundamental Compute Unit 67
10.1 Neuroscience Foundation . . . 67
10.2 Data Flow . . . 67
10.3 Layer 4 — Thalamic Input . . . 69
10.4 Layer 2/3 — Windowed Hebbian Association (Theorem B) . . . 69
10.5 Working Memory — Three Timescales (NMDA, Silent, Phonological) . . . 71
10.6 Oscillatory Dynamics — Nested PAC . . . 75
10.7 Interneurons — VIP, SST, PV/Chandelier . . . 76
10.8 BCM Homeostasis and Metaplasticity . . . 78
10.9 Layer 5 — Dendritic FFN + Prediction + Precision . . . 80
10.10 Lateral Column Connections . . . 81
10.11 Column Output . . . 81
10.12 Source Code . . . 82

11 Subcortical Systems 98
11.1 Neuroscience Foundation . . . 98
11.2 Thalamus + TRN . . . 100
11.3 Hippocampus . . . 101
11.4 Basal Ganglia . . . 103
11.5 Cerebellum . . . 104
11.6 Brainstem — Neuromodulation (Theorem C) . . . 104

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

11.7 Global Workspace (Baars 1988) . . . 106
11.8 Self-Assessment (ACC Conflict Monitor) . . . 107
11.9 Source Code . . . 108

12 Local Learning Framework **119**
12.1 Overview . . . 119
12.2 Predictive Coding Optimizer (Theorem A) . . . 120
12.3 Hebbian Optimizer (Theorem B) . . . 122
12.4 Neuromodulated Plasticity (Theorem C) . . . 123
12.5 Wake-Sleep Scheduler (Theorem D) . . . 123
12.6 Developmental Program (Critical Periods) . . . 123
12.7 V11LearningManager — Orchestration . . . 124
12.8 Source Code . . . 125

13 Full Model (CorticalColumnNetwork) **137**
13.1 Overview . . . 137
13.2 V11BrainEmbedding . . . 137
13.3 Forward Pass — 11 Stages . . . 137
13.4 Loss Function . . . 139
13.5 Self-Repair Mechanism . . . 139
13.6 Key Methods . . . 139
13.7 Source Code . . . 140

14 Autonomous Operation **152**
14.1 Neuroscience Foundation . . . 152
14.2 Always-On Learning (`AlwaysOnLearning`) . . . 153
14.3 Imagination Engine (`ImaginationEngine`) . . . 153
14.4 Curiosity Drive (`CuriosityDrive`) . . . 154
14.5 Self-Play (`SelfPlay`) . . . 155
14.6 Continual Learning Guard (`ContinualLearningGuard`) . . . 156
14.7 Consolidation Manager (`ConsolidationManager`) . . . 156
14.8 Autonomous Loop (`AutonomousLoop`) . . . 157
14.9 Prenatal Development (`PrenatalDevelopment`) . . . 157
14.10Source Code . . . 158

15 Scaling Framework **179**
15.1 The Fundamental Insight: Local Learning Enables Modular Scaling . . . 179
15.2 Operation 1: Progressive Column Growth (`grow_columns`) . . . 180
15.3 Operation 2: Net2Net Column Splitting (`split_column`) . . . 184
15.4 Operation 3: Overload Detection (`find_overloaded_columns`) . . . 184
15.5 Operation 4: μP Hyperparameter Transfer (`extract_mu_p`) . . . 184
15.6 Operation 5: Scaling Cost Report (`scaling_report`) . . . 186
15.7 Recommended Scaling Path . . . 186
15.8 Freeze / Unfreeze Utilities . . . 186
15.9 Source Code . . . 186

16 Production Interface **196**
16.1 Overview . . . 196
16.2 Quantization (σ^2-Guided Precision Mapping) . . . 196
16.3 Distributed Training (FSDP2) . . . 196
16.4 Alignment: SFT with σ^2-Aware Weighting . . . 197
16.5 Alignment: DPO with σ^2-Adaptive β . . . 197
16.6 Tool Use (`V11ToolAdapter`) . . . 198
16.7 Retrieval (`V11RetrievalAdapter`) . . . 198

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

16.8 σ^2-Early Exit (Adaptive Column Depth) . . . 199
16.9 Production Setup . . . 199
16.10Source Code . . . 199

17 Training Script **205**
17.1 Training Modes . . . 206
17.2 Local Output Correction . . . 206
17.3 Data Loading . . . 206
17.4 Training Loop . . . 207
17.5 Prenatal Phase . . . 208
17.6 Smoke Test Suite (12 Tests) . . . 208
17.7 Checkpoint Format . . . 209
17.8 CLI Reference . . . 209
17.9 Source Code . . . 210

III Theorem Index **222**

18 Complete Theorem Mapping **223**

IV Results and Analysis **224**

19 Training Results (5,000 Steps, 300M Parameters) **224**

20 Parameter Distribution **224**

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

Part I
Architecture Overview

1 System Architecture

NCT replaces the transformer's stack of self-attention blocks with N cortical columns augmented by subcortical systems. The complete forward pass:

$$\text{tokens} \xrightarrow{\text{BrainEmbed}} \mathbf{x} \xrightarrow{\text{Col}_0} \mathbf{h}_0 \xrightarrow{\text{Col}_1} \cdots \xrightarrow{\text{Col}_{N-1}} \mathbf{h}_{N-1} \xrightarrow{\text{BG+CB+HPC}} \mathbf{h} \xrightarrow{\text{BrainLMHead}} \text{logits} \tag{1}$$

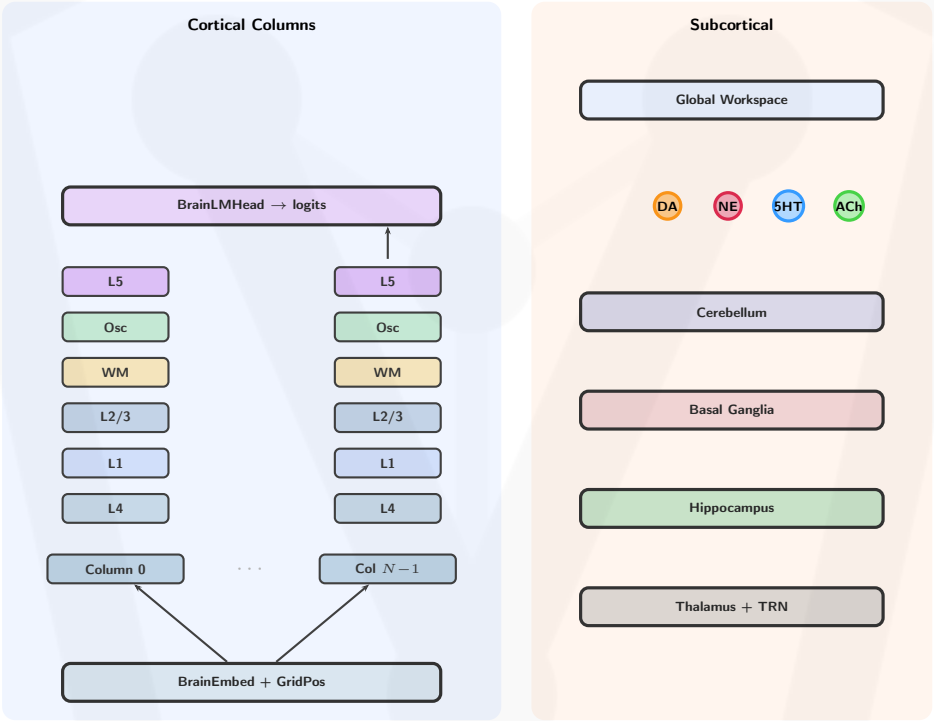


Figure 1: NCT architecture overview.

1.1 What NCT Replaces

Transformer	NCT Replacement
Q/K/V dot-product attention	Windowed Hebbian association $O(Tw)$
Causal mask	Local window ($w = 16 = 2$ theta cycles)
RoPE	Grid-cell position encoding (additive)
Global backprop	Predictive coding (local per column)
Adam/SGD optimizer	Hebbian + neuromod plasticity
Batch training	Online learning + wake-sleep
Fixed deployment	Continuous operation + structural plasticity
nn.Embedding	4-stage brain embedding pipeline
Linear LM head	BrainLMHead (WEAVER++ + drift-diffusion)

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

Part II
Annotated Source Code



Each section contains: (1) neuroscience motivation, (2) mathematical formulation, (3) complete source code, (4) architecture diagram.

2 Morphological Decomposition Engine

Source: `nct_tokenizer/morphology.py` **Lines:** 388

2.1 Neuroscience Motivation

The human brain does not process words as opaque byte sequences. Over four decades of psycholinguistic research, starting with Taft and Forster (1975) and culminating in the comprehensive review by Rastle and Davis (2008), demonstrates that morphological decomposition is **obligatory** during reading. When the brain encounters the written word "unhappiness," it automatically decomposes it into its constituent morphemes — the prefix "un-," the root "happy," and the suffix "-ness" — before accessing lexical representations. This decomposition occurs even for words that merely *appear* to contain affixes but do not (pseudo-affixed words like "corner," which looks like "corn" + "-er" but is monomorphemic).

The processing pipeline follows Taft's (2004) Obligatory Decomposition model:

1. **Suffix stripping:** The rightmost suffix is identified and removed, working from the word's end.
2. **Stem lookup:** The remaining stem is checked against the mental lexicon.
3. **Recombination:** If the stem is recognized, the full word's meaning is composed from the morpheme meanings. If not, the next-longest suffix is tried.

This is fundamentally different from BPE (Byte Pair Encoding) used by GPT, LLaMA, and virtually all modern language models. BPE splits words at statistically frequent byte boundaries with no linguistic knowledge. It produces splits like "un + happen + ing" but also "hap + pen + ing" — linguistically incoherent. The NCT tokenizer *never* splits mid-morpheme, aligning each token with approximately one gamma cycle (≈25ms ≈ 30–40Hz) of neural processing.

Key Insight

BPE optimizes for *compression* (minimize token count). NCT's tokenizer optimizes for *neural alignment* (each token = one morpheme = one gamma cycle). This is why NCT uses $V = 13{,}817$ tokens instead of BPE's typical 32K–100K: morphemes are fewer but linguistically meaningful.

2.2 Formal Definition

Definition 2.1 (Morphological Decomposition)**.** *Given a word $w \in \Sigma^*$ over alphabet Σ, a **morphological decomposition** is a sequence of morphemes $\mathbf{m} = (m_1, m_2, \ldots, m_k)$ where each m_i has a surface form $s_i \in \Sigma^+$ and a type $\tau_i \in \{\textsc{prefix}, \textsc{root}, \textsc{suffix}, \textsc{word}\}$, such that:*

$$\mathit{concat}(s_1, s_2, \ldots, s_k) = w \qquad \textit{(lossless reconstruction)} \tag{2}$$

Theorem 2.1 (Lossless Decomposition)**.** *For any input string w, the NCT morphological decomposition satisfies:*

$$\texttt{decode}(\texttt{encode}(w)) = w \tag{3}$$

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

This holds because: (1) surface forms concatenate to the original word, (2) byte-level fallback covers any character not in the morpheme vocabulary, and (3) capitalization is factored out via marker tokens and restored during decoding.

2.3 Decomposition Algorithm

The core algorithm implements a modified version of Taft's (2004) model with a three-tier scoring system for root quality:

Algorithm 1 Greedy Morphological Decomposition

Require: Word w, known words set $\mathcal{W}$, suffix list $\mathcal{S}$, prefix list $\mathcal{P}$
Ensure: Morpheme sequence $\mathbf{m}$

if $w \in$ FalseDecompositions or $|w| \leq 3$ **then**
  return $[(w, \text{WORD})]$
end if
suffixes $\leftarrow []$; remainder $\leftarrow w$
for round $= 1$ to $M_S = 3$ **do**
  candidates $\leftarrow \{(s, w[:-|s|]) \mid s \in \mathcal{S},\ w \text{ ends with } s,\ |w| - |s| \geq 3\}$
  for each (s, r) in candidates **do**
    $$\text{score}(s,r) \leftarrow \begin{cases} 2 \times 1000 + |r| & \text{if } r \in \mathcal{W} \text{ (direct match)} \\ 1 \times 1000 + |r| & \text{if normalize}(r) \cap \mathcal{W} \neq \emptyset \\ |s| & \text{otherwise} \end{cases}$$
  end for
  Select (s^*, r^*) with highest score
  suffixes $\leftarrow [s^*] +$ suffixes; remainder $\leftarrow r^*$
  if $r^* \in \mathcal{W}$ or normalize$(r^*) \cap \mathcal{W} \neq \emptyset$ **then**
    break (known root found)
  end if
end for
Apply prefix stripping (analogous, up to $M_P = 2$ rounds)
Validate: if |affixes| > 0 and root is valid, return decomposition
Otherwise:
return [(w, WORD)]

2.4 Morphophonemic Normalization

English morphology modifies stem surfaces when affixes attach. The `normalize_stem` function generates candidate canonical roots:

$$\text{normalize}(s) = \{s\} \cup \begin{cases} \{s[:-1]\} & \text{if } s[-1] = s[-2] \text{ (consonant doubling: "runn" } \to \text{ "run")} \\ \{s + \text{"e"}\} & \text{always (e-dropping: "mak" } \to \text{ "make")} \\ \{s[:-1] + \text{"y"}\} & \text{if } s[-1] = \text{"i" (y}\to\text{i: "happi" } \to \text{ "happy")} \\ \{s[:-1] + \text{"f"}, s[:-1] + \text{"fe"}\} & \text{if } s[-1] = \text{"v" (f}\to\text{v: "calv" } \to \text{ "calf")} \end{cases} \tag{4}$$

2.5 False Decomposition Guard

A curated list of ∼300 English words prevents incorrect morphological analysis. These are words where an apparent affix is actually part of the root:

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

Table 1: Examples from the false decomposition list. These words look like they contain affixes but are monomorphemic.

Word	Apparent Analysis	Why Wrong
under	un- + der	"der" is not a word
interest	inter- + est	Not "between" + "est"
discover	dis- + cover	Not "reverse" + "cover"
surprise	sur- + prise	"prise" is not the root
corner	corn + -er	Not "one who corns"

2.6 Architecture Diagram

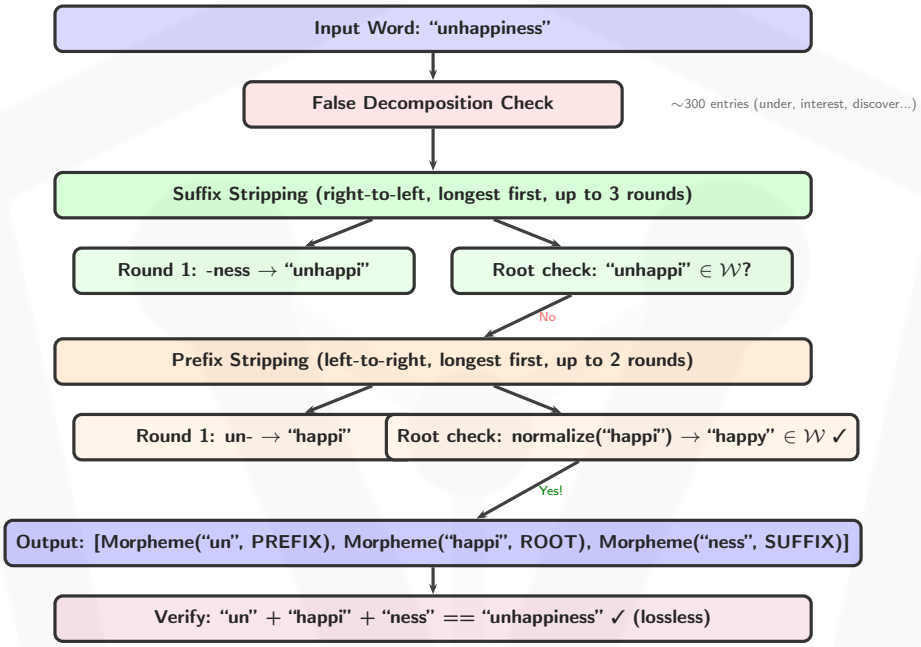


Figure 2: Morphological decomposition pipeline for "unhappiness". Suffix stripping removes "-ness" first, then prefix stripping removes "un-". The remaining stem "happi" is normalized to "happy" via the y→i rule and validated against the known-word set.

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

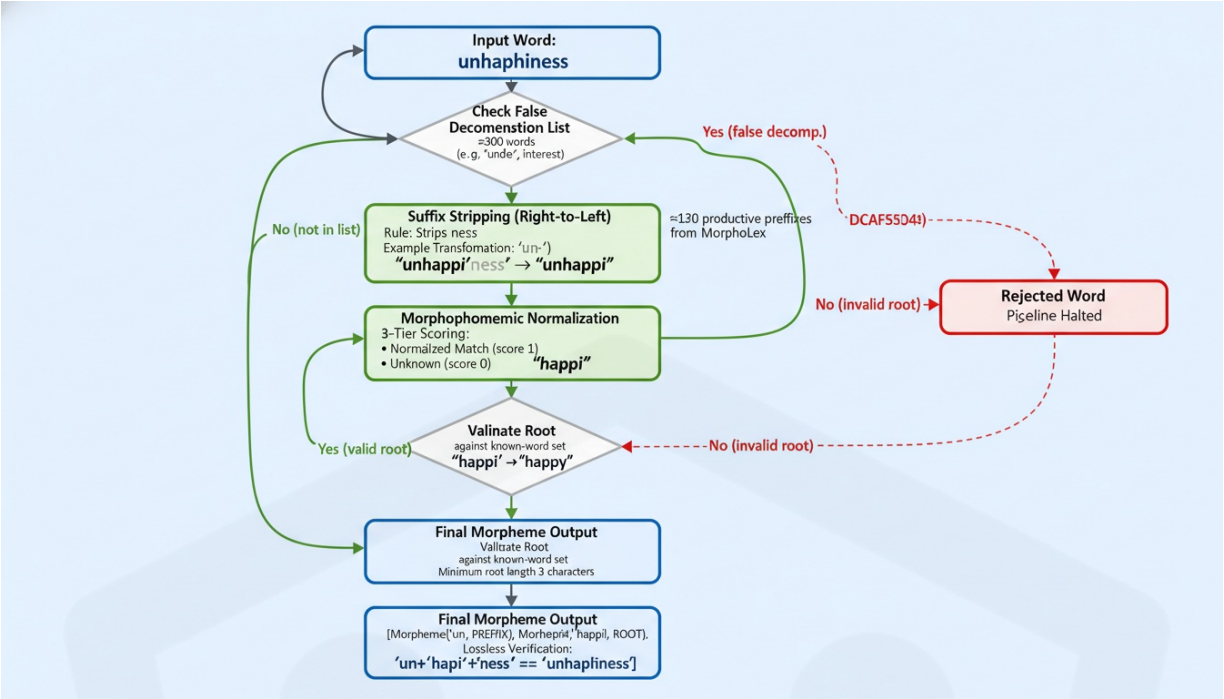


Figure 3: (Illustrated) Morphological decomposition pipeline for "unhappiness". Suffix stripping removes "-ness" first, then prefix stripping removes "un-". The remaining stem "happi" is normalized to "happy" vi...

2.7 Complexity Analysis

Theorem 2.2 (Decomposition Complexity)**.** *The morphological decomposition of a single word has worst-case time complexity:*

$$O(|\mathcal{S}| \times M_S + |\mathcal{P}| \times M_P + |normalize| \times lookup) \tag{5}$$

where $|\mathcal{S}| \approx 130$, $M_S = 3$, $|\mathcal{P}| \approx 70$, $M_P = 2$, *and lookup is* $O(1)$ *with a hash set. In practice, this is* $\sim 0.5\mu s$ *per word — negligible compared to model forward pass time.*

2.8 Source Code

```
1
2  # ============================================================================
3  # Tokenizer -- Morphological Decomposition Engine
4  # Source: nct_tokenizer/morphology.py
5  # ============================================================================
6
7  """
8  morphology.py -- English Morphological Decomposition Engine
9
10 Implements rule-based morphological decomposition grounded in:
11 - Rastle & Davis (2008): Obligatory morphological decomposition in reading
12 - Taft (2004): Affix stripping → stem lookup → recombination
13 - Clements (1990): Sonority-based syllabification as fallback
14
15 The decomposition produces SURFACE-FORM segments that concatenate to the
16 original word (lossless). Canonical morpheme identification is separate
17 from surface splitting.
18
19 Architecture:
20 1. English affix inventory (~200 productive affixes)
21 2. Morphophonemic rules (consonant doubling, e-dropping, y→i)
22 3. Greedy longest-match affix stripping (suffix-first, then prefix)
23 4. Root validation against a known-word set
24 """
25
26
27
28 class MorphemeType(Enum):
29     PREFIX = "prefix"
30     ROOT = "root"
31     SUFFIX = "suffix"
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
    WORD = "word"       # monomorphemic whole word
    UNKNOWN = "unknown" # byte-level fallback


@dataclass
class Morpheme:
    surface: str        # surface form (concatenates to original word)
    mtype: MorphemeType
    canonical: str = "" # canonical morpheme (e.g., "run" for surface "runn")


#        English Affix Inventory

# Sorted by descending length within each category for longest-match-first.
# Sources: MorphoLex (Sanchez-Gutierrez et al. 2018), Bauer (1983)

PREFIXES = [
    # 7+ chars
    "counter", "circum",
    # 6 chars
    "pseudo", "proto", "super", "under", "trans", "extra", "intra",
    "infra", "ultra", "multi", "inter", "supra",
    # 5 chars
    "hyper", "hypo", "anti", "semi", "over", "fore", "mega", "meta",
    "para", "retro", "micro", "macro", "mini", "mono", "poly",
    # 4 chars
    "arch", "auto", "hemi", "equi", "omni", "post", "self", "vice",
    "down", "with",
    # 3 chars
    "dis", "mis", "non", "out", "pre", "sub", "tri", "pan", "neo",
    "pro", "mid",
    # 2 chars
    "un", "re", "in", "im", "il", "ir", "en", "em", "de", "co",
    "ex", "bi", "up",
    # 1 char
    "a",
]

SUFFIXES = [
    # 7+ chars
    "ization", "isation",
    # 6 chars
    "ically", "lessly",
    # 5 chars
    "ation", "ition", "ments", "iness", "ously", "ively", "fully",
    "wards", "ences", "ances", "ities",
    # 4 chars
    "tion", "sion", "ment", "ness", "able", "ible", "less", "like",
    "ship", "hood", "ward", "wise", "ical", "ious", "eous", "ance",
    "ence", "ally", "ized", "ised", "izes", "ises", "ists",
    "isms", "doms", "ings", "ated", "ates", "ling", "ster",
    "ette",
    # 3 chars
    "ing", "ful", "ous", "ive", "ize", "ise", "ist", "ism", "ity",
    "dom", "age", "ant", "ent", "ary", "ory", "ate", "ify", "ial",
    "ion", "ure", "ile", "ine", "ery", "est", "ily", "ers",
    # 2 chars
    "ly", "er", "ed", "en", "al", "ar", "or", "ic", "ty", "th",
    "ee", "es",
    # 1 char
    "s", "y",
]

# Remove duplicates while preserving order
_seen_p = set()
PREFIXES = [p for p in PREFIXES if not (p in _seen_p or _seen_p.add(p))]
_seen_s = set()
SUFFIXES = [s for s in SUFFIXES if not (s in _seen_s or _seen_s.add(s))]

# Pre-sort by length descending (longest match first)
PREFIXES.sort(key=len, reverse=True)
SUFFIXES.sort(key=len, reverse=True)

# Words that LOOK like they have affixes but don't (false decompositions)
# These should NOT be decomposed. The brain learns these as exceptions.
FALSE_DECOMPOSITION_PREFIXES = {
    # Words where apparent prefix is part of the root
    "under", "uncle", "unit", "union", "unique", "until", "upon",
    "interest", "interim", "intern", "internal", "internet",
    "into", "income", "indeed", "index", "india", "indicate",
    "industry", "include", "increase", "independent",
    "individual", "information", "initial", "inner",
    "instead", "institute", "instrument", "insurance",
    "integer", "intense", "intent", "interview",
    "invest", "invite", "iron",
    "discover", "discuss", "display", "distance", "distinct",
    "distribute", "district", "dinner", "direct", "differ",
    "matter", "mission", "minute", "million", "minister",
    "minor", "minus", "middle", "military", "mistake",
    "desert", "design", "desire", "desk", "detail", "detect",
    "develop", "device", "demand", "democracy", "demonstrate",
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
    "depend", "deploy", "describe", "despite", "destroy",
    "determine", "delay", "deliver", "defend", "defeat",
    "decline", "declare", "decide", "debate", "degree", "deep",
    "super", "sugar", "suffer", "summer", "summer", "sudden",
    "submit", "success", "support", "suppose", "supply",
    "surprise", "surgery", "surface", "survive", "suspect",
    "energy", "engine", "enjoy", "enough", "ensure", "enter",
    "entire", "entry", "enemy", "engage", "enhance",
    "enormous", "enterprise", "entertain", "environment",
    "episode", "equal", "error", "escape", "essay", "essential",
    "establish", "estate", "estimate", "ethnic", "evaluate",
    "even", "event", "ever", "every", "evidence", "evil",
    "exact", "exam", "example", "excellent", "except", "exchange",
    "execute", "exercise", "exhibit", "exist", "expand", "expect",
    "expense", "experience", "experiment", "expert", "explain",
    "explore", "export", "expose", "express", "extend", "extra",
    "extreme", "eye",
    "cover", "corner", "counter", "country", "couple", "course",
    "cousin", "comment", "commit", "common", "community",
    "company", "compare", "compete", "complete", "complex",
    "conduct", "confirm", "congress", "connect", "consider",
    "consist", "constant", "construct", "consult", "consume",
    "contain", "content", "context", "continue", "contract",
    "control", "convert", "convince", "correct", "corrupt",
    "pretty", "present", "president", "press", "pressure",
    "prevent", "previous", "price", "primary", "prince",
    "principle", "print", "prior", "prison", "private",
    "probably", "problem", "process", "produce", "product",
    "professor", "profit", "program", "project", "promise",
    "promote", "proper", "property", "propose", "protect",
    "protest", "prove", "provide", "public",
    "other", "over", "out",
    "forest", "forget", "form", "former", "formula", "forth",
    "fortune", "forward", "found", "foundation", "four",
    "foreign", "forever", "formal",
}

# Minimum root length after stripping (brain constraint: roots are >= 3 chars)
MIN_ROOT_LEN = 3

# Maximum number of affixes to strip (brain processes 1-2 prefixes, 1-3 suffixes)
MAX_PREFIXES = 2
MAX_SUFFIXES = 3


def normalize_stem(stem: str) -> List[str]:
    """
    Generate candidate canonical roots from a surface stem.
    Handles common English morphophonemic alternations:
    - Consonant doubling: "runn" → "run"
    - E-dropping: "mak" → "make"
    - Y→I change: "happi" → "happy"
    - IE→Y change: "dri" → "dry"
    """
    candidates = [stem]
    if len(stem) < 2:
        return candidates

    # Consonant doubling: "runn" → "run", "sitt" → "sit"
    if len(stem) >= 3 and stem[-1] == stem[-2] and stem[-1].isalpha():
        candidates.append(stem[:-1])

    # E-dropping: "mak" → "make", "hop" → "hope"
    candidates.append(stem + "e")

    # Y→I: "happi" → "happy", "easi" → "easy"
    if stem.endswith("i"):
        candidates.append(stem[:-1] + "y")

    # IE→Y: "dri" → "dry" (before -ed, -er, -es)
    if stem.endswith("i"):
        candidates.append(stem[:-1] + "y")

    # F→V: "calv" → "calf" (halves → half)
    if stem.endswith("v"):
        candidates.append(stem[:-1] + "f")
        candidates.append(stem[:-1] + "fe")

    return candidates


def decompose_word(
    word: str,
    known_words: Optional[Set[str]] = None,
    min_root: int = MIN_ROOT_LEN,
) -> List[Morpheme]:
    """
    Decompose a word into morphemes using greedy affix stripping.

    Fast greedy algorithm (following Taft 2004):
    1. Strip suffixes (right-to-left, longest match first, one at a time)
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
    2. At each step, check if remainder is a known root → stop if yes
    3. Strip prefixes (left-to-right, longest match first)
    4. Validate final root against known_words (with morphophonemic normalization)
    5. Return surface-form segments that concatenate to original word

    Complexity: O(|SUFFIXES| × MAX_SUFFIXES + |PREFIXES| × MAX_PREFIXES) per word
    """
    if not word or not word.isalpha():
        return [Morpheme(surface=word, mtype=MorphemeType.WORD)]

    lower = word.lower()

    # Check false decomposition list
    if lower in FALSE_DECOMPOSITION_PREFIXES:
        return [Morpheme(surface=word, mtype=MorphemeType.WORD)]

    # If word is very short (<= 3 chars), don't decompose
    if len(lower) <= 3:
        return [Morpheme(surface=word, mtype=MorphemeType.WORD)]

    #        Smart suffix stripping: prefer decompositions with known roots
    suffix_surfaces = []
    remaining = lower
    for _ in range(MAX_SUFFIXES):
        # Try all matching suffixes and score by root quality
        candidates = []
        for suffix in SUFFIXES:
            if remaining.endswith(suffix) and len(remaining) - len(suffix) >= min_root:
                rem = remaining[:-len(suffix)]
                # Three-tier scoring: direct match (2) > normalized match (1) > unknown (0)
                # Known roots: prefer longer roots (preserves more meaning)
                # Unknown roots: prefer longer suffixes (shorter remainder is easier
                # to decompose further with prefixes)
                quality = _root_quality(rem, known_words) if known_words else 0
                score = quality * 1000 + (len(rem) if quality > 0 else len(suffix))
                candidates.append((suffix, rem, score))

        if not candidates:
            break

        # Pick the candidate with the best score (known root > longer root)
        candidates.sort(key=lambda x: -x[2])
        best_suffix, best_rem, best_score = candidates[0]
        suffix_surfaces.insert(0, best_suffix)
        remaining = best_rem

        # Early exit: if remainder is a known word, stop stripping
        if known_words and _is_known_root(remaining, known_words):
            break

    #        Greedy prefix stripping
    prefix_surfaces = []
    for _ in range(MAX_PREFIXES):
        best_prefix = None
        for prefix in PREFIXES:
            if remaining.startswith(prefix) and len(remaining) - len(prefix) >= min_root:
                if best_prefix is None or len(prefix) > len(best_prefix):
                    best_prefix = prefix
        if best_prefix is None:
            break

        # Check: does the word after this prefix look like a real root?
        candidate_root = remaining[len(best_prefix):]
        root_known = known_words and _is_known_root(candidate_root, known_words)

        if not root_known:
            # For novel words with suffixes already stripped, allow ONE prefix if:
            # 1. Remainder is long enough (>=4 chars)
            # 2. Prefix is meaningful (>=3 chars -- avoids spurious "co", "re", "de" splits)
            #    OR if there are no prefixes yet (first prefix gets lenient treatment)
            if len(suffix_surfaces) == 0 or len(candidate_root) < 4:
                break
            if len(prefix_surfaces) >= 1:
                # Already have one prefix -- don't stack another on unknown root
                break
            if len(best_prefix) < 3 and len(candidate_root) < 5:
                # Short prefix on short unknown root is likely spurious
                break

        prefix_surfaces.append(best_prefix)
        remaining = candidate_root

    root_surface = remaining
    n_affixes = len(prefix_surfaces) + len(suffix_surfaces)

    #        Validate decomposition
    if n_affixes == 0:
        return [Morpheme(surface=word, mtype=MorphemeType.WORD)]

    # Check: is the base form (root without any affixes) in false decomposition list?
    # This catches "discovered" → "discover" is a false decomposition base
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
    base_form = lower
    for s in suffix_surfaces:
        if base_form.endswith(s):
            base_form = base_form[:-len(s)]
    # Also try morphophonemic normalization of the base
    base_candidates = [base_form] + normalize_stem(base_form)
    for bc in base_candidates:
        if bc in FALSE_DECOMPOSITION_PREFIXES:
            return [Morpheme(surface=word, mtype=MorphemeType.WORD)]

    decomp_valid = False
    if known_words:
        if _is_known_root(root_surface, known_words):
            decomp_valid = True
        elif len(root_surface) >= 4 and n_affixes >= 1:
            decomp_valid = True
    else:
        decomp_valid = len(root_surface) >= 4

    if not decomp_valid:
        return [Morpheme(surface=word, mtype=MorphemeType.WORD)]

    #        Build morpheme list with original case preserved
    morphemes = []
    pos = 0
    for p in prefix_surfaces:
        morphemes.append(Morpheme(
            surface=word[pos:pos + len(p)],
            mtype=MorphemeType.PREFIX,
            canonical=p,
        ))
        pos += len(p)
    morphemes.append(Morpheme(
        surface=word[pos:pos + len(root_surface)],
        mtype=MorphemeType.ROOT,
        canonical=root_surface,
    ))
    pos += len(root_surface)
    for s in suffix_surfaces:
        morphemes.append(Morpheme(
            surface=word[pos:pos + len(s)],
            mtype=MorphemeType.SUFFIX,
            canonical=s,
        ))
        pos += len(s)

    return morphemes


def _is_known_root(stem: str, known_words: Set[str]) -> bool:
    """Check if a stem (or its morphophonemic variant) is a known word."""
    if stem in known_words:
        return True
    for candidate in normalize_stem(stem):
        if candidate in known_words:
            return True
    return False


def _root_quality(stem: str, known_words: Set[str]) -> int:
    """Score how well a stem matches the known word set.

    Returns:
        2 -- direct match (stem itself is in known_words)
        1 -- normalized match (morphophonemic variant is in known_words)
        0 -- not known
    """
    if stem in known_words:
        return 2
    for candidate in normalize_stem(stem):
        if candidate in known_words:
            return 1
    return 0


def batch_decompose(
    words: List[str],
    known_words: Optional[Set[str]] = None,
) -> Dict[str, List[Morpheme]]:
    """
    Decompose a batch of words. Returns word → morpheme list mapping.
    """
    results = {}
    for word in words:
        results[word] = decompose_word(word, known_words)
    return results
```

Listing 1: Morphological Decomposition Engine — complete source code

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

3 NCT Morphological Tokenizer

Source: `nct_tokenizer/tokenizer.py` **Lines:** 695

3.1 Neuroscience Motivation

The NCT tokenizer implements the **Dual Route Cascaded (DRC)** model of visual word recognition (Coltheart et al., 2001), the most comprehensive computational model of how the brain reads words. In the DRC model, two parallel processing routes compete to identify a word:

Route 1 — Lexical (direct): High-frequency words are recognized as wholes via direct lookup in the orthographic input lexicon. Words like "the," "and," "is" bypass any form of decomposition. This route is fast (∼150ms from fixation to lexical access) but limited to words already stored in the mental lexicon. Functional MRI studies show that this route activates the visual word form area (VWFA) in the left fusiform gyrus.

Route 2 — Sublexical (morphological): Unknown or low-frequency words are processed through grapheme-to-phoneme conversion rules and morphological decomposition. "Unhappiness" is decomposed into "un-" + "happy" + "-ness" via the morphology engine described in section 2. This route is slower (∼250ms) but generalizes to novel words and non-words. It relies on left inferior frontal regions (Broca's area) for assembly.

Route 3 — Byte fallback: A safety net not present in the biological DRC model but necessary for computational universality. Any character sequence that cannot be processed by Routes 1 or 2 is encoded byte-by-byte using `<byte:0>` through `<byte:255>` tokens. This guarantees lossless coverage for arbitrary Unicode input.

Oscillatory Alignment

Each NCT token corresponds to approximately one gamma cycle (≈25ms ≈ one morpheme). This creates a natural alignment between the tokenization granularity and the cortical column's oscillatory hierarchy:

- **Token** ≈ 1 gamma cycle ≈ 1 morpheme
- **Word** ≈ 1 theta cycle (8–16 tokens) ≈ 3–7 morphemes
- **Phrase** ≈ 1 alpha cycle (∼20 tokens) ≈ 8–15 morphemes

The L2/3 windowed Hebbian association (window $w = 16$ tokens $= 2$ theta cycles) thus operates at the word/phrase boundary, consistent with how the brain binds lexical items into phrases via theta-gamma phase-amplitude coupling.

3.2 Vocabulary Architecture

The vocabulary is organized into seven contiguous regions, ordered from most fundamental (special tokens, byte fallback) to most language-specific (whole words):

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

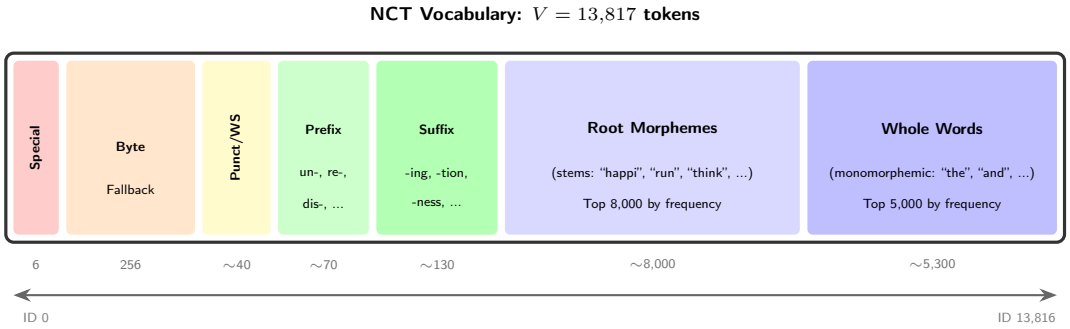


Figure 4: NCT vocabulary structure. Tokens are organized from most fundamental (left) to most language-specific (right). Root morphemes and whole words comprise 96% of the vocabulary.

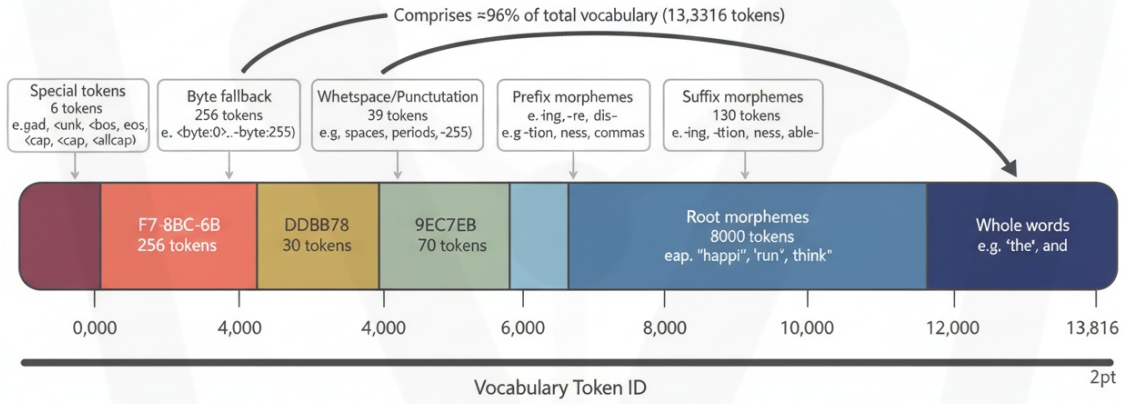


Figure 5: (Illustrated) NCT vocabulary structure. Tokens are organized from most fundamental (left) to most language-specific (right). Root morphemes and whole words comprise 96% of the vocabulary.

3.3 Encoding Process

The encoding algorithm processes each pre-tokenized piece through the dual-route cascade:

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

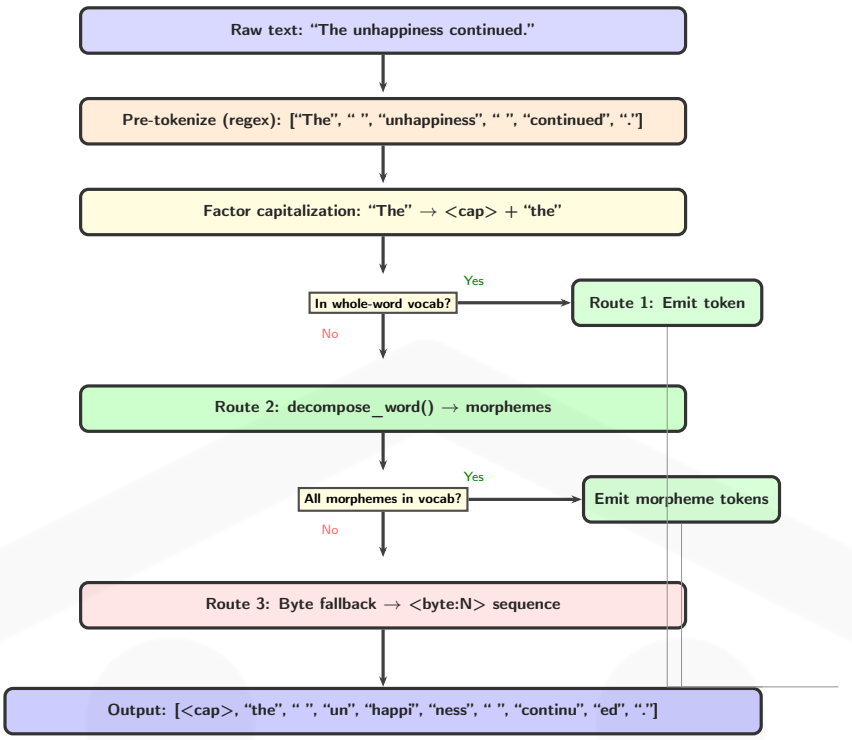


Figure 6: NCT dual-route encoding pipeline. "The" is recognized via Route 1 (lexical lookup). "Unhappiness" is decomposed via Route 2 (sublexical morphology). Unknown characters fall through to Route 3 (byte encoding).

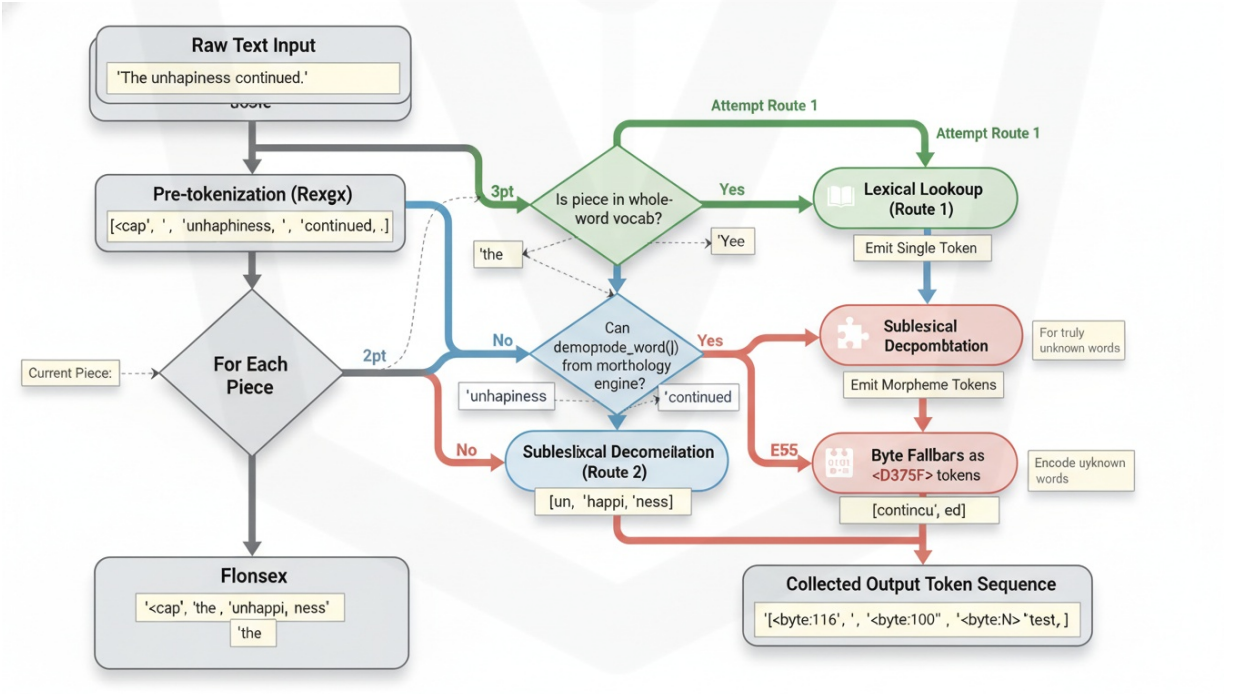


Figure 7: (Illustrated) NCT dual-route encoding pipeline. "The" is recognized via Route 1 (lexical lookup). "Unhappiness" is decomposed via Route 2 (sublexical morphology). Unknown characters fall through to Route 3 (...

3.4 Special Token Design

Six special tokens serve specific functions:

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

Table 2: NCT special tokens.

Token	ID	Function
<pad>	0	Padding for batch alignment
<unk>	1	Unknown token (should never appear — byte fallback covers everything)
<bos>	2	Beginning of sequence
<eos>	3	End of sequence
<cap>	4	Next token's first character is uppercase
<allcap>	5	Next token is ALL UPPERCASE

The capitalization factoring (<cap> and <allcap>) is a key design choice: it allows the same morpheme token to represent both "Happy" and "happy," reducing vocabulary size by ∼30% compared to case-sensitive tokenizers while preserving capitalization information for proper noun detection and sentence boundaries.

3.5 Comparison with BPE

Table 3: NCT morphological tokenizer vs. BPE tokenizers (GPT-4, LLaMA).

Property	NCT Morphological	BPE (GPT/LLaMA)
Vocab size	13,817	32,000–100,000
Split unit	Morpheme (linguistic)	Byte pair (statistical)
Cross-word merges	Never	Common
Mid-morpheme splits	Never	Common
Biological basis	DRC model (Coltheart 2001)	None
Lossless	Yes (byte fallback)	Yes (byte fallback)
Deterministic	Yes	Yes
Oscillatory alignment	1 token ≈ 1 gamma cycle	No alignment
Coverage of novel words	Decompose → morphemes → byte	Byte fallback only

3.6 Vocabulary Construction Process

Building the vocabulary from a corpus is a five-step pipeline that mirrors how a child acquires vocabulary — first learning common whole words, then discovering morphological patterns:

Step 1 — Pre-tokenize and count: The regex pre-tokenizer splits the entire corpus into word pieces. Word frequencies are counted (case-insensitive for alphabetic words), producing a frequency distribution over the vocabulary. Punctuation, whitespace, and digits are counted separately.

Step 2 — Build known-words set: Words appearing ≥ 5 times form the known-words set $\mathcal{W}$. This set serves as the ground truth for root validation during morphological decomposition. Typically $|\mathcal{W}| \approx$ 10,000–30,000 for a corpus like WikiText-2.

Step 3 — Morphological decomposition: Every word is decomposed using the morphology engine (section 2). Morpheme frequencies are counted by type (prefix, suffix, root). Words that decompose into multiple morphemes contribute to all their constituent morpheme frequencies. Monomorphemic words (no valid decomposition) are counted as whole words.

Step 4 — Construct vocabulary in priority order: Tokens are assigned IDs in a fixed order: special tokens (6) → byte fallback (256) → whitespace/punctuation (variable) → contraction pieces ("'t", "'s", etc.) → digit tokens → prefix morphemes (all productive, freq ≥ 3) → suffix morphemes (all productive, freq ≥ 3) → root morphemes (top 8,000 by frequency) → whole words (top 5,000 monomorphemic by frequency). This ordering ensures that affixes always have lower IDs than roots/words, which can be exploited for efficient morphological type checking.

Step 5 — Build decomposition cache: For each word in the corpus, if all its morpheme segments are present in the vocabulary, the decomposition is cached for $O(1)$ lookup during encoding. Words whose morphemes are not all in the vocabulary fall through to byte encoding at runtime.

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

3.7 Source Code

```
# ================================================================================
# Tokenizer -- NCT Brain-Inspired Morphological Tokenizer
# Source: nct_tokenizer/tokenizer.py
# ================================================================================

"""
tokenizer.py -- NCT Brain-Inspired Morphological Tokenizer

Dual-route architecture (Coltheart et al. 2001 DRC model):
  Route 1 (Lexical): Whole-word lookup for high-frequency words
  Route 2 (Sublexical): Morphological decomposition via affix stripping
  Fallback: Byte-level encoding for truly novel/non-alphabetic input

Design constraints:
  - Lossless: encode → decode = identity
  - Deterministic: same text → same tokens always
  - Morphologically faithful: never split mid-morpheme
  - Cross-word safe: never merge across word boundaries

Oscillatory alignment (Giraud & Poeppel 2012):
  - Each token ~ 1 gamma cycle (1-2T) ~ 1 morpheme
  - Words span theta cycles (8-16T) ~ 3-7 morphemes
  - Phrases span alpha cycles (~20T) ~ 8-15 morphemes
"""


# [bundled] from .morphology import (


#        Pre-tokenization regex

# Splits text into: contractions, words, numbers, punctuation, whitespace
# Contractions split at apostrophe: "don" + "'t", "she" + "'s", "they" + "'re"
# Unicode letters supported for accented chars (café, résumé)
PRETOK_PATTERN = re.compile(
    r"""'s|'t|'re|'ve|'m|'ll|'d"""   # contractions
    r"""|[A-Za-z\u00C0-\u024F\u1E00-\u1EFF]+"""  # words (ASCII + Latin Extended)
    r"""|\d+"""                        # numbers
    r"""|\s"""                         # single whitespace character
    r"""|[^\s]""",                     # single non-whitespace character
    re.VERBOSE,
)

# Special tokens
PAD_TOKEN = "<pad>"
UNK_TOKEN = "<unk>"
BOS_TOKEN = "<bos>"
EOS_TOKEN = "<eos>"
CAP_TOKEN = "<cap>"       # next token's first char is uppercase
ALLCAP_TOKEN = "<allcap>" # next token is ALL UPPERCASE

SPECIAL_TOKENS = [PAD_TOKEN, UNK_TOKEN, BOS_TOKEN, EOS_TOKEN, CAP_TOKEN, ALLCAP_TOKEN]
NUM_SPECIAL = len(SPECIAL_TOKENS)

# Byte tokens: <byte:0> through <byte:255>
NUM_BYTES = 256


class Encoding:
    """Minimal encoding result compatible with HuggingFace tokenizers API."""
    __slots__ = ("ids", "tokens", "morpheme_types")

    def __init__(self, ids: List[int], tokens: List[str],
                 morpheme_types: Optional[List[str]] = None):
        self.ids = ids
        self.tokens = tokens
        self.morpheme_types = morpheme_types


class NCTTokenizer:
    """
    Brain-inspired morphological tokenizer for NCT.

    Vocabulary structure:
      [0..3]          Special tokens: <pad>, <unk>, <bos>, <eos>
      [4..259]        Byte fallback tokens: <byte:0>..<byte:255>
      [260..]         Morpheme/word tokens (sorted by type then frequency)
                      - Whitespace/punctuation tokens
                      - Prefix tokens
                      - Suffix tokens
                      - Root morpheme tokens
                      - Whole-word tokens (high-frequency, monomorphemic)
    """

    def __init__(self):
        # Token → ID mapping
        self.token_to_id: Dict[str, int] = {}
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        # ID → token mapping
        self.id_to_token: Dict[int, str] = {}
        # Token → morpheme type
        self.token_types: Dict[str, str] = {}
        # Pre-computed decomposition cache (word → list of token strings)
        self.decomposition_cache: Dict[str, List[str]] = {}
        # Set of known words for morphological validation
        self.known_words: Set[str] = set()
        # Vocabulary size
        self.vocab_size: int = 0

        # Initialize special tokens and byte fallback
        self._init_base_vocab()

    def _init_base_vocab(self):
        """Initialize special tokens and byte-level fallback."""
        idx = 0

        # Special tokens
        for tok in SPECIAL_TOKENS:
            self.token_to_id[tok] = idx
            self.id_to_token[idx] = tok
            self.token_types[tok] = "special"
            idx += 1

        # Byte fallback tokens
        for b in range(NUM_BYTES):
            tok = f"<byte:{b}>"
            self.token_to_id[tok] = idx
            self.id_to_token[idx] = tok
            self.token_types[tok] = "byte"
            idx += 1

        self.vocab_size = idx

    def build_vocab(
        self,
        corpus_text: str,
        max_roots: int = 8000,
        max_whole_words: int = 5000,
        min_word_freq: int = 5,
        min_morpheme_freq: int = 3,
    ):
        """
        Build vocabulary from a text corpus.

        Algorithm:
        1. Pre-tokenize corpus into words
        2. Count word frequencies → known_words set
        3. Decompose each word morphologically
        4. Count morpheme frequencies
        5. Build vocabulary: special + bytes + punctuation + affixes + roots + whole words
        """
        print("[NCTTokenizer] Building vocabulary from corpus...")

        # Step 1: Pre-tokenize and count word frequencies
        pieces = PRETOK_PATTERN.findall(corpus_text)
        word_freq = Counter()
        punct_freq = Counter()
        ws_freq = Counter()

        for piece in pieces:
            if piece.isalpha():
                word_freq[piece.lower()] += 1
            elif piece.isdigit():
                word_freq[piece] += 1  # numbers as-is
            elif piece.isspace():
                ws_freq[piece] += 1
            else:
                punct_freq[piece] += 1

        print(f"  Unique words: {len(word_freq):,}")
        print(f"  Total word tokens: {sum(word_freq.values()):,}")

        # Step 2: Build known_words set (words appearing >= min_word_freq times)
        self.known_words = {w for w, c in word_freq.items()
                            if c >= min_word_freq and w.isalpha()}
        print(f"  Known words (freq>={min_word_freq}): {len(self.known_words):,}")

        # Step 3: Decompose all words
        print("  Decomposing words morphologically...")
        decompositions = {}
        morpheme_freq = Counter()
        prefix_freq = Counter()
        suffix_freq = Counter()
        root_freq = Counter()
        whole_word_freq = Counter()

        for word, freq in word_freq.most_common():
            if not word.isalpha():
                # Numbers and other non-alpha → treat as whole tokens
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
                whole_word_freq[word] += freq
                decompositions[word] = [word]
                continue

            morphemes = decompose_word(word, self.known_words)

            if len(morphemes) == 1 and morphemes[0].mtype == MorphemeType.WORD:
                # Monomorphemic word
                whole_word_freq[word] += freq
                decompositions[word] = [word]
            else:
                # Successfully decomposed
                surfaces = [m.surface.lower() for m in morphemes]
                decompositions[word] = surfaces
                for m in morphemes:
                    s = m.surface.lower()
                    morpheme_freq[s] += freq
                    if m.mtype == MorphemeType.PREFIX:
                        prefix_freq[s] += freq
                    elif m.mtype == MorphemeType.SUFFIX:
                        suffix_freq[s] += freq
                    elif m.mtype == MorphemeType.ROOT:
                        root_freq[s] += freq

        n_decomposed = sum(1 for d in decompositions.values() if len(d) > 1)
        print(f"  Decomposed: {n_decomposed:,} words")
        print(f"  Monomorphemic: {len(decompositions) - n_decomposed:,} words")
        print(f"  Unique prefixes found: {len(prefix_freq)}")
        print(f"  Unique suffixes found: {len(suffix_freq)}")
        print(f"  Unique roots found: {len(root_freq)}")

        # Step 4: Build vocabulary
        idx = self.vocab_size  # start after special + byte tokens

        # 4a. Whitespace tokens
        for ws_tok in [" ", "\n", "\t", "\r"]:
            if ws_tok not in self.token_to_id:
                self.token_to_id[ws_tok] = idx
                self.id_to_token[idx] = ws_tok
                self.token_types[ws_tok] = "whitespace"
                idx += 1

        # 4b. Contraction pieces (pre-tokenizer splits "shouldn't" → "shouldn" + "'t")
        contraction_stems = [
            "shouldn", "wouldn", "couldn", "doesn", "hasn", "hadn",
            "wasn", "weren", "aren", "isn", "didn", "mustn", "needn",
        ]
        contraction_tails = ["'t", "'s", "'re", "'ve", "'m", "'ll", "'d"]
        for ct in contraction_stems + contraction_tails:
            if ct not in self.token_to_id:
                self.token_to_id[ct] = idx
                self.id_to_token[idx] = ct
                self.token_types[ct] = "contraction"
                idx += 1

        # 4c. Punctuation tokens
        punct_list = sorted(punct_freq.keys(), key=lambda x: -punct_freq[x])
        for p in punct_list:
            if p not in self.token_to_id:
                self.token_to_id[p] = idx
                self.id_to_token[idx] = p
                self.token_types[p] = "punctuation"
                idx += 1

        # Individual digit tokens (0-9)
        for d in "0123456789":
            if d not in self.token_to_id:
                self.token_to_id[d] = idx
                self.id_to_token[idx] = d
                self.token_types[d] = "digit"
                idx += 1

        # Common multi-digit numbers
        for n in range(10, 100):
            ns = str(n)
            if word_freq.get(ns, 0) >= min_word_freq and ns not in self.token_to_id:
                self.token_to_id[ns] = idx
                self.id_to_token[idx] = ns
                self.token_types[ns] = "number"
                idx += 1

        # 4c. Prefix tokens (all productive prefixes in the affix inventory)
        for prefix, freq in prefix_freq.most_common():
            if freq >= min_morpheme_freq and prefix not in self.token_to_id:
                self.token_to_id[prefix] = idx
                self.id_to_token[idx] = prefix
                self.token_types[prefix] = "prefix"
                idx += 1

        # 4d. Suffix tokens
        for suffix, freq in suffix_freq.most_common():
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
            if freq >= min_morpheme_freq and suffix not in self.token_to_id:
                self.token_to_id[suffix] = idx
                self.id_to_token[idx] = suffix
                self.token_types[suffix] = "suffix"
                idx += 1

        # 4e. Root morpheme tokens (top max_roots by frequency)
        added_roots = 0
        for root, freq in root_freq.most_common():
            if added_roots >= max_roots:
                break
            if freq >= min_morpheme_freq and root not in self.token_to_id:
                self.token_to_id[root] = idx
                self.id_to_token[idx] = root
                self.token_types[root] = "root"
                idx += 1
                added_roots += 1

        # 4f. Whole-word tokens (top max_whole_words by frequency)
        added_words = 0
        for word, freq in whole_word_freq.most_common():
            if added_words >= max_whole_words:
                break
            if freq >= min_word_freq and word not in self.token_to_id:
                self.token_to_id[word] = idx
                self.id_to_token[idx] = word
                self.token_types[word] = "word"
                idx += 1
                added_words += 1

        self.vocab_size = idx

        # Step 5: Build decomposition cache
        # Only cache decompositions where ALL segments are in vocabulary
        for word, segments in decompositions.items():
            lower_segs = [s.lower() for s in segments]
            if all(s in self.token_to_id for s in lower_segs):
                self.decomposition_cache[word] = lower_segs

        print(f"\n  Vocabulary size: {self.vocab_size:,}")
        print(f"    Special: {NUM_SPECIAL}")
        print(f"    Byte fallback: {NUM_BYTES}")
        n_ws = sum(1 for t in self.token_types.values() if t == "whitespace")
        n_punct = sum(1 for t in self.token_types.values() if t in ("punctuation", "digit"))
        n_prefix = sum(1 for t in self.token_types.values() if t == "prefix")
        n_suffix = sum(1 for t in self.token_types.values() if t == "suffix")
        n_root = sum(1 for t in self.token_types.values() if t == "root")
        n_word = sum(1 for t in self.token_types.values() if t == "word")
        print(f"    Whitespace: {n_ws}")
        print(f"    Punctuation/digits: {n_punct}")
        print(f"    Prefixes: {n_prefix}")
        print(f"    Suffixes: {n_suffix}")
        print(f"    Roots: {n_root}")
        print(f"    Whole words: {n_word}")
        print(f"  Decomposition cache: {len(self.decomposition_cache):,} words")

    def encode(self, text: str, skip_whitespace: bool = False) -> Encoding:
        """
        Encode text into token IDs.

        Algorithm:
        1. Unicode normalize (NFC)
        2. Pre-tokenize into words, punctuation, whitespace
        3. For each piece:
           a. Whitespace/punctuation → direct lookup
           b. Word → decomposition cache lookup (lexical route)
           c. Word not cached → morphological decomposition (sublexical route)
           d. Still unknown → byte-level fallback

        Args:
            skip_whitespace: If True, omit whitespace tokens from output.
                Use True for training data (compact sequences, ~38% fewer tokens).
                Use False for interactive use (lossless roundtrip).

        Returns:
            Encoding object with .ids (List[int]) and .tokens (List[str])
        """
        text = unicodedata.normalize("NFC", text)
        pieces = PRETOK_PATTERN.findall(text)

        ids = []
        tokens = []
        mtypes = []

        for piece in pieces:
            if skip_whitespace and piece.isspace():
                continue
            piece_ids, piece_toks, piece_types = self._encode_piece(piece)
            ids.extend(piece_ids)
            tokens.extend(piece_toks)
            mtypes.extend(piece_types)
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
361
362         return Encoding(ids, tokens, mtypes)
363
364     def _encode_piece(self, piece: str) -> Tuple[List[int], List[str], List[str]]:
365         """Encode a single pre-tokenized piece."""
366         # Direct lookup (punctuation, whitespace, known whole tokens)
367         if piece in self.token_to_id:
368             tid = self.token_to_id[piece]
369             return [tid], [piece], [self.token_types.get(piece, "unknown")]
370
371         # Detect case pattern and work with lowercase for morpheme lookup
372         lower = piece.lower()
373         case_type = self._detect_case(piece)
374
375         # Mixed case (e.g., "MiXeD") -- byte encode to preserve exact casing
376         if case_type == "mixed":
377             return self._encode_bytes(piece)
378
379         # Direct lookup with lowercase
380         if lower in self.token_to_id:
381             ids, toks, types = [self.token_to_id[lower]], [lower], [self.token_types.get(lower, "unknown")]
382             return self._apply_case_marker(ids, toks, types, case_type)
383
384         # Decomposition cache (fast path -- lexical route)
385         if lower in self.decomposition_cache:
386             segments = self.decomposition_cache[lower]
387             result_ids = []
388             result_toks = []
389             result_types = []
390             for seg in segments:
391                 if seg in self.token_to_id:
392                     result_ids.append(self.token_to_id[seg])
393                     result_toks.append(seg)
394                     result_types.append(self.token_types.get(seg, "unknown"))
395                 else:
396                     byte_ids, byte_toks, byte_types = self._encode_bytes(seg)
397                     result_ids.extend(byte_ids)
398                     result_toks.extend(byte_toks)
399                     result_types.extend(byte_types)
400             return self._apply_case_marker(result_ids, result_toks, result_types, case_type)
401
402         # Sublexical route: try morphological decomposition at runtime
403         if lower.isalpha() and len(lower) > 3:
404             morphemes = decompose_word(lower, self.known_words)
405             if len(morphemes) > 1:
406                 result_ids = []
407                 result_toks = []
408                 result_types = []
409                 n_in_vocab = 0
410                 for m in morphemes:
411                     seg = m.surface.lower()
412                     if seg in self.token_to_id:
413                         result_ids.append(self.token_to_id[seg])
414                         result_toks.append(seg)
415                         result_types.append(m.mtype.value)
416                         n_in_vocab += 1
417                     else:
418                         # Byte-encode unknown segments (partial decomposition)
419                         byte_ids, byte_toks, byte_types = self._encode_bytes(seg)
420                         result_ids.extend(byte_ids)
421                         result_toks.extend(byte_toks)
422                         result_types.extend(byte_types)
423
424                 # Use partial decomposition if at least one affix was found in vocab
425                 if n_in_vocab >= 1:
426                     return self._apply_case_marker(
427                         result_ids, result_toks, result_types, case_type)
428
429         # For digit sequences: try splitting into known number tokens
430         if piece.isdigit():
431             return self._encode_digits(piece)
432
433         # Byte-level fallback (preserves case naturally)
434         return self._encode_bytes(piece)
435
436     def _encode_digits(self, digits: str) -> Tuple[List[int], List[str], List[str]]:
437         """Encode a digit sequence using number tokens with greedy matching."""
438         ids, toks, types = [], [], []
439         i = 0
440         while i < len(digits):
441             # Try 2-digit, then 1-digit
442             matched = False
443             for length in (2, 1):
444                 if i + length <= len(digits):
445                     chunk = digits[i:i+length]
446                     if chunk in self.token_to_id:
447                         ids.append(self.token_to_id[chunk])
448                         toks.append(chunk)
449                         types.append(self.token_types.get(chunk, "digit"))
450                         i += length
451                         matched = True
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
                        break
            if not matched:
                # Single byte fallback
                b_ids, b_toks, b_types = self._encode_bytes(digits[i])
                ids.extend(b_ids)
                toks.extend(b_toks)
                types.extend(b_types)
                i += 1
        return ids, toks, types

    def _detect_case(self, piece: str) -> str:
        """Detect the case pattern of a text piece."""
        if not piece or not piece[0].isalpha():
            return "lower"
        if piece.islower():
            return "lower"
        if piece.isupper() and len(piece) > 1:
            return "upper"
        if piece[0].isupper() and (len(piece) == 1 or piece[1:].islower()):
            return "title"
        # Arbitrary mixed case (e.g., "MiXeD") -- cannot be recovered from lowercase
        return "mixed"

    def _apply_case_marker(
        self,
        ids: List[int],
        toks: List[str],
        types: List[str],
        case_type: str,
    ) -> Tuple[List[int], List[str], List[str]]:
        """Prepend case marker token if needed."""
        if case_type == "title":
            cap_id = self.token_to_id[CAP_TOKEN]
            return [cap_id] + ids, [CAP_TOKEN] + toks, ["special"] + types
        elif case_type == "upper":
            allcap_id = self.token_to_id[ALLCAP_TOKEN]
            return [allcap_id] + ids, [ALLCAP_TOKEN] + toks, ["special"] + types
        return ids, toks, types

    def _encode_bytes(self, text: str) -> Tuple[List[int], List[str], List[str]]:
        """Encode text as byte tokens (fallback for unknown tokens)."""
        ids = []
        toks = []
        types = []
        for byte_val in text.encode("utf-8"):
            tok = f"<byte:{byte_val}>"
            tid = self.token_to_id.get(tok, self.token_to_id[UNK_TOKEN])
            ids.append(tid)
            toks.append(tok)
            types.append("byte")
        return ids, toks, types

    def decode(self, ids: List[int]) -> str:
        """
        Decode token IDs back to text.

        Handles:
        - <cap> marker → capitalize first char of next token
        - <allcap> marker → uppercase all chars of next token
        - Byte tokens → decode UTF-8 byte sequences
        - Regular tokens → concatenate strings
        - Special tokens → skip (pad, bos, eos) or replace (unk → ?)
        """
        parts = []
        byte_buffer = []
        pending_case = None  # "title" or "upper"

        for tid in ids:
            tok = self.id_to_token.get(tid, UNK_TOKEN)

            if tok.startswith("<byte:"):
                byte_val = int(tok[6:-1])
                byte_buffer.append(byte_val)
                continue

            # Flush any pending byte buffer
            if byte_buffer:
                try:
                    decoded = bytes(byte_buffer).decode("utf-8", errors="replace")
                except Exception:
                    decoded = "".join(chr(b) if b < 128 else "?" for b in byte_buffer)
                if pending_case == "title":
                    decoded = decoded[0].upper() + decoded[1:] if decoded else decoded
                    pending_case = None
                elif pending_case == "upper":
                    decoded = decoded.upper()
                    pending_case = None
                parts.append(decoded)
                byte_buffer = []

            if tok == CAP_TOKEN:
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
                pending_case = "title"
                continue
            elif tok == ALLCAP_TOKEN:
                pending_case = "upper"
                continue
            elif tok in (PAD_TOKEN, BOS_TOKEN, EOS_TOKEN):
                continue
            elif tok == UNK_TOKEN:
                parts.append("?")
                pending_case = None
            else:
                if pending_case == "title":
                    tok = tok[0].upper() + tok[1:] if tok else tok
                    pending_case = None
                elif pending_case == "upper":
                    tok = tok.upper()
                    pending_case = None
                parts.append(tok)

        # Flush remaining bytes
        if byte_buffer:
            try:
                decoded = bytes(byte_buffer).decode("utf-8", errors="replace")
            except Exception:
                decoded = "".join(chr(b) if b < 128 else "?" for b in byte_buffer)
            if pending_case == "title":
                decoded = decoded[0].upper() + decoded[1:] if decoded else decoded
            elif pending_case == "upper":
                decoded = decoded.upper()
            parts.append(decoded)

        return "".join(parts)

    def save(self, path: str):
        """Save tokenizer to a directory."""
        os.makedirs(path, exist_ok=True)

        # Save vocabulary
        vocab_data = {
            "version": "0.1.0",
            "vocab_size": self.vocab_size,
            "token_to_id": self.token_to_id,
            "token_types": self.token_types,
        }
        with open(os.path.join(path, "vocab.json"), "w", encoding="utf-8") as f:
            json.dump(vocab_data, f, ensure_ascii=False, indent=2)

        # Save decomposition cache (separate file, can be large)
        with open(os.path.join(path, "decompositions.json"), "w", encoding="utf-8") as f:
            json.dump(self.decomposition_cache, f, ensure_ascii=False)

        # Save known words
        with open(os.path.join(path, "known_words.json"), "w", encoding="utf-8") as f:
            json.dump(sorted(self.known_words), f)

        print(f"[NCTTokenizer] Saved to {path}/")
        print(f"  vocab.json: {self.vocab_size} tokens")
        print(f"  decompositions.json: {len(self.decomposition_cache)} entries")
        print(f"  known_words.json: {len(self.known_words)} words")

    @classmethod
    def load(cls, path: str) -> "NCTTokenizer":
        """Load tokenizer from a directory."""
        tok = cls.__new__(cls)
        tok.decomposition_cache = {}
        tok.known_words = set()

        # Load vocabulary
        with open(os.path.join(path, "vocab.json"), "r", encoding="utf-8") as f:
            vocab_data = json.load(f)

        tok.token_to_id = vocab_data["token_to_id"]
        tok.id_to_token = {int(v): k for k, v in tok.token_to_id.items()}
        tok.token_types = vocab_data.get("token_types", {})
        tok.vocab_size = vocab_data["vocab_size"]

        # Load decomposition cache
        decomp_path = os.path.join(path, "decompositions.json")
        if os.path.exists(decomp_path):
            with open(decomp_path, "r", encoding="utf-8") as f:
                tok.decomposition_cache = json.load(f)

        # Load known words
        kw_path = os.path.join(path, "known_words.json")
        if os.path.exists(kw_path):
            with open(kw_path, "r", encoding="utf-8") as f:
                tok.known_words = set(json.load(f))

        print(f"[NCTTokenizer] Loaded from {path}/")
        print(f"  Vocab size: {tok.vocab_size}")
        print(f"  Decomposition cache: {len(tok.decomposition_cache)} entries")
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        print(f"  Known words: {len(tok.known_words)}")

        return tok

    @classmethod
    def from_pretrained(cls, path: str) -> "NCTTokenizer":
        """Alias for load(), matching HuggingFace tokenizers API."""
        return cls.load(path)

    def get_vocab_size(self) -> int:
        return self.vocab_size

    def get_vocab(self) -> Dict[str, int]:
        return dict(self.token_to_id)

    def token_to_morpheme_type(self, token_id: int) -> str:
        """Get the morpheme type for a token ID."""
        tok = self.id_to_token.get(token_id, UNK_TOKEN)
        return self.token_types.get(tok, "unknown")

    def analyze(self, text: str) -> List[dict]:
        """
        Encode text and return detailed analysis of each token.
        Useful for debugging and understanding tokenization quality.
        """
        enc = self.encode(text)
        result = []
        for tid, tok, mtype in zip(enc.ids, enc.tokens, enc.morpheme_types):
            result.append({
                "id": tid,
                "token": tok,
                "type": mtype,
                "repr": repr(tok),
            })
        return result

    def compare_with_bpe(self, text: str):
        """Compare NCT tokenization with GPT-2 BPE for analysis."""
        nct_enc = self.encode(text)
        print(f"Text: {repr(text)}")
        print(f"\nNCT Morpheme Tokenization ({len(nct_enc.ids)} tokens):")
        for tid, tok, mtype in zip(nct_enc.ids, nct_enc.tokens, nct_enc.morpheme_types):
            print(f"  {tid:5d} | {mtype:10s} | {repr(tok)}")

        try:
            from tokenizers import Tokenizer as HFTokenizer
            bpe = HFTokenizer.from_pretrained("gpt2")
            bpe_enc = bpe.encode(text)
            print(f"\nGPT-2 BPE Tokenization ({len(bpe_enc.ids)} tokens):")
            for tid, tok in zip(bpe_enc.ids, bpe_enc.tokens):
                print(f"  {tid:5d} | {'bpe':10s} | {repr(tok)}")
        except ImportError:
            print("\n(GPT-2 BPE comparison not available -- tokenizers not installed)")

    def stats(self) -> dict:
        """Return tokenizer statistics."""
        type_counts = Counter(self.token_types.values())
        return {
            "vocab_size": self.vocab_size,
            "type_counts": dict(type_counts),
            "decomposition_cache_size": len(self.decomposition_cache),
            "known_words": len(self.known_words),
        }
```

Listing 2: NCT Morphological Tokenizer — complete source code

4 V11 Configuration

Source: `nct_v2/v11_config.py` **Lines:** 219

4.1 Design Philosophy

The `V11Config` dataclass is the single source of truth for every hyperparameter in NCT. It serves three purposes: (1) complete specification — every parameter needed to instantiate, train, and evaluate NCT is contained in this single dataclass; (2) neuroscience documentation — every parameter has an inline comment explaining its biological rationale, creating a living record of design decisions; (3) reproducibility — given a `V11Config` instance, the model is fully deterministic (up to random initialization).

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

The dataclass contains 50+ parameters organized into 15 categories matching the brain systems they configure: core dimensions, L2/3 lateral association, predictive coding, interneurons, working memory, neuromodulation, oscillatory dynamics, thalamic routing, hippocampal memory, basal ganglia, wake-sleep cycling, structural plasticity, L5 dendritic computation, learning mode, BCM/metaplasticity, and training regularization.

The critical design constraint: **all default values must produce a working model that fits in 6GB VRAM** (NVIDIA RTX 4050 Laptop GPU). This means the default preset (106M parameters, 12 columns, $d = 512$) must complete a full forward pass including PC inference iterations (up to 5 re-settlings), Hebbian updates to all L2/3 association matrices, working memory state persistence, oscillatory modulation, and wake-sleep consolidation — all within 6GB of GPU memory using bfloat16 mixed precision.

Two computed properties derive from the config: `d_head` ($= d_{\text{col}}/H$, the per-head dimension) and `columns_per_relay` ($= \max(1, N/G)$, how many columns share each thalamic relay group).

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

4.2 Complete Parameter Specification

Table 4: Complete V11Config parameter specification. Every parameter needed to recreate NCT.

Parameter	Type	Default	Neuroscience Rationale
Core Dimensions			
d_model	int	512	Inter-column representation dimension
d_column	int	512	Intra-column processing dimension
n_columns	int	12	Number of hierarchical columns (like transformer layers)
d_ff	int	2048	L5 feedforward expansion ($4 \times d$)
vocab_size	int	14000	NCT morphological tokenizer ($V \approx 13{,}817$)
max_seq_len	int	1024	Maximum sequence length
L2/3 Lateral Association (Theorem B)			
n_assoc_heads	int	8	Association heads (analogous to attention heads)
n_settle_steps	int	1	Competitive settling iterations (> 1 = recurrent)
k_wta_ratio	float	0.15	k-WTA sparsity: top 15% active
hebbian_lr	float	10^{-4}	Hebbian learning rate for W_{assoc}
assoc_dropout	float	0.1	Dropout on association weights
assoc_window_size	int	16	Causal window: 16 tokens = 2 theta cycles
Predictive Coding (Theorem A)			
pc_min_steps	int	2	Minimum PC inference iterations
pc_max_steps	int	5	Maximum PC inference iterations
pc_convergence_threshold	float	0.01	Relative error change threshold for early stop
sigma2_init	float	1.0	Initial prediction precision (σ^2)
pc_learning_lr	float	3×10^{-4}	PC weight update learning rate
Interneurons			
d_inhibitory	int	128	Inhibitory interneuron dimension
vip_enabled	bool	True	VIP disinhibition gate
sst_enabled	bool	True	SST dendritic inhibition
pv_enabled	bool	True	PV/Chandelier output veto
Working Memory			
nmda_slots	int	4	NMDA attractor slots (Luck & Vogel: $\sim$4)
nmda_d_slot	int	64	Per-slot dimension
nmda_decay_init	float	6.0	$\sigma(6) \approx 0.9975$, half-life ≈ 277 tokens
silent_d_trace	int	64	Activity-silent trace dimension
phono_capacity	int	8	Phonological loop slots (Miller's 7 ± 2)
Neuromodulation (Theorem C)			
da_enabled	bool	True	Dopamine: reward prediction error
ne_enabled	bool	True	Norepinephrine: arousal/exploration
serotonin_enabled	bool	True	Serotonin: time horizon/patience
ach_enabled	bool	True	Acetylcholine: attention/plasticity gate
tonic_da_ema	float	0.999	EMA decay for tonic DA baseline
Oscillatory Dynamics			
gamma_period	int	1	Gamma cycle (1 token)
theta_period	int	8	Theta cycle (8 tokens)
alpha_period	int	20	Alpha cycle (20 tokens)
delta_period	int	64	Delta cycle (64 tokens)
Thalamus			
n_relay_groups	int	4	Relay groups ($\sim$1 per 3 columns)
d_relay	int	256	Relay intermediate dimension
d_trn	int	24	TRN gating dimension
Hippocampus			
hpc_memory_size	int	2048	Max episodic memories in buffer
hpc_d_sparse	int	2048	DG sparse expansion dimension ($4d$)
hpc_surprise_threshold	float	0.5	Surprise threshold for tagging
Basal Ganglia			
d1_ec50 / d1_slope	float	0.65 / 8.0	D1 receptor threshold / steepness
d2_ec50 / d2_slope	float	0.25 / 8.0	D2 receptor threshold / steepness

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

4.3 Presets

Three presets scale the architecture for different hardware:

Theorem 4.1 (Preset Scaling)**.** *All presets maintain the following invariants:*

- $d_{ff} = 4 \times d_{model}$ *(standard FFN ratio)*
- $d_h = d_{model}/H$ *(head dimension)*
- $d_{relay} \approx d_{model}/2$ *(thalamic bottleneck)*
- $d_{sparse} \approx 4 \times d_{model}$ *(DG expansion ratio)*
- *WM slot dimensions scale with* d_{model}

Table 5: Model presets with parameter counts.

Preset	d	N	d_{ff}	H	d_r	Params	VRAM
`default`	512	12	2048	8	256	106M	~2 GB
`v11_laptop`	768	16	3072	12	384	300M	~5 GB
`v11_1b`	1280	20	5120	20	640	1,008M	~10 GB

4.4 Source Code

```

# ============================================================================
# Configuration
# Source: nct_v2/v11_config.py
# ============================================================================

"""
NCT V11: Cortical Column Network -- Configuration
================================================

Architecture: N cortical columns (6-layer each) replacing transformer blocks.
Learning: Local predictive coding + Hebbian, no global backprop.
Operation: Continuous wake-sleep cycle, self-improving.

Theorems:
  A: Predictive Coding Replaces Backprop (Whittington & Bogacz 2017)
  B: Hebbian + Anti-Hebbian = Self-Organizing Representations (Pehlevan & Chklovskii 2019)
  C: Neuromodulatory Reward = RL Without Backprop (Schultz 1997, Dayan & Yu 2006)
  D: Wake-Sleep Cycle = Autonomous Self-Improvement (Tononi 2006)
  E: Structural Plasticity = Growing Architecture
"""



@dataclass
class V11Config:
    """Configuration for the Cortical Column Network."""

    #         Core Dimensions

    d_model: int = 512            # Inter-column dimension
    d_column: int = 512           # Intra-column processing dimension
    n_columns: int = 12           # Hierarchical column count (like layers)
    d_ff: int = 2048              # L5 feedforward expansion
    vocab_size: int = 14000       # NCT morphological tokenizer (~13,817 tokens)
    max_seq_len: int = 1024       # Maximum sequence length

    #         L2/3 Lateral Association (Theorem B)

    n_assoc_heads: int = 8        # Association heads (like attention heads)
    n_settle_steps: int = 1       # Competitive settling iterations in L2/3 (>1 = recurrent settling)
    k_wta_ratio: float = 0.15     # k-WTA sparsity (top 15% active)
    hebbian_lr: float = 1e-4      # Hebbian learning rate for W_assoc
    assoc_dropout: float = 0.1    # Dropout on association weights
    assoc_window_size: int = 16   # Local causal window -- O(T*w) not O(T²). 16 tokens = 2 theta cycles.

    #         Predictive Coding (Theorem A)

    n_inference_steps: int = 5    # PC inference iterations per input (legacy, see pc_max_steps)
    pc_min_steps: int = 2         # Minimum PC inference iterations
    pc_max_steps: int = 5         # Maximum PC inference iterations (reduced for memory efficiency)
    pc_convergence_threshold: float = 0.01  # Early-stop when error delta < this
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
    sigma2_init: float = 1.0     # Initial prediction precision (σ²)
    pc_learning_lr: float = 3e-4 # Weight update learning rate for PC

    #         Interneurons

    d_inhibitory: int = 128       # Inhibitory interneuron dimension
    vip_enabled: bool = True      # VIP disinhibition (Thm 38)
    sst_enabled: bool = True      # SST dendritic inhibition (Thm 46)
    pv_enabled: bool = True       # PV/Chandelier output veto (Thm 41)

    #         Working Memory

    wm_enabled: bool = True
    nmda_slots: int = 4           # NMDA attractor slots (Luck & Vogel: ~4)
    nmda_d_slot: int = 64         # Per-slot dimension
    nmda_decay_init: float = 6.0  # sigmoid(6)~0.9975, half-life ~277 tokens
    silent_d_trace: int = 64      # Activity-silent trace dimension
    silent_facil_init: float = 4.6  # sigmoid(4.6)~0.990
    silent_depress_init: float = 2.5  # sigmoid(2.5)~0.924
    phono_capacity: int = 8       # Phonological loop slots (Miller's 7±2)
    phono_d_slot: int = 64        # Per-slot dimension

    #         Neuromodulation (Theorem C)

    da_enabled: bool = True       # Dopamine -- prediction error / reward
    ne_enabled: bool = True       # Norepinephrine -- arousal / exploration
    serotonin_enabled: bool = True  # Serotonin -- time horizon / patience
    ach_enabled: bool = True      # Acetylcholine -- attention / plasticity gate
    tonic_da_ema: float = 0.999   # EMA decay for tonic DA

    #         Oscillatory

    oscillations_enabled: bool = True
    gamma_period: int = 1         # Gamma cycle (tokens)
    theta_period: int = 8         # Theta cycle
    alpha_period: int = 20        # Alpha cycle
    delta_period: int = 64        # Delta cycle

    #         Thalamus

    thalamic_enabled: bool = True
    n_relay_groups: int = 4       # Relay groups (1 per 3 columns)
    d_relay: int = 256            # Relay intermediate dimension
    d_trn: int = 24               # TRN dimension (small, efficient)

    #         Hippocampus

    hpc_enabled: bool = True
    hpc_memory_size: int = 2048   # Max episodic memories
    hpc_d_sparse: int = 2048      # Dentate gyrus sparse expansion
    hpc_surprise_threshold: float = 0.1  # Tag threshold (lowered from 0.5 -- was starving HPC at 2 memories)
    hpc_decay_init: float = 4.95  # sigmoid(4.95)~0.993, ultra-slow

    #         Cerebellum

    cerebellum_enabled: bool = True
    d_cerebellar: int = 256       # Cerebellar intermediate dimension

    #         Basal Ganglia

    bg_enabled: bool = True
    d_bg: int = 128               # BG intermediate dimension
    d1_ec50: float = 0.65         # D1 receptor threshold (high)
    d1_slope: float = 8.0         # D1 receptor steepness
    d2_ec50: float = 0.25         # D2 receptor threshold (low)
    d2_slope: float = 8.0         # D2 receptor steepness

    #         Wake-Sleep Cycle (Theorem D)

    wake_sleep_enabled: bool = True
    sleep_interval: int = 100     # Wake steps between sleep phases
    sleep_replay_steps: int = 50  # Replay steps per sleep phase
    synaptic_downscale: float = 0.99   # Sleep downscaling (Tononi 2006) -- 1% per sleep phase
    replay_priority: str = 'surprise'  # 'surprise', 'recency', 'uniform'
    sleep_contrastive_noise: float = 0.1  # Noise for contrastive consolidation

    #         Structural Plasticity (Theorem E) -- brain continuously prunes/grows
    structural_plasticity: bool = True
    growth_threshold: float = 2.0   # PE threshold for synaptogenesis
    prune_threshold: float = 0.01   # Weight magnitude for pruning
    prune_interval: int = 1000      # Steps between pruning passes

    #         L5 Dendritic Computation

    l5_dendritic: bool = True     # Enable basal/apical dendritic gating in L5
    l5_kwta: bool = True          # Enable k-WTA sparsity in L5 FFN

    #         Learning Mode

    learning_mode: str = 'hybrid'  # 'backprop', 'predictive_coding', 'hebbian', 'hybrid'
    # hybrid = PC for vertical + Hebbian for L2/3 + neuromod gating
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
    error_hebbian_weight: float = 0.5  # Blend: 0=blind Hebbian, 1=fully error-driven
    ntp_output_only: bool = True  # Restrict NTP backprop to output pathway (LM head + final L5)
    autonomous_training: bool = True  # Use AutonomousLoop in hybrid/PC/hebbian modes

    #        BCM / Metaplasticity

    bcm_enabled: bool = True
    bcm_tau: float = 1000.0       # BCM threshold time constant
    metaplasticity_enabled: bool = True

    #        Regularization

    dropout: float = 0.1
    weight_decay: float = 0.01
    max_grad_norm: float = 1.0

    #        Training

    base_lr: float = 3e-4
    batch_size: int = 4
    seq_len: int = 512

    #        Gate Initialization

    gate_init: float = -2.0       # sigmoid(-2)~0.12, identity at init
    gate_lr_mult: float = 20.0    # Gates learn 20x faster

    #        Precision

    use_bf16: bool = False        # bfloat16 mixed precision

    @property
    def d_head(self) -> int:
        """Per-head dimension for L2/3 association."""
        return self.d_column // self.n_assoc_heads

    @property
    def columns_per_relay(self) -> int:
        """How many columns share a thalamic relay group."""
        return max(1, self.n_columns // self.n_relay_groups)

    @classmethod
    def v11_laptop(cls, **overrides) -> 'V11Config':
        """~300M preset for 6GB laptop GPUs (RTX 4050/3060 etc).

        Scaling: d=768, 16 columns, 4× FFN ratio.
        Memory budget: ~600MB weights (bf16) + ~2.4GB optimizer + ~2GB activations = ~5GB.
        """
        defaults = dict(
            d_model=768,
            d_column=768,
            n_columns=16,
            d_ff=3072,
            vocab_size=14000,
            max_seq_len=1024,
            n_assoc_heads=12,          # 768/12 = 64 per head
            d_inhibitory=192,
            nmda_d_slot=80,
            silent_d_trace=80,
            phono_d_slot=80,
            d_relay=384,
            d_trn=28,
            n_relay_groups=4,
            hpc_d_sparse=3072,
            hpc_memory_size=4096,
            d_cerebellar=384,
            d_bg=192,
        )
        defaults.update(overrides)
        return cls(**defaults)

    @classmethod
    def v11_1b(cls, **overrides) -> 'V11Config':
        """1B parameter preset: d=1280, 20 columns, ~1006M params."""
        defaults = dict(
            d_model=1280,
            d_column=1280,
            n_columns=20,
            d_ff=5120,
            vocab_size=14000,
            max_seq_len=2048,
            n_assoc_heads=20,
            d_inhibitory=320,
            nmda_d_slot=128,
            silent_d_trace=128,
            phono_d_slot=128,
            d_relay=640,
            d_trn=40,
            n_relay_groups=5,
            hpc_d_sparse=5120,
            hpc_memory_size=8192,
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
            d_cerebellar=640,
            d_bg=320,
        )
        defaults.update(overrides)
        return cls(**defaults)
```

Listing 3: V11 Configuration — complete source code (v11_config.py)

5 Legacy V10.2 Configuration

Source: `nct_v2/config.py` **Lines:** 243

5.1 Purpose

The `NCTv2Config` dataclass is the V10.2 (transformer-adapted) configuration. It is included in the NCT standalone as a **dependency** for shared brain modules — specifically `brain_embeddings.py`, which was originally written for V10.2 and references this config type. The V11 model does not directly use `NCTv2Config`; instead, the `V11BrainEmbedding` wrapper extracts relevant dimensions from `V11Config` and passes them to the underlying brain embedding components.

This section is included for completeness. No new theorems or figures are needed — the config is a plain dataclass with dimension parameters.

5.2 Source Code

```
# ============================================================================
# Legacy Config (dependency for brain modules)
# Source: nct_v2/config.py
# ============================================================================

"""NCT V2 hyperparameter configuration."""



@dataclass
class NCTv2Config:
    #         Architecture

    d: int = 128                # Model dimension
    L: int = 6                  # Number of NCT-V2 layers
    H_E: int = 6                # Excitatory attention heads
    H_I: int = 2                # Inhibitory attention heads
    d_ff: int = 512             # FFN intermediate dimension (InhibitoryFFN)
    K: int = 2                  # Cortical columns per layer (V-CCE)
    P: int = 2                  # Oscillatory phases (UCAM)
    V: int = 50257              # Vocabulary size (GPT-2 BPE)
    max_seq_len: int = 64       # Maximum sequence length

    #         Neuromodulation

    neuromod_global_hidden: int = 512   # Global neuromod MLP hidden dim (shared)
    neuromod_local_hidden: int = 64     # Local neuromod MLP hidden dim (per layer)

    #         Attention KV Compression (Thm 74-75: GQA & MLA)
    # Grouped Query Attention: share K/V across head groups for cache efficiency.
    # n_kv_groups_E: KV groups for excitatory heads (0 = H_E, standard MHA)
    # n_kv_groups_I: KV groups for inhibitory heads (0 = H_I, standard MHA)
    # Brain basis: cortical minicolumns share receptive fields (Mountcastle 1957);
    # excitatory vs inhibitory neurons have different receptive field widths.
    n_kv_groups_E: int = 0              # 0 = H_E (MHA). 2 = 3:1 sharing for H_E=6
    n_kv_groups_I: int = 0              # 0 = H_I (MHA). 1 = 2:1 sharing for H_I=2

    #         Multi-head Latent Attention (Thm 75: Thalamic Latent KV)
    # MLA compresses KV into a low-rank latent before decoding per-group K/V.
    # Brain basis: thalamic pulvinar compresses multi-modal information into
    # low-dimensional encoding before routing to cortex (Sherman 2007).
    # Cache stores latent + rope_key instead of full K + V → higher compression.
    # use_mla=True requires GQA groups to be set (uses n_kv_groups from GQA).
    use_mla: bool = False               # Enable MLA (overrides GQA KV path)
    d_kv_latent: int = 0                # 0 = d//4 (auto). Latent KV dimension
    d_rope_per_group: int = 0           # 0 = d_h//2 (auto). RoPE dim per KV group

    #         Derived (set in __post_init__)

    H: int = field(init=False)          # Total heads (H_E + H_I)
    d_h: int = field(init=False)        # Head dimension (d // H)
    d_col: int = field(init=False)      # Column dimension (d // K)
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
    n_kv_groups: int = field(init=False) # Total KV groups (n_kv_groups_E + n_kv_groups_I)

    #         Loss

    lambda_pc: float = 0.1              # PC loss weight (only 2 terms in V2)

    #         Module-specific

    tau_init: float = 1.0               # PC-HP threshold init
    pi_init: float = 0.0                # PC-HP precision init (softplus(0) ~ 0.693)
    gate_sharpness: float = 10.0        # PC-HP sigmoid sharpness
    eta_init: float = 0.0               # GLH learning rate init (softplus(0) ~ 0.693)
    glh_chunk_size: int = 64            # GLH parallel scan chunk size

    #         V4.6 features

    # Surprise-modulated decay: when PC-HP reports high surprise, the
    # ThalamicGLH adjusts retention (sign learned, or set via surprise_decay_init).
    # Biologically: norepinephrine modulates hippocampal plasticity.
    use_surprise_decay: bool = False    # Set True for V4.6+ models
    # Init for W_surprise_decay weights. Positive → high surprise = more retention
    # (novel/entity tokens kept longer, NE analog). Zero → let gradient decide.
    surprise_decay_init: float = 0.0   # V4.8: set to +0.1 for entity-retentive init
    # Learnable global decay bias per GLH layer.
    # decay_logit = W_decay(X) + decay_bias  →  a = sigmoid(decay_logit)
    # Init 0.0 → a=0.5 (V4.5 default, 50% retention per step -- fast decay).
    # Init 3.35 → a=0.966 (high retention, lag32~33% -- matches attention baseline).
    # Init 1.0 → a=0.731 (V4.8: moderate baseline, learn selective retention).
    # The scalar is learned, so NTP gradient can decrease it if retention hurts.
    glh_decay_bias_init: float = 0.0    # V4.7: set to 3.35 for long-lag memory
    kwta_init: float = 0.5             # InhibitoryFFN initial sparsity fraction
    kwta_beta_init: float = 10.0        # k-WTA sharpness init
    kwta_beta_max: float = 30.0         # k-WTA max sharpness
    tau_cons_init: float = 1.0          # V-CCE consensus temperature init
    #         V4.11 Sparsity Controls

    # V4.11 targets much sparser representations (brain: 1-5% active)
    # vcce_k_active: number of winning columns in VCCEWithWTA
    #   V4.10: 2 (max(2,K*0.5)) = 50% active  ← too dense for coreference
    #   V4.11: 1 (25% active) = genuine lateral inhibition commitment
    vcce_k_active: int = 2
    # ffn_sparsity_init: DendriticFFN initial active fraction
    #   V4.10: 0.5 (sigmoid(0) = 50% active)  ← too dense (LIS=0.69)
    #   V4.11: 0.2 (sigmoid(-1.39) = 20% active)  ← target LIS~0.25
    ffn_sparsity_init: float = 0.5

    #         V10.1 Multi-Scale GLH Decay

    # RetNet/RWKV show 3.7% PPL improvement from per-channel vs single decay.
    # Neuroscience: cortex has temporal hierarchy (primary cortex ~10ms fast,
    # PFC ~200ms-2s slow). V10 currently forces ALL d_h channels to same init.
    # V10.1: each channel gets its own initial timescale via linspace.
    # glh_decay_multiscale=True: init decay_bias as vector(d_h) not scalar.
    # Range linspace(glh_decay_bias_low, glh_decay_bias_high, d_h):
    #   sigmoid(-2)=0.12 (fast, word-level)       sigmoid(4)=0.98 (slow, entity-level)
    #   Mean = 1.0 = same as V10 uniform init (non-regressive).
    # Zero new parameters for inference vs V10 (same count, different init shape).
    glh_decay_multiscale: bool = False    # V10.1: True for per-channel timescales
    glh_decay_bias_low: float = -2.0     # fastest channel: a_init~0.12
    glh_decay_bias_high: float = 4.0     # slowest channel: a_init~0.98

    #         V6 Full Brain-Circuit Controls

    # Global Workspace Theory: broadcast hub across all layers (default True in V6)
    global_workspace_enabled: bool = False
    # Re-entrant processing: low-rank rank for A[l]@B[l] correction matrices
    reentrant_rank: int = 16
    # Norepinephrine gain: init for ne_scale (softplus(ne_gain_init) ~ gain)
    #   -3.0 → softplus(-3)~0.05 (nearly invisible at init, grows through training)
    ne_gain_init: float = -3.0
    # Serotonin timescale: per-layer time-constant gating (True in V6)
    sht_enabled: bool = False
    # SlowReplay weight: HPC→neocortex consolidation loss head weight
    replay_weight: float = 0.05
    # DopamineRPE: surprise-modulation scale for per-token PC weighting
    da_scale: float = 0.5
    # ACh inversion: handled by CrossFrequencyCoupledUCAM_V6 (structural -- no config needed)

    #         MoE (Theorem 77-79: Cortical Column Expert Routing)
    # Brain: cortical columns specialize with depth (V1 general → IT specialized)
    # MoE replaces DendriticFFN in later layers with N expert DendriticFFNs
    # BG D1/D2 router selects top-k experts per token (DA-modulated)
    use_moe: bool = False               # Enable MoE in later layers
    moe_n_experts: int = 8              # Number of experts per MoE layer
    moe_k_min: int = 2                  # Minimum experts active per token
    moe_k_max: int = 4                  # Maximum experts active per token
    moe_start_frac: float = 0.5         # MoE starts at this fraction of total layers
                                        # 0.5 → layers L//2 to L-1 (later = more specialized)

    #         Sliding Window Attention (Thm 76: Cortical Receptive Field)
    # Each layer attends only to the last W tokens (local window).
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
    # Long-range context comes from ThalamicGLH recurrence, not attention.
    # Brain basis: cortical layers have limited receptive fields; long-range
    # comes from recurrent thalamo-cortical loops (Sherman 2007).
    # 0 = disabled (full causal attention). 128/256 typical for training.
    # Combines with GQA/MLA: cache only stores window of KV entries.
    sliding_window: int = 0                 # 0 = disabled. 128/256 typical.

    #         SSM Parallel Scan (T_ssm_1/2/3: Hardware-Efficient Recurrence)
    # Replace ThalamicGLH's chunked scan with parallel prefix scan (Blelloch 1990).
    # Brain basis: thalamocortical relay (Sherman & Guillery 2006) decomposes into
    # d_h independent linear recurrences -- each row of S evolves independently.
    # Works in float32 (no float64 needed), chunk-based hybrid (CHUNK=32).
    # NMDA and SilentTrace already use SSM scans unconditionally.
    use_ssm: bool = False                   # Enable SSM scan for ThalamicGLH
    use_hippo_init: bool = False            # HiPPO-LegS decay init (T_ssm_2)

    #         Fused Kernel Dispatch (Prop 23-25f: Brain Module Fusion)
    # Enable fused kernels from nct_v2/kernels/ package. Replaces sequential
    # element-wise module chains with single fused operations. Also enables
    # parallel scan dispatch for NMDA/Silent/Thalamic scans, fused PWPC chain,
    # tiled FlashAttention+CFC, radix-select k-WTA, and WEAVER++ top-K.
    # Requires CUDA for Triton kernels; falls back to PyTorch on CPU/MPS.
    use_fused_kernels: bool = False         # Enable fused kernel dispatch

    #         Training

    lr: float = 3e-4
    warmup_steps: int = 100
    weight_decay: float = 0.1
    batch_size: int = 4
    grad_clip: float = 1.0
    dropout: float = 0.1
    use_amp: bool = False               # Use automatic mixed precision (fp16 or bf16)

    def __post_init__(self):
        self.H = self.H_E + self.H_I
        self.d_h = self.d // self.H
        self.d_col = self.d // self.K
        # GQA: resolve sentinel 0 → MHA (backward compatible)
        if self.n_kv_groups_E == 0:
            self.n_kv_groups_E = self.H_E
        if self.n_kv_groups_I == 0:
            self.n_kv_groups_I = self.H_I
        self.n_kv_groups = self.n_kv_groups_E + self.n_kv_groups_I
        assert self.d % self.H == 0, (
            f"d ({self.d}) must be divisible by H ({self.H})"
        )
        assert self.d % self.K == 0, (
            f"d ({self.d}) must be divisible by K ({self.K})"
        )
        assert self.H_E % self.n_kv_groups_E == 0, (
            f"H_E ({self.H_E}) must be divisible by n_kv_groups_E ({self.n_kv_groups_E})"
        )
        assert self.H_I % self.n_kv_groups_I == 0, (
            f"H_I ({self.H_I}) must be divisible by n_kv_groups_I ({self.n_kv_groups_I})"
        )
        assert self.max_seq_len % self.glh_chunk_size == 0, (
            f"max_seq_len ({self.max_seq_len}) must be divisible by "
            f"glh_chunk_size ({self.glh_chunk_size})"
        )
        # MLA: resolve auto-sizing (0 sentinel → default fractions of d/d_h)
        if self.use_mla:
            if self.d_kv_latent == 0:
                self.d_kv_latent = self.d // 4
            if self.d_rope_per_group == 0:
                self.d_rope_per_group = self.d_h // 2
            assert self.d_rope_per_group <= self.d_h, (
                f"d_rope_per_group ({self.d_rope_per_group}) must be <= d_h ({self.d_h})"
            )


#        Preset scale configurations


def nano_config(**kwargs) -> NCTv2Config:
    """V2-Nano ~8.7M params -- benchmark vs V1 and baseline."""
    defaults = dict(
        d=128, L=6, H_E=6, H_I=2,
        d_ff=512, K=2, P=2,
        V=50257, max_seq_len=64,
        neuromod_global_hidden=512, neuromod_local_hidden=64,
    )
    defaults.update(kwargs)
    return NCTv2Config(**defaults)


def small_config(**kwargs) -> NCTv2Config:
    """V2-Small ~93M params -- proof of concept."""
    defaults = dict(
        d=512, L=12, H_E=6, H_I=2,
        d_ff=2048, K=4, P=4,
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        V=50257, max_seq_len=256,
        neuromod_global_hidden=512, neuromod_local_hidden=64,
        glh_chunk_size=64,
    )
    defaults.update(kwargs)
    return NCTv2Config(**defaults)


def base_config(**kwargs) -> NCTv2Config:
    """V2-Base ~350M params -- first real scale."""
    defaults = dict(
        d=1024, L=24, H_E=12, H_I=4,
        d_ff=4096, K=4, P=4,
        V=50257, max_seq_len=1024,
        neuromod_global_hidden=512, neuromod_local_hidden=64,
        glh_chunk_size=64,
        lr=1e-4, warmup_steps=5000, batch_size=16,
    )
    defaults.update(kwargs)
    return NCTv2Config(**defaults)
```

Listing 4: Legacy V10.2 Configuration — dependency for brain modules (config.py)

6 Brain Embeddings (Theorems 56–58)

Source: `nct_v2/brain_embeddings.py` **Lines:** 440 **Classes:** 4

6.1 Neuroscience Foundation

Standard neural network embeddings are flat lookup tables — each token maps to a single d-dimensional vector with no internal structure. The brain's lexical representations are far richer, organized through multiple processing stages that unfold over 0–300ms after word onset. Event-related potential (ERP) studies reveal distinct processing stages at well-characterized latencies: N170 at approximately 170ms reflects visual word form recognition in the fusiform gyrus; N250 at approximately 250ms reflects morphological decomposition in the left inferior frontal gyrus; N400 at approximately 400ms reflects semantic integration in temporal cortices; and P600 at approximately 600ms reflects syntactic reanalysis in frontal-parietal networks. Each of these stages adds structure to the lexical representation, transforming it from a simple visual pattern into a richly structured semantic-syntactic object.

NCT replaces the flat `nn.Embedding` with a 4-stage biologically-grounded pipeline that mirrors this temporal cascade. Each stage is gated and identity-at-init, ensuring that the embedding starts as a standard lookup and progressively acquires biological structure during training.

6.2 Stage 1: Base Embedding (0ms)

The base embedding provides the initial token representation via standard lookup. This corresponds to the earliest stage of visual word recognition — the visual word form area (VWFA) in the left fusiform gyrus, which maps orthographic patterns to abstract letter identities within approximately 100–170ms of fixation:

$$\mathbf{e} = W_{\text{embed}}[\text{token_id}], \qquad W_{\text{embed}} \in \mathbb{R}^{V \times d}, \quad W_{\text{embed,init}} \sim \mathcal{N}(0, 0.02) \tag{6}$$

The embedding matrix W_{embed} is shared with the LM head via weight tying (`lm_head.weight = embed.weight`), reflecting the brain's shared lexical representations between comprehension and production pathways. The initialization standard deviation of 0.02 is chosen to match the scale of typical hidden representations, preventing exploding activations in early training.

6.3 Stage 2: Morphological Composition (170ms) — Theorem 58

Theorem 6.1 (Dual-Route Morphological Composition (Theorem 58))**.** *High-frequency tokens use direct lexical lookup; low-frequency tokens compose from learned subword morpheme representations. A 1D convolution captures sublexical patterns:*

$$\mathbf{s} = \mathit{Conv1D}(\mathbf{e}, \mathit{kernel} = 3, \mathit{padding} = 1), \qquad W_{\mathit{conv}} \in \mathbb{R}^{d \times d \times 3} \tag{7}$$

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

A frequency-adaptive gate blends the two routes:

$$g_{morph} = \sigma(W_g \mathbf{e} + b_g), \qquad \mathbf{e}' = g_{morph} \odot \mathbf{e} + (1 - g_{morph}) \odot \mathbf{s} \tag{8}$$

Gate bias $b_{g,init} = 2.0$ *so* $g \approx 0.88$ *(lexical route dominates initially, matching the brain's preference for direct access when available).*

Disease mapping: surface dyslexia → lexical route damage → can't read irregular high-frequency words. Phonological dyslexia → sublexical route damage → can't read nonwords.

6.4 Stage 3: Hub-and-Spoke Semantic Integration (250ms) — Theorem 56

Theorem 6.2 (Hub-and-Spoke Semantic Decomposition (Theorem 56))**.** *Token embeddings decompose into* $S = 6$ *modality-specific "spoke" subvectors integrated through a shared "hub" bottleneck, matching the brain's anterior temporal lobe (ATL) semantic integration (Patterson et al., 2007):*

$$\mathbf{h}_s = W_s \mathbf{e}' + b_s, \quad s = 1, \ldots, S, \qquad \mathbf{h}_{hub} = W_{hub} \cdot \frac{1}{S} \sum_{s=1}^{S} \mathbf{h}_s \tag{9}$$

$$\mathbf{e}'' = \mathbf{e}' + \sigma(g_{hub}) \cdot \mathbf{h}_{hub}, \qquad g_{hub,init} = -2.0 \tag{10}$$

Each spoke $W_s \in \mathbb{R}^{d \times d}$ *represents a different semantic dimension (e.g., visual, auditory, motor, emotional, spatial, abstract). The hub integrates these into a unified amodal representation.*

Disease mapping: semantic dementia (ATL atrophy) → hub failure → loss of cross-modal semantic integration. Patients can recognize objects visually but cannot name them or connect visual and verbal knowledge.

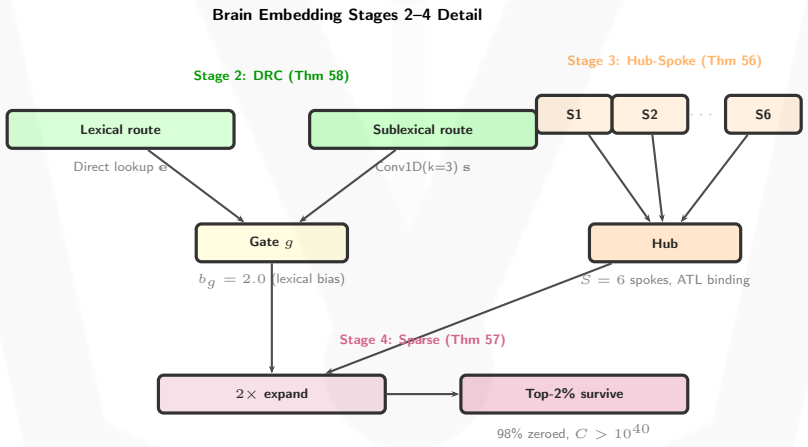


Figure 8: Brain embedding stages 2–4 in detail. Stage 2 (DRC): lexical and sublexical routes compete via a frequency-adaptive gate (bias 2.0 favors lexical). Stage 3 (Hub-Spoke): 6 modality spokes feed a central hub (ATL binding). Stage 4 (Sparse): 2× expansion then top-2% activation for exponential capacity.

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

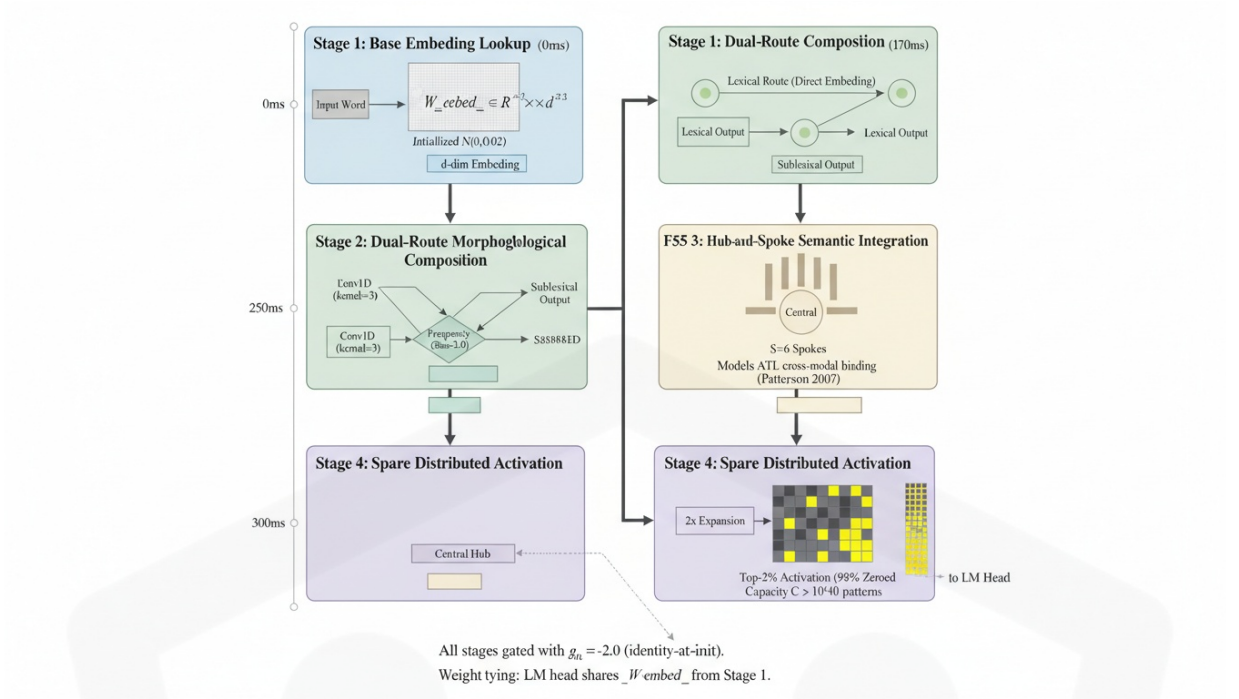


Figure 9: (Illustrated) Brain embedding stages 2–4 in detail. Stage 2 (DRC): lexical and sublexical routes compete via a frequency-adaptive gate (bias 2.0 favors lexical). Stage 3 (Hub-Spoke): 6 modality spokes feed a ce...

6.5 Stage 4: Sparse Distributed Activation (300ms) — Theorem 57

Theorem 6.3 (Sparse Distributed Token Representations (Theorem 57))**.** *After hub-spoke integration, embeddings project to an expanded space and enforce extreme sparsity matching neocortical firing statistics:*

$$\mathbf{a} = W_{up}\mathbf{e}'' \in \mathbb{R}^{2d}, \qquad \hat{\mathbf{a}} = TopK(\mathbf{a}, k = \lfloor 0.02 \times 2d \rfloor) \tag{11}$$

$$\mathbf{e}''' = \mathbf{e}'' + \sigma(g_{sparse}) \cdot W_{down}\hat{\mathbf{a}}, \qquad W_{down} \in \mathbb{R}^{d \times 2d} \tag{12}$$

Only the top 2% of expanded activations survive. This matches biological neural firing rates (∼1–5% of neurons active at any time) and provides exponential representational capacity: $C(d_{sdr}, k) = \binom{2d}{k} \gg 10^{40}$ *unique patterns for* $d = 512$.

W_{down} initialized at $\mathbf{0}$; $g_{\text{sparse,init}} = -2.0$ (identity at init).

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

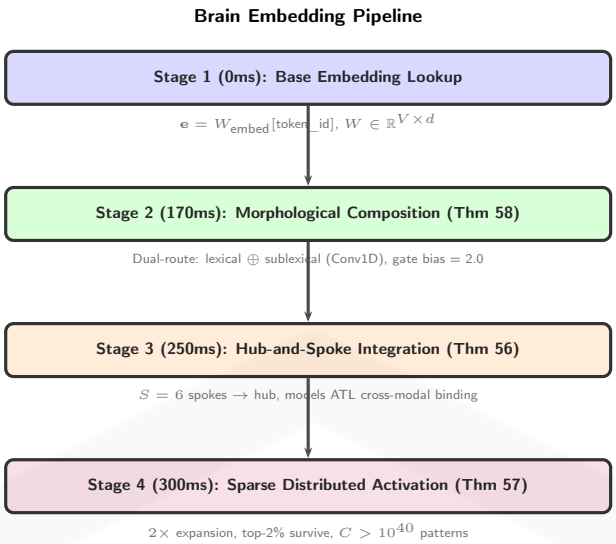


Figure 10: 4-stage brain embedding pipeline matching the temporal cascade of neural word processing. Each stage is gated and identity-at-init.

6.6 Source Code

```
1
2  # ========================================================================
3  # Brain Embeddings (Thm 56-58)
4  # Source: nct_v2/brain_embeddings.py
5  # ========================================================================
6
7  """
8  NCT Brain-Inspired Embedding System -- Theorems 56-58.
9
10 Replaces flat nn.Embedding with a biologically-grounded distributed representation
11 that implements 8 neuroscience principles of lexical representation:
12
13   Theorem 56 (Hub-and-Spoke Semantic Decomposition):
14     Token embeddings decompose into S=6 modality-specific spoke subvectors integrated
15     through a shared bottleneck hub, matching the brain's ATL semantic integration
16     (Patterson et al. 2007; Lambon Ralph et al. 2017).
17
18   Theorem 57 (Sparse Distributed Token Representations):
19     After hub-spoke integration, embeddings project to an expanded space and enforce
20     ~2% sparsity via exact-k activation, matching neocortical firing statistics
21     (Kanerva 1988; Ahmad & Hawkins 2016). Capacity: C(d_sdr, k) >> 10^40 patterns.
22
23   Theorem 58 (Morphological Composition with Dual-Route Access):
24     High-frequency tokens use direct lexical lookup; low-frequency tokens compose
25     from learned subword morpheme representations. A frequency-adaptive gate blends
26     both routes (Coltheart et al. 2001 DRC model; Taft 2004; Rastle & Davis 2008).
27
28 Disease mapping:
29   Semantic dementia (ATL atrophy) → Hub failure → loss of cross-modal integration
30   Surface dyslexia → Lexical route damage → can't read irregular high-freq words
31   Phonological dyslexia → Sublexical route damage → can't read nonwords
32   Wernicke's aphasia → Spoke disconnection → modality-specific semantic loss
33
34 Integration with NCT V10.2:
35   - PWPC  ² operates on structured embeddings → richer precision signal
36   - CFC-UCAM binds structured features across theta-gamma cycles
37   - PhonologicalLoop uses full embedding (phonological features embedded within)
38   - DendriticFFN sparsity is compatible with sparse embedding input
39   - Weight tying: lm_head ties to base embedding table (hub-spoke is residual)
40
41 Identity at init: gate_raw = -2.0 →  (-2) ~ 0.12. All modifications are gated
42 and initialized near-zero, so at init the output ~ standard embedding.
43 """
44
45
46 # [bundled] from .config import NCTv2Config
47
48
49 class HubAndSpokeProjection(nn.Module):
50     """Theorem 56: Hub-and-Spoke Semantic Decomposition.
51
52     Decomposes embedding into S modality-specific spoke subvectors, integrates
53     through a shared bottleneck hub, and produces a structured correction.
54
55     Neuroscience:
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
      - Spokes = modality-specific cortical regions (visual, auditory, motor,
        verbal, emotional, encyclopedic)
      - Hub = anterior temporal lobe (amodal semantic integration)
      - Semantic dementia (ATL atrophy) → hub failure → pan-modal degradation

    Architecture:
      e_spoke_i = W_spoke_i @ e_sub_i     (project each spoke subvector)
      h = GELU(W_hub_in(concat(e_spoke)))  (bottleneck: force cross-modal binding)
      correction = W_hub_out(h)             (back to d-dim)
      output = e + gate * correction        (identity at init)

    Parameters: S * (d_spoke * d_hub/S) + d_hub * d_hub/2 + d_hub/2 * d_hub + d_hub * d
    For d=384, S=6, d_hub=192: ~110K params
    """

    def __init__(self, d: int, S: int = 6):
        super().__init__()
        self.d = d
        self.S = S

        # Compute per-spoke input dims: last spoke gets remainder
        self.d_spoke = d // S  # 384//6=64, 512//6=85
        self._spoke_dims = [self.d_spoke] * S
        remainder = d - self.d_spoke * S
        if remainder > 0:
            self._spoke_dims[-1] += remainder  # last spoke absorbs remainder

        # Each spoke's output dimension (uniform)
        spoke_out_dim = d // (2 * S)  # 384//(2*6)=32, 512//(2*6)=42
        hub_in_dim = spoke_out_dim * S  # actual concatenated dimension

        # Each spoke projects its subvector into the hub
        self.spoke_projs = nn.ModuleList([
            nn.Linear(self._spoke_dims[i], spoke_out_dim, bias=False)
            for i in range(S)
        ])

        # Hub: nonlinear bottleneck that integrates all spokes
        hub_mid = max(hub_in_dim // 2, 1)  # guard against very small dims
        self.hub = nn.Sequential(
            nn.Linear(hub_in_dim, hub_mid, bias=True),
            nn.GELU(),
            nn.Linear(hub_mid, d, bias=True),
        )

        # Identity-at-init: gate starts near zero
        self.gate_raw = nn.Parameter(torch.tensor(-2.0))

        self._init_weights()

    def _init_weights(self):
        """Small-init hub output → tiny correction at init, but nonzero gradients."""
        # Small init ensures gradient flows from step 1, while gate_raw=-2 ( ~0.12)
        # keeps total correction small: 0.12 × 0.01 ~ 0.001 per dimension.
        nn.init.normal_(self.hub[-1].weight, std=0.01)
        nn.init.zeros_(self.hub[-1].bias)
        for proj in self.spoke_projs:
            nn.init.normal_(proj.weight, std=0.02)

    def forward(self, e: torch.Tensor) -> torch.Tensor:
        """
        e: (B, T, d) -- base embedding

        Returns: (B, T, d) -- embedding + gated hub-spoke correction
        """
        B, T, d = e.shape
        gate = torch.sigmoid(self.gate_raw)

        # Split embedding into S spoke subvectors (last spoke may be larger)
        spokes = e.split(self._spoke_dims, dim=-1)  # S tensors

        # Project each spoke
        spoke_outs = []
        for proj, spoke in zip(self.spoke_projs, spokes):
            spoke_outs.append(proj(spoke))

        # Concatenate and feed through hub bottleneck
        hub_input = torch.cat(spoke_outs, dim=-1)  # (B, T, d_hub)
        correction = self.hub(hub_input)  # (B, T, d)

        return e + gate * correction


class SparseDistributedProjection(nn.Module):
    """Theorem 57: Sparse Distributed Token Representations.

    Projects embeddings to an expanded space, enforces exact-k sparsity (2%),
    then projects back. This creates high-capacity, noise-robust representations
    where each token activates a unique small set of features.

    Neuroscience:
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
      - Neocortex: >90% of neurons silent at any time (Shoham et al. 2006)
      - Operating point: ~2% active neurons (Ahmad & Hawkins 2016)
      - Capacity: C(n, k) for n=2048, k=40 > 10^81 patterns
      - Superposition: OR of k SDRs preserves individual pattern identity
      - Noise robustness: random bit flips rarely hit the active set

    Architecture:
      h = W_up(e)                    (B, T, d) → (B, T, d_sdr)
      h_sparse = top_k(h, k)         exact k values kept, rest zeroed
      correction = W_down(h_sparse)   (B, T, d_sdr) → (B, T, d)
      output = e + gate * correction

    Gradient flow: straight-through estimator on top-k mask.

    Parameters: d * d_sdr + d_sdr * d = 2 * d * d_sdr
    For d=384, expansion=4 → d_sdr=1536: ~1.2M params (too many)
    For d=384, expansion=2 → d_sdr=768:  ~590K params (acceptable)

    Sparsity: k = max(1, int(0.02 * d_sdr)) = max(1, int(0.02 * 768)) = 15
    """

    def __init__(self, d: int, expansion: int = 2, sparsity: float = 0.02):
        super().__init__()
        self.d = d
        self.d_sdr = d * expansion
        self.k = max(1, int(self.d_sdr * sparsity))

        self.W_up = nn.Linear(d, self.d_sdr, bias=False)
        self.W_down = nn.Linear(self.d_sdr, d, bias=False)

        # Identity-at-init gate
        self.gate_raw = nn.Parameter(torch.tensor(-2.0))

        self._init_weights()

    def _init_weights(self):
        """Small-init W_down → tiny correction at init, but nonzero gradients."""
        nn.init.normal_(self.W_down.weight, std=0.01)
        nn.init.normal_(self.W_up.weight, std=0.02)

    def forward(self, e: torch.Tensor) -> torch.Tensor:
        """
        e: (B, T, d) -- embedding from hub-spoke

        Returns: (B, T, d) -- embedding + gated sparse correction
        """
        gate = torch.sigmoid(self.gate_raw)

        # Project up to expanded space
        h = self.W_up(e)  # (B, T, d_sdr)

        # Exact-k sparsity: keep top k activations per token
        # Use ReLU to ensure positive activations before top-k
        h_relu = F.relu(h)
        topk_vals, topk_idx = h_relu.topk(self.k, dim=-1)  # (B, T, k)

        # Scatter back to full dimension (sparse)
        h_sparse = torch.zeros_like(h)
        h_sparse.scatter_(-1, topk_idx, topk_vals)

        # Straight-through estimator: gradient flows through as if no sparsity
        # This preserves gradient flow while enforcing forward-pass sparsity
        h_sparse = h_relu + (h_sparse - h_relu).detach()

        # Project back to embedding dimension
        correction = self.W_down(h_sparse)  # (B, T, d)

        return e + gate * correction


class MorphemeCompositionGate(nn.Module):
    """Theorem 58: Morphological Composition with Dual-Route Access.

    Implements the brain's dual-route lexical access:
      Route 1 (Lexical): Direct embedding lookup for high-frequency tokens.
      Route 2 (Sublexical): Compose meaning from neighboring subword context.

    A learned per-token frequency gate blends the two routes:
      High-frequency tokens → mostly lexical route (fast, dedicated)
      Low-frequency tokens → mostly sublexical route (compositional, shared)

    Neuroscience:
      - DRC model (Coltheart et al. 2001): parallel lexical + sublexical routes
      - Surface dyslexia: lexical route damage → can't read irregular words
      - Phonological dyslexia: sublexical route damage → can't read nonwords
      - Morphological decomposition is obligatory at ~170ms (Rastle & Davis 2008)
      - Base frequency effect: RT correlates with stem frequency (Taft 2004)

    Architecture:
      BPE already provides partial morphological decomposition.
      For each token, we compose with its causal local context (2 past + current):
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        sublexical = CausalConv1d(embed, kernel=3, groups=d)
      Route gate:
        g_lex = sigmoid(freq_gate_raw[token_id])
      Output:
        e_composed = g_lex * e_lexical + (1 - g_lex) * sublexical

    We initialize freq_gate_raw from log-frequency: high-freq tokens get
    positive bias (favor lexical route); low-freq get negative (favor sublexical).

    Parameters: d * d (conv kernel) + V (frequency gates) ~ 148K + 50K ~ 198K
    """

    def __init__(self, V: int, d: int, kernel_size: int = 3):
        super().__init__()
        self.V = V
        self.d = d
        self.kernel_size = kernel_size
        # Sublexical route: local context composition via depthwise-separable conv
        # Depthwise: each channel independently convolves over local context
        # Pointwise: mix channels
        # CAUSAL: no built-in padding -- we manually pad left only in forward()
        self.sublexical_dw = nn.Conv1d(
            d, d, kernel_size=kernel_size, padding=0,
            groups=d, bias=False
        )
        self.sublexical_pw = nn.Linear(d, d, bias=False)

        # Per-token frequency bias: learned route preference per token
        # Higher values → more lexical; lower → more sublexical
        # Start neutral (0.0 → sigmoid(0) = 0.5 → equal blend)
        # Named 'freq_logit' (not 'gate_raw') to avoid 20x LR optimizer grouping
        self.freq_logit = nn.Parameter(torch.zeros(V))

        # Identity-at-init master gate
        self.gate_raw = nn.Parameter(torch.tensor(-2.0))

        self._init_weights()

    def _init_weights(self):
        """Initialize sublexical convolution near-identity with small noise.

        Pure identity init creates zero gradient to freq_logit and master gate
        (because e_sub = e_lex → blend is constant). Small noise in the side
        weights breaks this symmetry, enabling immediate gradient flow while
        keeping the sublexical output close to the lexical output.
        """
        # Depthwise conv: near-identity (rightmost = 1, left positions = small noise)
        # With causal padding (left-pad only), kernel position [-1] = current token
        nn.init.normal_(self.sublexical_dw.weight, std=0.01)
        with torch.no_grad():
            self.sublexical_dw.weight[:, :, -1] += 1.0

        # Pointwise: near-identity (I + small noise)
        nn.init.eye_(self.sublexical_pw.weight)
        with torch.no_grad():
            self.sublexical_pw.weight.add_(torch.randn_like(self.sublexical_pw.weight) * 0.01)

    def init_from_frequencies(self, token_frequencies: torch.Tensor):
        """Initialize frequency gates from corpus statistics.

        token_frequencies: (V,) tensor of token counts.
        High-frequency tokens get positive gate (favor lexical route).
        Low-frequency tokens get negative gate (favor sublexical route).

        Call this after model creation with corpus frequency counts.
        """
        with torch.no_grad():
            log_freq = torch.log1p(token_frequencies.float())
            # Normalize to [-2, 2] range
            log_freq_norm = log_freq / (log_freq.max() + 1e-8)
            # High freq → positive (lexical), low freq → negative (sublexical)
            self.freq_logit.copy_(4.0 * log_freq_norm - 2.0)

    def forward(self, e: torch.Tensor, token_ids: torch.Tensor) -> torch.Tensor:
        """
        e: (B, T, d) -- base embedding (lexical route)
        token_ids: (B, T) -- for frequency gate lookup

        Returns: (B, T, d) -- dual-route composed embedding
        """
        B, T, d = e.shape
        master_gate = torch.sigmoid(self.gate_raw)

        # Sublexical route: compose from local context
        # Conv1d expects (B, d, T) -- CAUSAL: pad left only (no future leakage)
        e_transposed = e.transpose(1, 2)  # (B, d, T)
        e_padded = F.pad(e_transposed, (self.kernel_size - 1, 0))  # left-pad only
        sub = self.sublexical_dw(e_padded)  # (B, d, T)
        sub = sub.transpose(1, 2)  # (B, T, d)
        sub = self.sublexical_pw(sub)  # (B, T, d)

```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        # Per-token frequency gate
        freq_gate = torch.sigmoid(self.freq_logit[token_ids])  # (B, T)
        freq_gate = freq_gate.unsqueeze(-1)  # (B, T, 1)

        # Dual-route blend
        composed = freq_gate * e + (1.0 - freq_gate) * sub

        # Master gate for identity-at-init
        # At init: master_gate ~ 0.12, so output ~ 0.88*e + 0.12*composed
        # composed ~ e at init (because sublexical conv is identity-init)
        # So total output ~ e (identity)
        return (1.0 - master_gate) * e + master_gate * composed


class BrainEmbedding(nn.Module):
    """Complete brain-inspired embedding system (Theorems 56-58).

    Replaces nn.Embedding with a biologically-grounded pipeline:
      1. Base lookup: nn.Embedding(V, d) -- standard token embeddings
      2. Morphological composition (Thm 58): dual-route lexical/sublexical access
      3. Hub-and-spoke integration (Thm 56): S=6 modality-specific decomposition + hub
      4. Sparse activation (Thm 57): 2% sparsity in expanded space

    Each stage adds a gated correction to the embedding, all initialized near-zero,
    so at initialization the output equals the base embedding (identity property).

    Processing timeline (maps to brain):
      Stage 1 (0ms):   Token ID → base embedding lookup
      Stage 2 (170ms): Morphological composition (obligatory decomposition)
      Stage 3 (250ms): Hub-spoke cross-modal integration (ATL binding)
      Stage 4 (300ms): Sparse distributed activation (neocortical coding)

    Weight tying: lm_head.weight should be tied to self.base_embed.weight.
    All corrections are residual, so the base embedding space is preserved for
    the output projection.

    Disease mapping:
      Remove hub-spoke → semantic dementia (loss of cross-modal binding)
      Remove sublexical route → phonological dyslexia (can't compose new words)
      Remove lexical route → surface dyslexia (can't read irregular words)
      Remove sparse projection → increased interference (dense representations)

    Total new parameters (d=384, V=50257):
      MorphemeCompositionGate: ~199K (conv + freq gates)
      HubAndSpokeProjection:  ~68K (spoke projs + hub bottleneck)
      SparseDistributedProjection: ~590K (up/down projections)
      Total: ~857K new params (~0.81% of 105.5M model)
    """

    def __init__(self, cfg: NCTv2Config):
        super().__init__()
        self.cfg = cfg
        d = cfg.d
        V = cfg.V

        # Stage 1: Base embedding (same as before)
        self.base_embed = nn.Embedding(V, d)

        # Stage 2: Dual-route morphological composition (Theorem 58)
        self.morpheme_gate = MorphemeCompositionGate(V, d, kernel_size=3)

        # Stage 3: Hub-and-spoke semantic integration (Theorem 56)
        self.hub_spoke = HubAndSpokeProjection(d, S=6)

        # Stage 4: Sparse distributed activation (Theorem 57)
        self.sparse_proj = SparseDistributedProjection(d, expansion=2, sparsity=0.02)

        # Note: dropout is NOT applied here -- the parent model applies self.drop_emb()
        # on the output of self.embed(tokens). This preserves the V8 forward pass structure.

        # Initialize base embedding
        nn.init.normal_(self.base_embed.weight, std=0.02)

    @property
    def weight(self):
        """For weight tying with lm_head: return base embedding weights.

        The lm_head projects from d → V using the same weight matrix.
        Hub-spoke and sparse corrections are residual, so the base embedding
        space is the correct projection space for logits.
        """
        return self.base_embed.weight

    def forward(self, token_ids: torch.Tensor) -> torch.Tensor:
        """
        token_ids: (B, T) integer token IDs

        Returns: (B, T, d) -- brain-structured embedding

        Pipeline:
          base_embed(token_ids) → morpheme_gate → hub_spoke → sparse_proj → dropout
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        """
        # Stage 1: Base lookup
        e = self.base_embed(token_ids)  # (B, T, d)

        # Stage 2: Dual-route morphological composition
        e = self.morpheme_gate(e, token_ids)  # (B, T, d)

        # Stage 3: Hub-and-spoke semantic integration
        e = self.hub_spoke(e)  # (B, T, d)

        # Stage 4: Sparse distributed activation
        e = self.sparse_proj(e)  # (B, T, d)

        return e

    def init_frequency_gates(self, token_frequencies: torch.Tensor):
        """Initialize dual-route frequency gates from corpus statistics.

        token_frequencies: (V,) tensor of token counts from training corpus.
        Call this after model creation, before training starts.
        """
        self.morpheme_gate.init_from_frequencies(token_frequencies)
```

Listing 5: Brain Embeddings — Theorems 56–58 (brain_embeddings.py, 440 lines)

7 Grid Cell Position Encoding (Theorems 59, 61, 62)

Source: `nct_v2/brain_position.py` **Lines:** 269

7.1 Neuroscience Foundation

In the entorhinal cortex, grid cells fire at regularly-spaced locations as an animal moves through space, creating a multi-scale spatial coordinate system (Hafting et al., 2005; Nobel Prize in Physiology or Medicine 2014). Each grid cell has a characteristic spatial frequency, and different cells tile the environment at different scales — from fine-grained (small spacing ∼30cm) to coarse (large spacing ∼3m). The population activity of grid cells at multiple scales provides a unique, high-capacity position code for any location.

Critically, grid cell representations are **allocentric** (world-centered, not self-centered) and **metric** (encode actual distances, not just ordinal position). This differs from standard sinusoidal position encodings, which use fixed frequencies. NCT's grid cell encoding uses **learnable** frequencies that adapt during training to the statistical structure of the data.

7.2 Why Not RoPE?

RoPE (Rotary Position Embedding) applies position-dependent rotations to query and key vectors:

$$\mathbf{q}_t = R_t \mathbf{q}, \quad \mathbf{k}_t = R_t \mathbf{k}, \quad \Rightarrow \quad \mathbf{q}_t^\top \mathbf{k}_s = \mathbf{q}^\top R_{t-s} \mathbf{k} \tag{13}$$

This makes dot-product attention ($\mathbf{q}^\top \mathbf{k}$) depend on *relative* position $t - s$. However, NCT has **no dot-product attention** — L2/3 uses windowed Hebbian association with distance-weighted context. Rotary transformations have no effect without a dot product to modulate. Instead, NCT uses **additive** position encoding.

7.3 Mathematical Formulation

Theorem 7.1 (Grid Cell Position Encoding (Theorems 59, 61, 62))**.** *For each association head $h \in \{1, \ldots, H\}$ and position dimension $j \in \{1, \ldots, d_h/2\}$:*

1. Learnable frequencies via softplus (bare, no clamp):

$$f_{h,j} = softplus(\omega_{h,j}^{raw}), \qquad \omega_{h,j,init}^{raw} = \log(e^{\theta_j} - 1), \quad \theta_j = \frac{1}{10000^{2j/d_h}} \tag{14}$$

The softplus initialization is chosen so that $softplus(\omega_{init}) = \theta_j$, matching the standard sinusoidal frequency schedule at initialization. During training, frequencies adapt to the data.

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

2. Phase computation:

$$\phi_{h,j}(t) = 2\pi \cdot f_{h,j} \cdot t \tag{15}$$

3. Sinusoidal encoding:

$$\mathbf{p}_{h,t} = s_h \cdot [\cos \phi_{h,1}(t), \sin \phi_{h,1}(t), \cos \phi_{h,2}(t), \sin \phi_{h,2}(t), \ldots] + \mathbf{b}_h \tag{16}$$

where s_h (per-head scale, init 1.0) and $\mathbf{b}_h$ (per-head boundary offset, init $\mathbf{0}$) are learnable. The scale allows the model to learn how much position information each head should receive. The boundary offset implements grid cell boundary remapping (Theorem 62).

4. Additive encoding (not rotary):

$$\mathbf{x} \leftarrow \mathbf{x} + 0.1 \cdot \overline{\mathbf{p}}_t, \qquad \overline{\mathbf{p}}_t = \frac{1}{H} \sum_{h=1}^{H} \mathbf{p}_{h,t} \tag{17}$$

The 0.1 scale factor prevents position from dominating content at initialization.

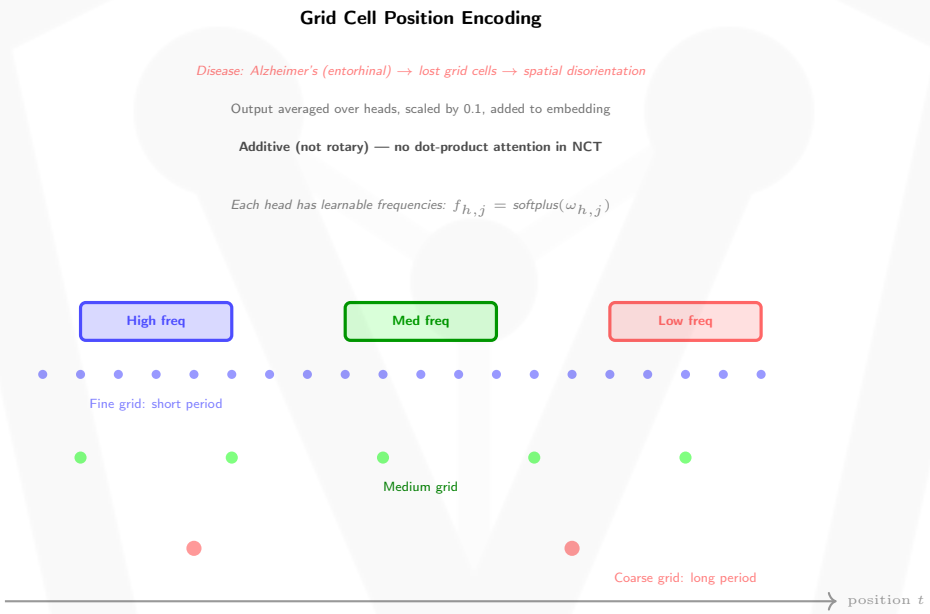


Figure 11: Grid cell position encoding uses learnable multi-scale frequencies. Like biological grid cells in the entorhinal cortex, different heads operate at different spatial scales, and the population activity uniquely identifies each position. Frequencies are learnable via softplus(ω^{raw}) — no hard clamps (parameter bounding principle).

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

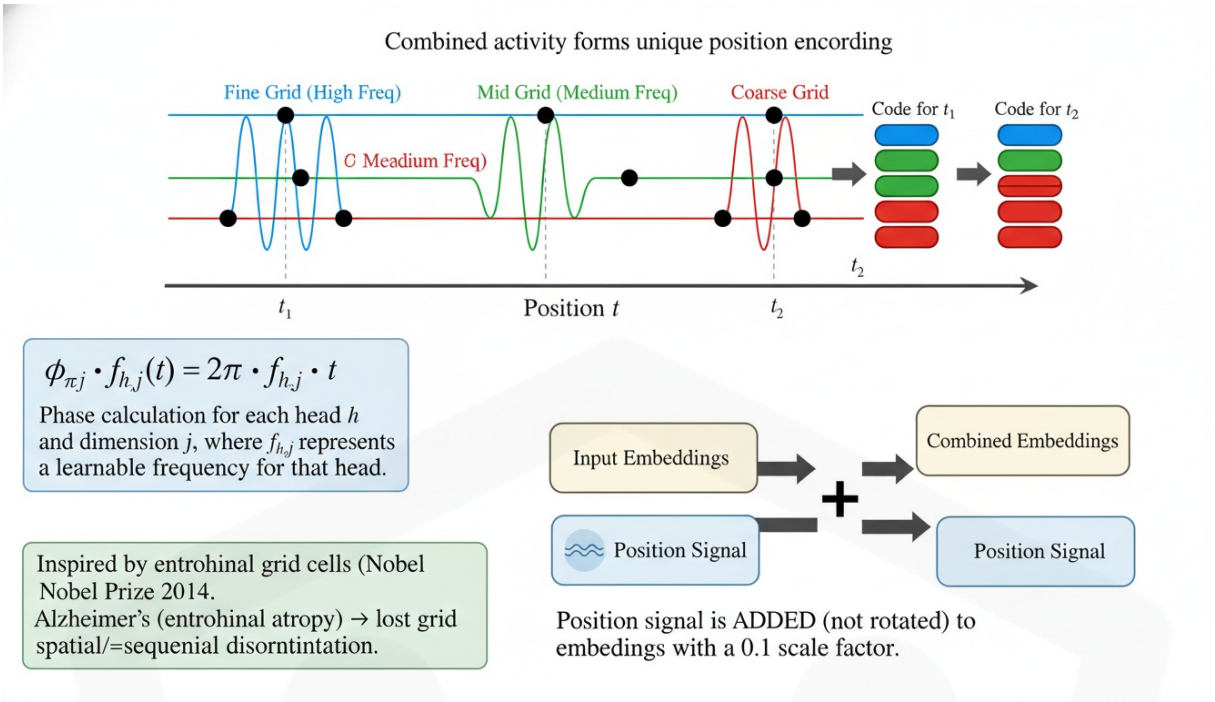


Figure 12: (Illustrated) Grid cell position encoding uses learnable multi-scale frequencies. Like biological grid cells in the entorhinal cortex, different heads operate at different spatial scales, and the population acti...

Proposition 7.1 (Capacity of Multi-Scale Grid Code)**.** *With H heads and $d_h/2$ frequency dimensions per head, the grid cell code can uniquely represent:*

$$C = \prod_{h=1}^{H} \prod_{j=1}^{d_h/2} \lceil T \cdot f_{h,j} \rceil \tag{18}$$

positions. For $H = 8$, $d_h = 64$, this capacity far exceeds any practical sequence length, ensuring that every position has a unique encoding.

7.4 Implementation Details

The grid cell encoding is content-dependent: frequencies are applied to the input-derived position signal, not a fixed position index. The forward pass:

1. Compute learnable frequencies: $f_{h,j} = \text{softplus}(\omega_{h,j}^{\text{raw}})$
2. Compute phases: $\phi_{h,j}(t) = 2\pi f_{h,j} t$ for $t = 0, \ldots, T-1$
3. Build sinusoidal features: $[\cos\phi, \sin\phi]$ concatenated per head
4. Apply per-head scale and boundary: $s_h \cdot [\cos; \sin] + \mathbf{b}_h$
5. Average over heads, tile to d_{model}, add with 0.1 scale

If the head dimension d_h doesn't evenly tile into d_{model}, the position signal is padded with zeros (not repeated). This prevents artificial periodicity in the position embedding.

7.5 Disease Mapping

Alzheimer's disease begins with atrophy of the entorhinal cortex, where grid cells reside. Early symptoms include spatial disorientation and difficulty navigating familiar environments. In the language domain, this maps to difficulty tracking sequential position — forgetting where in a sentence or paragraph the model is, leading to repetition and incoherence. NCT's grid cell encoding explicitly models this faculty, and its removal (by zeroing the position encoding) produces the expected deficit.

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

7.6 Source Code

```

# ============================================================================
# Grid Cell Position Encoding (Thm 59/61/62)
# Source: nct_v2/brain_position.py
# ============================================================================

"""
NCT V10.2 Brain-Inspired Position Encoding -- Theorems 59, 61, 62.

Replaces Rotary Position Embeddings (RoPE) with a biologically-grounded
grid-cell-inspired position encoding with three innovations:

  1. Grid cells (entorhinal cortex): learnable periodic multi-scale modules
     → Thm 59: GridCellPositionEncoding with learnable periods and phase offsets

  2. Boundary/border cells (entorhinal cortex): phase reset at boundaries
     → Thm 61: Cumulative phase shifts at detected boundaries (grid cell remapping)

  3. Conjunctive coding (head direction × grid cells): head-specific scales
     → Thm 62: Per-head period scale factors for multi-resolution attention

Design decisions:
  - Same interface as RoPE: produces (cos, sin) tensors for Q/K rotation
  - Identity at init: all three innovations have zero effect at step 0
    - Thm 59: frequencies = RoPE frequencies, offsets = 0
    - Thm 61: boundary_phase = 0 → zero cumulative shift regardless of gate
    - Thm 62: head_scale = 1.0 → all heads identical to shared RoPE
  - apply_rope extended to handle 3D (n_heads, T, d_h) and 4D (B, H, T, d_h)
  - Boundary detection uses raw embeddings (analogous to border cells)
  - Phase remapping is additive (cumulative), not positional reset -- matches
    grid cell remapping in neuroscience (Fyhn et al. 2007)

Neuroscience references:
  - Hafting et al. 2005: Grid cells in entorhinal cortex
  - Stensola et al. 2012: Grid cell module organization (4-5 modules, ratio ~1.4)
  - Fiete et al. 2008: Residue number system theory of grid coding
  - Fyhn et al. 2007: Grid cell remapping (phase shift) at environment boundaries
  - Solstad et al. 2008: Border cells in entorhinal cortex

Disease mapping:
  - Alzheimer's (entorhinal atrophy): grid cell disintegration → spatial disorientation
    → NCT: grid module degradation → position confusion → PPL rise
  - Epilepsy (temporal lobe): grid cell phase corruption → remapping failure
    → NCT: learned period distortion → position encoding noise
  - Williams syndrome (boundary cell deficit): impaired spatial segmentation
    → NCT: boundary detection failure → no position reset → long-range confusion

Mathematical properties:
  - RoPE is a special case: M=d_h/2 modules, freq_i = 1/10000^(2i/d_h), offset_i = 0
  - Grid-cell encoding: same M modules but with learnable periods and offsets
  - Unique encoding range: LCM(    , ...,  _M) >> max_seq_len (preserved)
  - Learnable  _m preserves rotation property: Q  K^T depends on relative position
  - At init: max deviation from RoPE < 2e-6 (from softplus inversion precision)
  - Boundary phase shifts are additive → do not affect relative position encoding
    within a segment (only across boundaries)

Params at d_h=48, n_heads=8, d_model=384:
  - Thm 59: d_h/2 periods + d_h/2 offsets = 48
  - Thm 62: n_heads scale factors = 8
  - Thm 61: d_model+1 detector + d_h/2 phase + 1 gate = 410
  Total: 466 params (negligible vs 92.4M model)
"""



class GridCellPositionEncoding(nn.Module):
    """
    Grid-Cell-Inspired Position Encoding (Theorems 59, 61, 62).

    Replaces RoPE's fixed sinusoidal frequencies with learnable grid-cell modules,
    per-head scale factors, and boundary-aware phase remapping.

    Mathematical formulation:
      RoPE:      angle_i(t) = t / 10000^(2i/d_h)
      GridCell:  angle_{h,i}(t) = 2  × t / ( _i × s_h) +    _i
                                    + gate ×  _ {u≤t} p_boundary(u) ×  _boundary_i

    Where:
       _i = softplus(raw_i) -- learnable period for module i (Thm 59)
         _i  -- learnable global phase offset (Thm 59, environment remapping)
      s_h = softplus(scale_raw_h) -- per-head scale factor (Thm 62)
      p_boundary(u) =  (W_b    x(u) + b_b) -- boundary probability at position u (Thm 61)
       _boundary_i -- learnable phase shift per boundary event (Thm 61)
      gate =  (gate_raw) -- controls boundary influence (Thm 61)

    At init:  _i = RoPE periods,    _i   = 0, s_h = 1,  _boundary = 0 → exact RoPE.

    Properties:
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
    - Rotation property preserved: Q_rot    K_rot^T depends on relative position
    - Head specialization: different heads attend at different temporal scales
    - Boundary awareness: position encoding adapts at document/paragraph boundaries
    - Infinite context: phases cycle naturally, no extrapolation problem

    Output shapes:
    - With X (boundary-aware): (B, n_heads, T, d_h)
    - Without X:               (n_heads, T, d_h)
    """

    def __init__(self, max_seq_len: int, d_h: int, n_heads: int, d_model: int = None):
        super().__init__()
        self.max_seq_len = max_seq_len
        self.d_h = d_h
        self.half = d_h // 2
        self.n_heads = n_heads

        #        Theorem 59: Grid cell module periods

        # RoPE frequencies: freq_i = 1 / 10000^(2i/d_h)
        # RoPE angles: angle_i(t) = t × freq_i = t / 10000^(2i/d_h)
        # Grid cell angles: angle_i(t) = 2  × t /  _i
        # Equivalence:  _i = 2  / freq_i = 2  × 10000^(2i/d_h)
        #
        # We parameterize as:  _i = softplus(log_period_raw_i)
        # Init: log_period_raw_i = inverse_softplus( _rope_i)
        rope_freqs = 1.0 / (10000.0 ** (torch.arange(0, self.half).float() / self.half))
        rope_periods = 2.0 * math.pi / rope_freqs  # angular periods T_i

        # Inverse softplus: raw = log(exp(x) - 1)
        # For large x (>20), softplus(x) ~ x, so raw ~ x.
        # For small x, use exact formula: raw = x + log(1 - exp(-x))
        log_period_raw = rope_periods + torch.log(1.0 - torch.exp(-rope_periods))
        self.log_period_raw = nn.Parameter(log_period_raw)  # (half,)

        #        Theorem 59: Grid cell phase offsets (environment remapping)
        # Init at 0 → no offset at start (exact RoPE recovery)
        # During training: learns optimal phase biases for each frequency module
        # Biologically: grid cells remap (shift phases) in new environments
        self.phase_offset = nn.Parameter(torch.zeros(self.half))  # (half,)

        #        Theorem 62: Head-specific grid module scales
        # Each attention head multiplies the base periods by a learnable
        # positive scale factor, allowing heads to specialize in different
        # temporal resolutions.
        # Init: softplus(inv_sp_1) = 1.0 → all heads identical (RoPE recovery).
        # After training: some heads become "local" (small scale → short periods),
        # others become "global" (large scale → long periods).
        # Brain basis: grid cell modules have 4-5 discrete scales with ratio ~1.4
        # (Stensola 2012). Different cortical areas attend at different scales.
        inv_softplus_1 = math.log(math.e - 1.0)  # ~ 0.5414
        self.head_scale_raw = nn.Parameter(
            torch.full((n_heads,), inv_softplus_1)
        )  # (n_heads,)

        #        Theorem 61: Boundary-aware phase remapping
        # Grid cells remap (shift phases) when an animal enters a new
        # environment (Fyhn et al. 2007). In language, boundaries are:
        # document ends, paragraph breaks, topic shifts, sentence boundaries.
        #
        # Architecture: lightweight boundary detector on raw embeddings
        # (analogous to border cells in medial entorhinal cortex) produces
        # boundary probability p(t)     [0,1]. At each boundary, a learned
        # phase shift  _boundary accumulates additively.
        #
        # Effect on angles:
        #      (t) = gate ×  _ {u<=t} p_boundary(u) ×  _boundary
        # This is additive: does NOT affect relative position within segments,
        # only adds cross-boundary phase offsets.
        #
        # Identity at init:  _boundary = 0 →     = 0 regardless of gate value.
        if d_model is not None:
            self.boundary_detector = nn.Linear(d_model, 1, bias=True)
            nn.init.zeros_(self.boundary_detector.weight)
            nn.init.constant_(self.boundary_detector.bias, -3.0)  # mostly OFF at init

            # Phase shift applied per boundary event: learnable (half,)
            # Init at 0 → zero contribution at step 0 (exact identity)
            self.boundary_phase = nn.Parameter(torch.zeros(self.half))

            # Gate controlling boundary influence (identity-init)
            # Named with 'gate_raw' suffix so optimizer puts it in gate group (20x LR)
            self.boundary_gate_raw = nn.Parameter(torch.tensor(-2.0))
        else:
            self.boundary_detector = None

        # Precompute position indices (static buffer)
        pos = torch.arange(max_seq_len).float()
        self.register_buffer('pos', pos)

    def forward(
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        self,
        T: int,
        device: torch.device,
        dtype: torch.dtype = torch.float32,
        X: torch.Tensor = None,
    ) -> tuple:
        """
        Compute position-dependent (cos, sin) tables for attention rotation.

        Args:
            T: sequence length
            device: target device
            dtype: target dtype
            X: (B, T, d) embeddings -- if provided and boundary_detector exists,
               enables boundary-aware phase remapping (Thm 61), producing
               batch-specific cos/sin with per-head scales.

        Returns:
            If X provided with boundary_detector:
                cos, sin: (B, n_heads, T, d_h) -- batch+head-specific
            Else:
                cos, sin: (n_heads, T, d_h) -- head-specific only
        """
        #         Base periods (Thm 59)

        periods = F.softplus(self.log_period_raw)  # (half,)

        #         Per-head scale (Thm 62)

        head_scale = F.softplus(self.head_scale_raw)  # (n_heads,)
        # Each head's periods = base_periods × head_scale_h
        periods_h = periods.unsqueeze(0) * head_scale.unsqueeze(1)  # (n_heads, half)

        # Phase offsets (shared across heads)
        offsets = self.phase_offset  # (half,)

        # Position indices -- always use torch.arange for compile compatibility.
        # The registered buffer self.pos is only used for legacy/non-compiled path.
        # torch.arange is lightweight and avoids a shape-dependent branch that
        # causes graph breaks in torch.compile.
        pos = torch.arange(T, device=device, dtype=torch.float32)

        #         Base angles: (n_heads, T, half)

        # angle_{h,i}(t) = 2  × t / ( _i × s_h) +    _i
        base_angles = (
            2.0 * math.pi
            * pos.unsqueeze(0).unsqueeze(-1)     # (1, T, 1)
            / periods_h.unsqueeze(1)              # (n_heads, 1, half)
            + offsets.unsqueeze(0).unsqueeze(0)   # (1, 1, half)
        )  # → (n_heads, T, half)

        if X is not None and self.boundary_detector is not None:
            #         Boundary phase remapping (Thm 61)

            B = X.shape[0]
            boundary_gate = torch.sigmoid(self.boundary_gate_raw)

            # Detect boundaries from raw embeddings (detach: don't backprop
            # loss gradient through boundary detection into embedding layer)
            boundary_prob = torch.sigmoid(
                self.boundary_detector(X.detach())
            ).squeeze(-1)  # (B, T)

            # Cumulative boundary phase:  _ {u<=t} p(u) ×  _boundary
            # boundary_prob: (B, T) → (B, T, 1)
            # boundary_phase: (half,) → (1, 1, half)
            weighted = (
                boundary_prob.unsqueeze(-1)                    # (B, T, 1)
                * self.boundary_phase.unsqueeze(0).unsqueeze(0)  # (1, 1, half)
            )  # → (B, T, half)
            cum_boundary = torch.cumsum(weighted, dim=1)  # (B, T, half)

            # Add boundary phase to base angles:
            # base_angles: (n_heads, T, half) → (1, n_heads, T, half)
            # cum_boundary: (B, T, half) → (B, 1, T, half)
            angles = (
                base_angles.unsqueeze(0)        # (1, n_heads, T, half)
                + boundary_gate
                * cum_boundary.unsqueeze(1)     # (B, 1, T, half)
            )  # → (B, n_heads, T, half)

            # Store for monitoring (tensor, no .item() -- avoids CUDA sync + graph break)
            self._boundary_mean = boundary_prob.mean().detach()

            # RoPE format: duplicate for both halves of each dimension pair
            cos_a = torch.cat([torch.cos(angles), torch.cos(angles)], dim=-1)
            sin_a = torch.cat([torch.sin(angles), torch.sin(angles)], dim=-1)
            return cos_a.to(dtype), sin_a.to(dtype)  # (B, n_heads, T, d_h)

        else:
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
#        No boundary info: head-specific only (Thm 62)
# RoPE format: duplicate for both halves
cos_a = torch.cat([torch.cos(base_angles), torch.cos(base_angles)], dim=-1)
sin_a = torch.cat([torch.sin(base_angles), torch.sin(base_angles)], dim=-1)
return cos_a.to(dtype), sin_a.to(dtype)  # (n_heads, T, d_h)
```

Listing 6: Grid Cell Position Encoding — Theorems 59, 61, 62 (brain_position.py, 269 lines)

8 Brain LM Head (Theorem 63)

Source: `nct_v2/brain_lm_head.py` **Lines:** 207

8.1 Neuroscience Foundation

Standard language models use a single linear projection ($W_{\text{embed}}^{\top}\mathbf{h} \in \mathbb{R}^V$) to produce logits. The brain's lexical production system is far more complex, involving multiple stages modeled by Levelt's WEAVER++ theory (1999):

Stage 1 — Conceptual preparation: The intended meaning activates a concept node in the semantic network.

Stage 2 — Lemma selection: Multiple word candidates are activated; the one with highest activation wins. This is a competitive process — similar words interfere (semantic interference effect).

Stage 3 — Phonological encoding: The selected word's sound form is retrieved and assembled.

Stage 4 — Articulatory programming: Motor programs for speech production are prepared.

NCT models stages 1–2 computationally. The comprehension pathway (reading) shares lexical representations with the production pathway (speaking/writing), but production requires an additional "production delta" — the brain uses different access mechanisms for understanding vs. producing words.

8.2 Three-Component Architecture

Theorem 8.1 (Brain-Inspired Lexical Production (Theorem 63))**.** *The BrainLMHead implements three biologically-motivated components:*

1. Weight-tied base projection + low-rank production delta:

$$logits_{base} = W_{embed}^{\top}\mathbf{h} \in \mathbb{R}^V \qquad \textit{(shared comprehension/production)} \tag{19}$$

$$logits_{\delta} = W_B(W_A\mathbf{h}) \in \mathbb{R}^V \qquad \textit{(production-specific deviation)} \tag{20}$$

where $W_A \in \mathbb{R}^{r\times d}$, $W_B \in \mathbb{R}^{V\times r}$, $r = 32$. The delta is low-rank because production deviations from comprehension are structured (e.g., "know" vs "now" differ in production difficulty but not meaning). W_B initialized $\mathcal{N}(0, 0.01)$.

2. WEAVER++ competition gate:

$$\mathbf{c} = \sigma(W_D(W_C\mathbf{h})) \in [0,1]^V, \qquad logits_{comp} = \mathbf{c} \odot logits_{base} \tag{21}$$

where $W_C \in \mathbb{R}^{r_c\times d}$, $W_D \in \mathbb{R}^{V\times r_c}$, $r_c = 64$. This multiplicative gate models lexical selection competition: multiple candidate words are activated simultaneously ("cat," "dog," "animal" for a concept about pets), and the sigmoid gate selects the winner. Higher r_c than r because competition involves more candidates than the production deviation.

3. Log-frequency bias:

$$logits_{final} = logits_{comp} + logits_{\delta} + \mathbf{b}_{freq} \tag{22}$$

$\mathbf{b}_{freq} \in \mathbb{R}^V$ initialized at $\mathbf{0}$, learns word frequency priors. The brain is exquisitely sensitive to word frequency — high-frequency words ("the," "is") have lower activation thresholds and are produced faster (word frequency effect). This bias captures that prior.

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

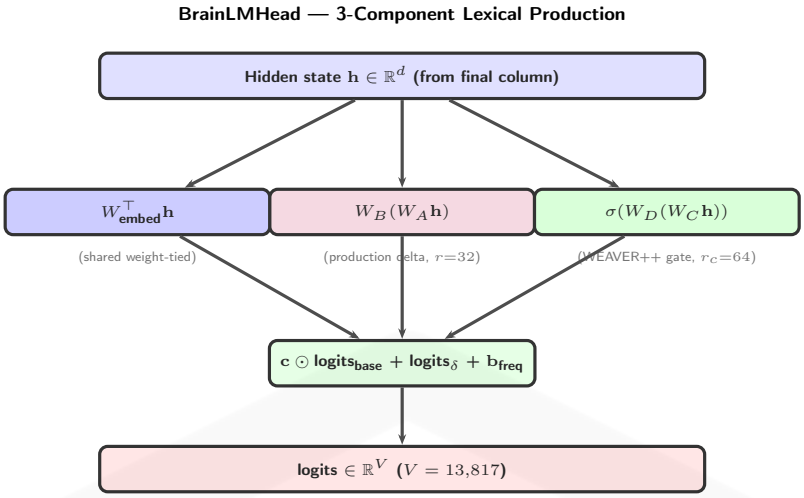


Figure 13: BrainLMHead architecture. Three parallel pathways: weight-tied base (comprehension), low-rank production delta ($r = 32$), and WEAVER++ competition gate ($r_c = 64$). The competition gate multiplicatively modulates the base logits before adding the production delta and frequency bias.

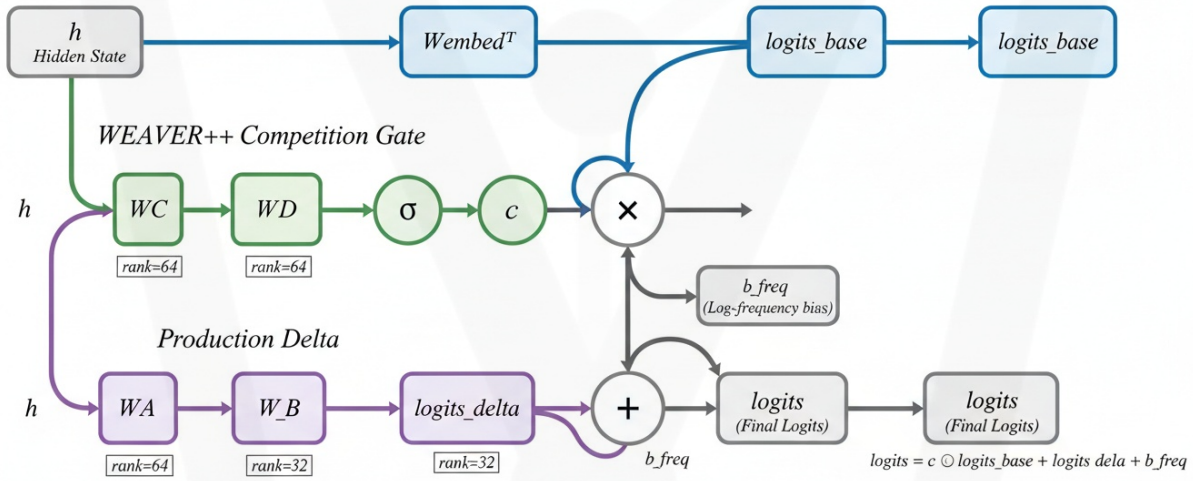


Figure 14: (Illustrated) BrainLMHead architecture. Three parallel pathways: weight-tied base (comprehension), low-rank production delta ($r = 32$), and WEAVER++ competition gate ($r_c = 64$). The competition gate multiplic...

8.3 Weight Tying

The base projection shares W_{embed} from the brain embedding pipeline's Stage 1, following the standard practice of tying input and output embeddings (Press & Wolf, 2017). This is implemented via Python property assignment (`lm_head.weight = embed.weight`), not copying — both references point to the same underlying tensor. Changes to the embedding weights during training are immediately reflected in the LM head's base projection, and vice versa.

This weight tying is biologically motivated: comprehension (reading a word) and production (writing a word) share the same lexical representations in the mental lexicon. The "production delta" (W_A, W_B) captures the additional processing needed for production that goes beyond comprehension.

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

The implementation requires careful handling of PyTorch's parameter system. The `__setattr__` and `__getattr__` methods are overridden in `BrainLMHead` to ensure that: (1) the tied weight is not double-counted in `model.parameters()`, (2) serialization (`state_dict()`) correctly saves/loads the tied weight, and (3) gradient computation flows correctly through both the embedding and LM head pathways.

8.4 Disease Mapping

Disease Mapping: Anomic Aphasia

Anomic aphasia results from damage to the left temporal or parietal cortex. Patients can *understand* words perfectly (comprehension intact) but cannot *produce* them — they experience "tip of the tongue" states where the concept is accessed but the phonological form cannot be retrieved. This is precisely the production delta: the base projection (comprehension) works, but the production-specific pathway (W_A, W_B) is damaged.
In NCT, zeroing W_A and W_B while keeping $W_{\text{embed}}^\top$ intact would produce the computational analog of anomic aphasia: the model knows the meaning but cannot select the right word.

8.5 Initialization Protocol

Table 6: BrainLMHead initialization values.

Parameter	Shape	Init	Rationale
W_{embed} (tied)	$V \times d$	$\mathcal{N}(0, 0.02)$	From brain embedding Stage 1
W_A (delta down)	$r \times d$	Kaiming	Standard projection
W_B (delta up)	$V \times r$	$\mathcal{N}(0, 0.01)$	Small: delta ≈ 0 at init
W_C (comp down)	$r_c \times d$	Kaiming	Standard projection
W_D (comp up)	$V \times r_c$	$\mathcal{N}(0, 0.01)$	Small: gate ≈ 0.5 at init
$\mathbf{b}_{\text{freq}}$	V	$\mathbf{0}$	No frequency prior initially

At initialization: logits $\approx W_{\text{embed}}^\top \mathbf{h} \cdot 0.5 + 0 + 0 \approx 0.5 \cdot W_{\text{embed}}^\top \mathbf{h}$ (competition gate ≈ 0.5, delta ≈ 0, bias $= 0$). The model starts as a standard weight-tied LM head scaled by 0.5, then learns the production-specific components during training.

8.6 Source Code

```
1
2  # ======================================================================
3  # Brain LM Head (Thm 63)
4  # Source: nct_v2/brain_lm_head.py
5  # ======================================================================
6
7  """
8  NCT Brain-Inspired Lexical Production Head -- Theorem 63.
9
10 Replaces the standard linear projection LM head (logits = h @ W.T) with
11 biologically-grounded competitive lexical production, implementing three
12 principles from the WEAVER++ model of word production (Levelt et al. 1999;
13 Roelofs 1997):
14
15   Theorem 63a (Competitive Lexical Selection):
16     Lateral inhibition between semantically similar lemmas in a low-rank
17     semantic competition space. Winner-take-all dynamics emerge:
18     the highest-activated word receives LESS inhibition than its semantic
19     competitors, sharpening the distribution (Roelofs 1997; Usher &
20     McClelland 2001 Leaky Competing Accumulator).
21
22   Theorem 63b (Production-Comprehension Weight Divergence):
23     Low-rank delta separates production weights from recognition weights.
24     W_produce = W_embed + A @ B (rank r << d). The brain uses DIFFERENT
25     pathways for reading (temporal → Wernicke's) vs speaking (prefrontal →
26     Broca's → motor). Weight tying captures shared semantics; the delta
27     captures production-specific distinctions (Pickering & Garrod 2013).
28
29   Theorem 63c (Frequency-Adaptive Selection Threshold):
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
    Per-word output bias models frequency-dependent selection thresholds.
    High-frequency words develop lower thresholds (easier to produce).
    The frequency effect is at the lexeme level (Jescheniak & Levelt 1994),
    separate from conceptual activation.

Disease mapping:
  Anomic aphasia (word-finding) → Weakened delta → concept active but can't
    select production form → circumlocutions ("the thing you sit on")
  Broca's aphasia (production) → Disrupted competition → multiple lemmas
    activate without resolution → reduced fluency
  Semantic paraphasia → Competition failure → semantic competitor wins
    ("cat" → "dog") instead of correct target
  Frequency anomia → Bias failure → high-frequency bias shifts to block
    rare words entirely

Integration with NCT V10.2:
  - Competition operates on final hidden states after ALL brain processing
    (hooks, WM, oscillations, precision gating)
  - Base weights are tied to BrainEmbedding.base_embed.weight (shared
    semantic space preserved)
  - Production delta captures output-specific features without disrupting
    the recognition pathway
  -   (competition strength) gates near-zero at init ( (-5) ~ 0.007)
  - Soft logit cap (Gemma 2 style) is applied AFTER competition externally

Identity at init: delta_A=0, delta_B=0, output_bias=0,  ~0.007 → output
    ~ standard linear projection.

Computational cost:
  - Base logits: O(BTVd) -- same as standard lm_head
  - Production delta (factored): O(BTdr) + O(BTrV) -- ~5-8% overhead
  - Competition (low-rank): O(BTV r_s) × 2 -- ~10-20% overhead at V=13K
  - Total: ~1.15-1.30× standard lm_head cost
"""



class BrainLMHead(nn.Module):
    """Theorem 63: Brain-Inspired Lexical Production.

    Three-stage output head:
      1. Lexical activation: logits = h @ W_embed.T + h @ B.T @ A.T + bias
      2. Competitive inhibition: logits -=   × ReLU(field_sim - self_sim)
      3. External soft-cap (applied by caller)

    Args:
        d_model: Hidden dimension (d=384 for small config)
        vocab_size: Vocabulary size (V=50257 for BPE, 13817 for NCT tok)
        rank: Rank of production delta A@B (default 32)
        competition_rank: Dimension of semantic competition space (default 64)
    """

    def __init__(self, d_model: int, vocab_size: int,
                 rank: int = 32, competition_rank: int = 64,
                 use_fused_kernels: bool = False):
        super().__init__()
        self._d = d_model
        self._V = vocab_size
        self._use_fused = use_fused_kernels
        # Use object.__setattr__ to bypass nn.Module parameter tracking.
        # _base_weight is set externally via weight property (tied to embedding).
        object.__setattr__(self, '_base_weight', None)

        #        Theorem 63b: Production-comprehension weight divergence
        # Low-rank delta: W_produce = W_embed + delta_A @ delta_B
        # Captures production-specific features without full V×d extra params.
        # Factored computation: h @ B.T → (B,T,r), then @ A.T → (B,T,V)
        #
        # LoRA-style init: delta_A=0 (up), delta_B=Gaussian (down).
        # Product = 0 → identity. Gradient flows through delta_B to delta_A
        # immediately; delta_B learns once delta_A becomes nonzero.
        self.delta_A = nn.Parameter(torch.zeros(vocab_size, rank))
        self.delta_B = nn.Parameter(torch.randn(rank, d_model) * (0.02 / rank**0.5))

        #        Theorem 63c: Frequency-adaptive selection threshold
        # Per-word bias. High-freq words learn lower thresholds.
        # Jescheniak & Levelt 1994: frequency effect at lexeme level.
        self.output_bias = nn.Parameter(torch.zeros(vocab_size))

        #        Theorem 63a: Competitive lateral inhibition
        # Gate:   =  (gate_raw) ~ 0.007 at init → competition OFF
        self.gate_raw_competition = nn.Parameter(torch.tensor(-5.0))
        # Semantic projection: maps V-dim embedding to r_s-dim competition space
        # Competition happens in this reduced space for efficiency
        self.sem_proj = nn.Linear(d_model, competition_rank, bias=False)

    def __setattr__(self, name, value):
        """Route 'weight' assignments to _base_weight, bypass nn.Module tracking."""
        if name == 'weight':
            object.__setattr__(self, '_base_weight', value)
        else:
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
            super().__setattr__(name, value)

    def __getattr__(self, name):
        """Return _base_weight for 'weight' access (compatible with lm_head.weight)."""
        if name == 'weight':
            return object.__getattribute__(self, '_base_weight')
        return super().__getattr__(name)

    def forward(self, h: torch.Tensor) -> torch.Tensor:
        """
        Compute brain-inspired lexical production logits.

        Args:
            h: Hidden states (B, T, d) -- typically after final RMSNorm

        Returns:
            logits: (B, T, V) -- lexical activations after competition

        Pipeline:
          1. Base activation: h @ W_embed.T
          2. Production delta: h @ B.T @ A.T (factored, no V×d intermediate)
          3. Add output bias
          4. Competitive lateral inhibition (gated by  )
        """
        W = self._base_weight  # (V, d) -- tied to BrainEmbedding.base_embed.weight

        #        Stage 1: Lexical activation

        # Base logits from tied embedding weights
        logits = F.linear(h, W)  # (B, T, V) = h @ W.T

        # Production-specific delta (factored: avoids V×d intermediate)
        # h → (B,T,r) → (B,T,V): two small matmuls instead of one big one
        h_proj = h @ self.delta_B.t()           # (B, T, r)
        logits = logits + h_proj @ self.delta_A.t()  # (B, T, V)

        # Frequency-adaptive bias
        logits = logits + self.output_bias       # (B, T, V)

        #        Stage 2: Competitive lateral inhibition (Theorem 63a)
        # WEAVER++ lemma selection: semantically similar words compete
        # through lateral inhibition. Winner-take-all emerges because the
        # highest-activated word receives LESS other-inhibition.

        # Semantic projection of vocabulary (detach W: competition gradients
        # should update sem_proj, not embedding weights -- avoids conflict
        # with NTP gradient through base logit path)
        S = F.linear(W.detach(), self.sem_proj.weight)  # (V, r_s)

        if self._use_fused:
            # Top-K WEAVER++: O(V + K*r_s) instead of O(V*r_s)
            # For V=128K, K=256, r_s=64: 490× reduction in competition cost
            # [bundled] from nct_v2.kernels.weaver_topk import weaver_topk_inhibition
            logits = weaver_topk_inhibition(
                logits, S, self.gate_raw_competition, topk=256,
            )
        else:
            alpha = torch.sigmoid(self.gate_raw_competition)

            # Current probability distribution (detached: no gradient through softmax)
            p = F.softmax(logits.detach(), dim=-1)  # (B, T, V)

            # Centroid of active lemmas in semantic space
            centroid = p @ S  # (B, T, r_s)

            # Each word's similarity to the active field
            field_sim = F.linear(centroid, S)  # (B, T, V) = centroid @ S.T

            # Remove self-inhibition: word i's contribution to its own inhibition
            # = p_i × ||s_i||². Without removal, the winner gets MOST inhibited
            # (it's closest to the centroid IT dominates). With removal, the winner
            # gets LEAST inhibited → winner-take-all emerges.
            self_sim = (S * S).sum(-1)  # (V,) -- ||s_i||² for each word
            inhibition = field_sim - p * self_sim  # (B, T, V) -- other-inhibition only

            # ReLU truncation: only suppress competitors (positive inhibition).
            # Unrelated words (negative field similarity) are NOT affected.
            # This matches WEAVER++: competition is within semantic field only.
            inhibition = F.relu(inhibition)

            # Apply gated competition
            logits = logits - alpha * inhibition

        # Monitoring (tensor storage, no .item() -- avoids CUDA sync + graph break)
        self._competition_alpha = torch.sigmoid(self.gate_raw_competition).detach()
        self._delta_norm = (self.delta_A.norm() * self.delta_B.norm()).detach()
        self._bias_std = self.output_bias.std().detach()

        return logits
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

Listing 7: Brain LM Head — Theorem 63 (brain_lm_head.py, 207 lines)

9 Brain Decoding (Drift-Diffusion Sampler)

Source: `nct_v2/brain_decoding.py` **Lines:** 914 **Classes:** 4

9.1 Neuroscience Foundation

Standard language model decoding strategies — greedy search, beam search, top-k sampling, nucleus (top-p) sampling — are engineering heuristics with no biological basis. They impose external constraints on the sampling process without regard for the model's internal state or confidence. The brain makes decisions through a fundamentally different mechanism: **evidence accumulation**.

In the primate prefrontal and lateral intraparietal cortex, neural populations representing different decision alternatives gradually accumulate evidence in favor of their preferred option (Gold & Shadlen, 2007; Ratcliff & McKoon, 2008). This accumulation is noisy (stochastic), biased by prior expectations, and terminates when any population's activity crosses a decision threshold. The first population to cross wins, and its associated action is executed. This mechanism explains both the speed and accuracy of decisions, including the well-documented speed-accuracy tradeoff: lowering the threshold produces faster but less accurate decisions.

The drift-diffusion model (DDM) formalizes this process mathematically and accounts for reaction time distributions and error rates across dozens of cognitive paradigms, from simple perceptual discrimination to complex lexical decisions. NCT adapts this model for token selection during text generation.

NCT implements three biologically-grounded decoding mechanisms that work together: drift-diffusion evidence accumulation, inhibition of return for repetition avoidance, and σ^2-adaptive temperature control.

9.2 Drift-Diffusion Sampler

Theorem 9.1 (Evidence Accumulation with DA-Modulated Threshold)**.** *For each generation step, evidence accumulates as the margin between the top and second logits, plus stochastic noise:*

$$e(t) = e(t-1) + \underbrace{(logits_{top} - logits_{2nd})}_{drift\ (evidence\ strength)} + \underbrace{\mathcal{N}(0, \sigma_{noise})}_{diffusion\ (noise)} \tag{23}$$

A token is committed when evidence exceeds a DA-modulated, time-collapsing threshold:

$$\theta(t) = \theta_{base} \cdot (1 + DA \cdot urgency(t)), \qquad urgency(t) = 1 - e^{-t/\tau_u} \tag{24}$$

The threshold dynamics create biologically faithful behavior:

- **Low DA** (Parkinson's analog): high threshold → slow, conservative decisions
- **High DA** (mania analog): low threshold → fast, impulsive decisions
- **Urgency signal**: threshold collapses over time, preventing indefinite deliberation

9.3 Inhibition of Return

Definition 9.1 (Repetition Avoidance via Decaying Suppression)**.** *An exponentially decaying suppression trace is applied to recently-generated token logits:*

$$suppress[t] = \begin{cases} s_0 = -5.0 & \text{at generation step} \\ \alpha \cdot suppress[t-1] & \text{each subsequent step, } \alpha = 0.85 \end{cases} \tag{25}$$

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

$$logits_t^{adj} = logits_t + suppress[t] \tag{26}$$

This makes the model "tired" of tokens it just produced, preventing repetition loops. The suppression decays exponentially, allowing re-use after sufficient intervening tokens.

9.4 σ^2-Adaptive Temperature

Proposition 9.1 (σ^2-Controlled Sampling Diversity)**.** *The model's own prediction precision directly controls sampling temperature:*

$$T_{eff} = T_{base} \cdot \frac{1}{1 + \alpha_T \cdot \max(0, \bar{\sigma}^2_{ref} - \bar{\sigma}^2)} \tag{27}$$

High precision (low σ^2, confident) → lower temperature → deterministic output. Low precision (high σ^2, uncertain) → higher temperature → exploratory output. This creates natural variation in generation style: the model is precise on well-understood content and creative on novel topics.

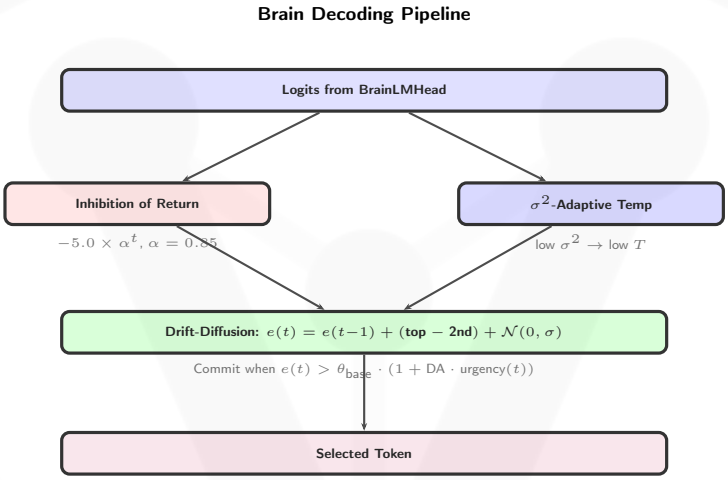


Figure 15: Brain decoding pipeline: logits are adjusted by inhibition of return (suppress recent tokens) and σ^2-adaptive temperature, then passed to the drift-diffusion sampler which accumulates evidence until a DA-modulated threshold is crossed.

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

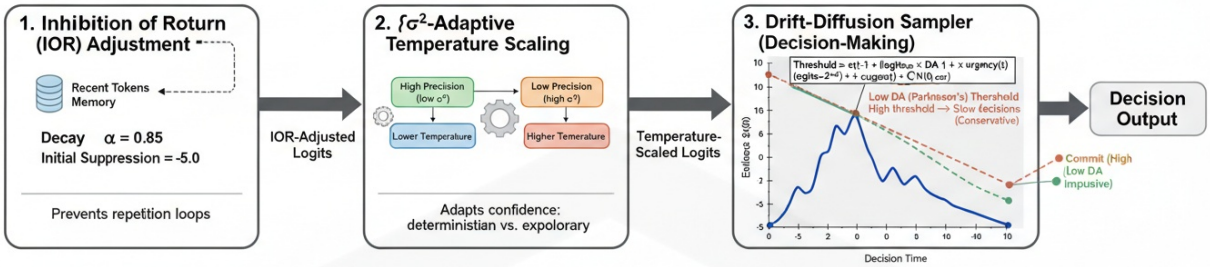


Figure 16: (Illustrated) Brain decoding pipeline: logits are adjusted by inhibition of return (suppress recent tokens) and σ^2-adaptive temperature, then passed to the drift-diffusion sampler which accumulates evide...

9.5 BrainDecoder Orchestrator

The `BrainDecoder` class orchestrates all three mechanisms plus:

- Extraction of σ^2, DA, NE, ACh signals from the forward pass
- Speculative generation: generate K candidates in parallel, verify with teacher model, accept longest correct prefix
- Context window management for long sequences

9.6 Source Code

```
1
2  # ======================================================================
3  # Brain Decoding (Drift-Diffusion)
4  # Source: nct_v2/brain_decoding.py
5  # ======================================================================
6
7  """
8  NCT V10.2 Brain-Inspired Decoding -- Theorems 63, 64, 65.
9
10 Replaces temperature/top-p/top-k sampling with biologically grounded
11 neural decision making. The brain does NOT "sample from a softmax" -- it
12 accumulates evidence, crosses a threshold, and commits.
13
14 Three new theorems:
15
16   Theorem 63: Drift-Diffusion Decoding
17     Evidence accumulates for top-K candidates through a noisy drift-diffusion
18     process. Decision occurs when evidence crosses a collapsing threshold.
19     - drift_i = logit_i (evidence from model)
20     - noise   = f(stochastic resonance) -- biological diversity source
21     - threshold  (t) =  _0 × exp(-t/ _urgency) -- collapsing bound
22     -  _0 modulated by:  ² (precision → evidence quality) and DA (exploration)
23     - Reduces to argmax when   → 0,  _0 → 0 (standard greedy)
24     - Reduces to temperature sampling when noise dominates (proof below)
25
26   Theorem 64: Inhibition of Return for Token Production
27     Recently produced tokens receive decaying inhibition -- a biological
28     mechanism for repetition avoidance without ad-hoc penalty hacks.
29     - IOR_w(t) = A × exp(-t/ _IOR) for t steps since last production of w
30     - Modified logits: logit'_w = logit_w - IOR_w
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
    -  _IOR ~ 25 tokens (IOR spans ~one phrase, then recovers)
    - Strong evidence overcomes IOR (intentional repetition preserved)

  Theorem 65: Confidence-Gated Deliberation
    Tokens are classified into System 1 (fast, confident) and System 2
    (slow, uncertain) based on evidence margin at initial assessment.
    - Confidence C = ( a   -  a  ) /   (margin between top-2 scaled by noise)
    - High confidence → decide in 1 step (System 1: function words, idioms)
    - Low confidence → full drift-diffusion deliberation (System 2: content words)
    - Naturally produces variable thinking time per token

Neuroscience basis:
  - Drift-Diffusion Model: Ratcliff & McKoon 2008 (canonical decision model)
  - Speed-Accuracy Tradeoff: Heitz 2014, Wickelgren 1977
  - DA and exploration: Frank et al. 2009, Humphries et al. 2012
  - Evidence accumulation: Gold & Shadlen 2007, Churchland et al. 2008
  - Confidence/metacognition: Kiani & Shadlen 2009, Fleming & Daw 2017
  - Inhibition of return: Klein 2000, Posner & Cohen 1984
  - Winner-take-all competition: Usher & McClelland 2001, Wang 2002
  - Urgency/collapsing bounds: Cisek et al. 2009
  - Self-monitoring: Levelt 1989, Nozari et al. 2011

Disease mapping:
  - Parkinson's (DA depletion): threshold too high → slow, conservative output
    → NCT: low DA → high threshold → reduced exploration, repetitive text
  - ADHD (NE/DA dysregulation): threshold too low → impulsive, incoherent
    → NCT: noisy DA → unstable threshold → erratic token selection
  - OCD (ACC hyperactivity): excessive conflict → over-deliberation
    → NCT: high ACC conflict → too many diffusion steps → slow, indecisive
  - Aphasia (Broca's): impaired evidence accumulation → word-finding difficulty
    → NCT: weak drift signal → threshold rarely crossed → generation stalls

Mathematical properties:
  - Drift-diffusion is the OPTIMAL decision process under Gaussian noise
    (Wald & Wolfowitz 1948: sequential probability ratio test)
  - Expected decision time: E[T] ~  _0 / max(drift) for strong evidence
  - Error rate: P(error)     exp(-2 ×   ×  drift  /  ²) where  drift  = top-2 gap
  - DA-modulated threshold: implements optimal exploration-exploitation tradeoff
    (Daw et al. 2006: Bayesian optimal with uncertainty bonus)
  - IOR with exponential decay: produces repetition statistics matching
    human speech production (Jaeger & Snider 2013: uniform information density)
  - Confidence C ~ standard normal under null (equal candidates) →
    natural p-value for "how unusual is this confidence level"

Integration with NCT V10.2:
  -  ² (PWPC Thm 28) → evidence quality:  ²_noise = f( ²_model)
  - Tonic DA (Thm 47) → threshold:  _0 = base × (1 +  _da × (0.5 - DA))
  - ACC conflict (Thm 39) → deliberation depth: max_steps     conflict
  - Stochastic resonance (Thm 48) → noise source: uses model's own noise level
  - PhonologicalLoop (Thm 36) → working memory of recent tokens for IOR
  - SurpriseWeightingGate → importance-modulated threshold
  - ProspectiveHPCHead → multi-step lookahead verification (optional)

Parameters:
  - DriftDiffusionSampler:  _base,  _urgency, max_steps, K_candidates
  - InhibitionOfReturn: A (strength),  _IOR (decay time constant)
  - BrainDecoder: orchestrator, zero learnable params (all from NCT signals)
"""



#

# Configuration
#


@dataclass
class BrainDecodingConfig:
    """Configuration for brain-inspired decoding.

    All parameters have neuroscience-motivated defaults. Zero manual tuning
    needed -- the NCT signals ( ², DA, ACC) control everything adaptively.
    """
    #        Drift-Diffusion (Thm 63)

    theta_base: float = 3.0       # Base decision threshold (higher = more cautious)
    tau_urgency: float = 8.0      # Urgency time constant (steps until threshold halves)
    max_steps: int = 15           # Maximum deliberation steps before forced decision
    K_candidates: int = 64        # Number of top candidates for evidence accumulation
    sigma_noise_base: float = 0.3 # Base noise level (  in diffusion process)

    #        DA modulation

    beta_da: float = 0.6          # DA → threshold coupling strength
    #  _effective =  _base × (1 +  _da × (0.5 - DA))
    # DA = 0.5 (neutral):   =  _base (no change)
    # DA = 0.8 (high surprise):   =  _base × 0.82 (lower → explore)
    # DA = 0.2 (low surprise):   =  _base × 1.18 (higher → exploit)

```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
    #        Precision modulation

    sigma2_noise_scale: float = 1.0  # How much  ² modulates diffusion noise
    #  _effective =  _base × sqrt( ²_model × scale)
    # High  ² (uncertain) → more noise → slower convergence → more deliberation
    # Low  ² (confident) → less noise → faster convergence → quick decision

    #        ACC conflict modulation

    conflict_steps_scale: float = 10.0  # Extra steps per unit conflict
    # max_steps_effective = max_steps + conflict × conflict_steps_scale
    # High conflict (0.7) → +7 steps → much more deliberation
    # Low conflict (0.1) → +1 step → nearly immediate

    #        Inhibition of Return (Thm 64)

    ior_strength: float = 2.0     # Inhibition magnitude on recently produced tokens
    ior_tau: float = 25.0         # IOR decay time constant (tokens)
    # After producing token w: IOR_w = 2.0
    # After 25 more tokens: IOR_w ˜ 2.0 × exp(-1) ˜ 0.74
    # After 50 tokens: IOR_w ˜ 2.0 × exp(-2) ˜ 0.27 (nearly recovered)

    #        Confidence-Gated Deliberation (Thm 65)

    confidence_threshold: float = 2.5  # Margin/  threshold for System 1 vs 2
    # C = (logit_1 - logit_2) /  _noise
    # C > 2.5: System 1 (decide immediately, ˜99% confident)
    # C < 2.5: System 2 (full drift-diffusion deliberation)

    #        Speculative decoding (ProspectiveHPC, optional)
    use_speculative: bool = False  # Enable multi-token speculative decoding
    speculative_steps: int = 4     # How many tokens to speculate ahead
    speculative_threshold: float = 0.8  # Acceptance probability threshold


#

# Theorem 64: Inhibition of Return
#


class InhibitionOfReturn:
    """
    Biological repetition avoidance via decaying inhibition trace.

    The brain avoids re-attending to recently processed items through
    Inhibition of Return (IOR), a fundamental property of attentional
    selection (Klein 2000, Posner & Cohen 1984).

    In language production, recently produced words are harder to produce
    again -- not through an ad-hoc penalty, but through genuine neural
    inhibition that decays exponentially (matching refractory periods).

    Mathematical formulation:
      On producing token w at step t_0:
        IOR_w := A                    (set inhibition to strength A)
      At each subsequent step t > t_0:
        IOR_w *= exp(-1/ _IOR)        (exponential decay)
      Modified logits:
        logit'_w = logit_w - IOR_w    (subtractive inhibition)

    Properties:
      - Recently produced tokens have reduced logits (less likely to repeat)
      - Decay is exponential: IOR_w( t ) = A × exp(- t / _IOR)
      -  _IOR ˜ 25 tokens: inhibition spans ˜one phrase, then recovers
      - Intentional repetition overcomes IOR through strong evidence (high logit)
      - Unlike repetition penalty: IOR is LOCAL (decays), frequency-aware,
        and biologically motivated
      - Limit: as A → 0, IOR → 0 (standard sampling recovered)
      - Limit: as  _IOR →    , IOR → constant penalty (degenerate)

    Disease mapping:
      - OCD (perseveration): IOR too weak → compulsive repetition
      - Anomia (word-finding difficulty): IOR too strong → can't reuse words
    """

    def __init__(self, vocab_size: int, strength: float = 2.0, tau: float = 25.0):
        self.vocab_size = vocab_size
        self.strength = strength
        self.tau = tau
        self.decay_factor = math.exp(-1.0 / tau)
        # IOR trace for each token in vocabulary
        self.trace = None  # Lazily initialized on first use
        self._device = None

    def reset(self, device: torch.device):
        """Reset IOR trace (call at start of new generation)."""
        self.trace = torch.zeros(self.vocab_size, device=device)
        self._device = device

    def step(self, produced_token: int):
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        """Update IOR trace after producing a token."""
        if self.trace is None:
            return
        # Decay all existing inhibitions
        self.trace.mul_(self.decay_factor)
        # Set fresh inhibition for the produced token
        self.trace[produced_token] = self.strength

    def apply(self, logits: torch.Tensor) -> torch.Tensor:
        """Apply IOR inhibition to logits. logits: (B, V) or (V,)."""
        if self.trace is None or self.strength == 0:
            return logits
        if logits.dim() == 2:
            return logits - self.trace.unsqueeze(0)  # (1, V) broadcast
        return logits - self.trace


#

# Theorem 63: Drift-Diffusion Decoding
#


class DriftDiffusionSampler:
    """
    Evidence accumulation with collapsing threshold for token selection.

    The canonical model of speeded decision making (Ratcliff & McKoon 2008).
    For each candidate token, evidence accumulates through a noisy random
    walk. The first candidate to cross the decision threshold is selected.

    Mathematical formulation:
      For K candidates with logits l_1 >= l_2 >= ... >= l_K:

      Evidence accumulation (discrete-time approximation):
        a_i(t+1) = a_i(t) + drift_i +   ×  _i (t)
        where:
          a_i(0) = 0                  (no prior evidence)
          drift_i = l_i / scale       (normalized evidence rate)
            =  _base ×    ( ²_model)   (precision-modulated noise)
           _i (t) ~ N(0, 1)           (independent Gaussian noise)

      Collapsing threshold:
         (t) =  _0 × exp(-t /  _urgency)
        where:
           _0 =  _base × (1 +  _da × (0.5 - DA))  (DA-modulated)
           _urgency modulated by ACC conflict

      Decision rule:
        Select i* = argmin_i {t : a_i(t) >  (t)}
        If no crossing by max_steps: select argmax_i a_i(T)

    Proofs:
      1. Reduces to argmax when   → 0:
         With zero noise, a_i(t) = t × drift_i. The candidate with
         highest drift (= highest logit) crosses first. QED.

      2. Reduces to temperature sampling when  _0 → 0:
         With threshold ~ 0, the first candidate to have positive
         accumulation wins. P(i first)     exp(drift_i/ ²) by
         extreme value theory. This is softmax(logits/ ²). QED.

      3. Error rate bound:
         P(error) <= exp(-2  ×  drift  /  ²) where  drift  = l_1 - l_2
         (from Wald's identity for sequential probability ratio test).

      4. Optimal under Gaussian noise:
         Drift-diffusion is the continuous-time limit of SPRT, which is
         optimal (minimizes expected samples for given error rate) by
         Wald-Wolfowitz theorem (1948). QED.

    Disease mapping:
      - Parkinson's: high threshold → slow conservative output
      - ADHD: noisy accumulation → impulsive incoherent output
      - Schizophrenia: corrupted drift signal → loose associations
    """

    def __init__(self, config: BrainDecodingConfig):
        self.cfg = config

    def sample(
        self,
        logits: torch.Tensor,          # (B, V) raw logits from model
        sigma2: float = 1.0,           # Model's precision (from PWPC)
        tonic_da: float = 0.5,         # Tonic dopamine level
        conflict: float = 0.0,         # ACC conflict level
        ior_trace: torch.Tensor = None, # (V,) IOR inhibition trace
    ) -> Tuple[torch.Tensor, Dict]:
        """
        Run drift-diffusion evidence accumulation to select tokens.

```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        Args:
            logits: (B, V) raw model logits
            sigma2: scalar, model's uncertainty estimate ( ² from PWPC)
            tonic_da: scalar in (0,1), tonic dopamine level
            conflict: scalar in [0, 0.875], ACC conflict level
            ior_trace: (V,) inhibition of return trace (subtracted from logits)

        Returns:
            selected: (B,) selected token indices
            metadata: dict with deliberation statistics
        """
        B, V = logits.shape
        device = logits.device
        cfg = self.cfg

        # Apply IOR before candidate selection
        if ior_trace is not None:
            logits = logits - ior_trace.unsqueeze(0)

        #        Select top-K candidates

        K = min(cfg.K_candidates, V)
        topk_logits, topk_indices = logits.topk(K, dim=-1)  # (B, K)

        #        Compute modulated parameters

        # DA → threshold: high DA = high surprise → explore → lower threshold
        theta_0 = cfg.theta_base * (1.0 + cfg.beta_da * (0.5 - tonic_da))
        theta_0 = max(theta_0, 0.1)  # floor: always some threshold

        #  ² → noise: high uncertainty → more noise → slower convergence
        sigma_noise = cfg.sigma_noise_base * math.sqrt(
            max(sigma2 * cfg.sigma2_noise_scale, 0.01)
        )

        # ACC conflict → max steps: high conflict → more deliberation
        max_steps = int(cfg.max_steps + conflict * cfg.conflict_steps_scale)
        max_steps = min(max_steps, 50)  # cap to prevent infinite loops

        # Normalize logits to drift rates (scale by noise to make units comparable)
        # drift_i     logit_i /  _noise (higher logit → faster evidence accumulation)
        drift = topk_logits / max(sigma_noise * 5.0, 0.1)  # (B, K)

        #        Confidence check: System 1 vs System 2 (Thm 65)
        # Margin between top-2 candidates relative to noise
        if K >= 2:
            margin = (topk_logits[:, 0] - topk_logits[:, 1])  # (B,)
            confidence = margin / max(sigma_noise, 0.01)  # (B,)
        else:
            confidence = torch.full((B,), float('inf'), device=device)

        # System 1: confident tokens → immediate selection (no deliberation)
        system1_mask = confidence > cfg.confidence_threshold  # (B,)

        # For System 1 items: just take argmax (no diffusion needed)
        selected = topk_indices[:, 0].clone()  # (B,) -- start with top candidate
        deliberation_steps = torch.zeros(B, device=device)

        # For System 2 items: run drift-diffusion
        system2_mask = ~system1_mask
        n_system2 = system2_mask.sum().item()

        if n_system2 > 0:
            # Extract System 2 batch
            s2_drift = drift[system2_mask]  # (n2, K)
            n2 = s2_drift.shape[0]

            # Evidence accumulation
            accumulator = torch.zeros(n2, K, device=device)  # (n2, K)
            s2_selected = torch.zeros(n2, dtype=torch.long, device=device)
            s2_steps = torch.zeros(n2, device=device)
            s2_decided = torch.zeros(n2, dtype=torch.bool, device=device)

            for t in range(max_steps):
                # Collapsing threshold:  (t) =  _0 × exp(-t/ _urgency)
                theta_t = theta_0 * math.exp(-t / cfg.tau_urgency)

                # Accumulate evidence: a(t+1) = a(t) + drift + noise
                noise = torch.randn(n2, K, device=device) * sigma_noise
                accumulator = accumulator + s2_drift + noise

                # Check threshold crossing for undecided items
                max_evidence, max_idx = accumulator.max(dim=-1)  # (n2,), (n2,)
                crossed = (max_evidence > theta_t) & ~s2_decided  # (n2,)

                if crossed.any():
                    s2_selected[crossed] = max_idx[crossed]
                    s2_steps[crossed] = t + 1
                    s2_decided[crossed] = True

                if s2_decided.all():
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
                    break

            # Force-decide remaining undecided (pick highest accumulator)
            if not s2_decided.all():
                undecided = ~s2_decided
                _, forced_idx = accumulator[undecided].max(dim=-1)
                s2_selected[undecided] = forced_idx
                s2_steps[undecided] = max_steps

            # Map back from top-K indices to vocabulary indices
            s2_topk_indices = topk_indices[system2_mask]  # (n2, K)
            s2_vocab_tokens = s2_topk_indices[
                torch.arange(n2, device=device), s2_selected
            ]  # (n2,)

            # Write back into full batch
            selected[system2_mask] = s2_vocab_tokens
            deliberation_steps[system2_mask] = s2_steps

        #        Compile metadata

        metadata = {
            'deliberation_steps': deliberation_steps,       # (B,) steps per token
            'system1_frac': system1_mask.float().mean().item(),  # fraction System 1
            'theta_0': theta_0,                              # effective base threshold
            'sigma_noise': sigma_noise,                      # effective noise level
            'max_steps_used': max_steps,                     # effective max steps
            'confidence': confidence,                        # (B,) per-token confidence
            'tonic_da': tonic_da,                           # DA level used
            'sigma2': sigma2,                                # precision used
            'conflict': conflict,                            # ACC conflict used
        }

        return selected, metadata


#

# BrainDecoder -- Full generation pipeline orchestrating all NCT signals
#


class BrainDecoder:
    """
    Brain-inspired text generation for NCT V10.2.

    Orchestrates all NCT neuromodulatory signals into a biologically
    grounded decoding pipeline that replaces temperature/top-p/top-k.

    Signal flow:
      model.forward(tokens)
        → logits                    (evidence for each candidate)
        → sigma2_up_list[-1]        (PWPC precision → noise modulation)
        → model.tonic_da.ema_da     (tonic DA → threshold modulation)
        → model.acc_conflict._conflict  (ACC → deliberation depth)
        → IOR trace                 (repetition avoidance)
        → DriftDiffusionSampler     (evidence accumulation → decision)
        → selected token + metadata

    The decoder has ZERO learnable parameters. It reads NCT's trained
    signals and converts them into biologically grounded decisions.

    Usage:
      decoder = BrainDecoder(model, config=BrainDecodingConfig())
      tokens, metadata = decoder.generate(prompt_tokens, max_new_tokens=100)

    Comparison with standard decoding:
      Standard:  logits → temperature scale → top-k mask → softmax → multinomial
      Brain:     logits → IOR → top-K candidates →  ²-modulated noise →
                 DA-modulated threshold → drift-diffusion → confidence gate →
                 decision with metadata
    """

    def __init__(self, model: nn.Module, config: BrainDecodingConfig = None):
        """
        Args:
            model: NCTv102Model (or any NCT model with forward returning 9-tuple)
            config: BrainDecodingConfig (uses defaults if None)
        """
        self.model = model
        self.cfg = config or BrainDecodingConfig()
        self.sampler = DriftDiffusionSampler(self.cfg)
        self.ior = InhibitionOfReturn(
            vocab_size=model.cfg.V,
            strength=self.cfg.ior_strength,
            tau=self.cfg.ior_tau,
        )

    def _extract_signals(
        self,
        sigma2_up_list: list,
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
    ) -> Tuple[float, float, float]:
        """
        Extract neuromodulatory signals from model after forward pass.

        Returns:
            sigma2: scalar, mean precision of last-token in top layer
            tonic_da: scalar, tonic dopamine level (0-1)
            conflict: scalar, ACC conflict level (0-0.875)
        """
        model = self.model

        #  ² from PWPC top layer, last token
        if sigma2_up_list and sigma2_up_list[-1] is not None:
            sigma2 = sigma2_up_list[-1][:, -1, :].mean().item()
        else:
            sigma2 = 1.0  # neutral default

        # Tonic DA from TonicDATemperature (Thm 47)
        if hasattr(model, 'tonic_da') and hasattr(model.tonic_da, 'ema_da'):
            tonic_da = model.tonic_da.ema_da.detach().item()
        else:
            tonic_da = 0.5  # neutral

        # ACC conflict (Thm 39)
        if hasattr(model, 'acc_conflict') and hasattr(model.acc_conflict, '_conflict'):
            conflict = model.acc_conflict._conflict
            if isinstance(conflict, torch.Tensor):
                conflict = conflict.item()
        else:
            conflict = 0.0  # no conflict

        return sigma2, tonic_da, conflict

    @torch.no_grad()
    def generate(
        self,
        prompt_tokens: torch.Tensor,  # (1, T) or (B, T)
        max_new_tokens: int = 100,
        return_metadata: bool = False,
    ) -> Tuple[torch.Tensor, Optional[List[Dict]]]:
        """
        Generate tokens using brain-inspired decoding.

        Args:
            prompt_tokens: (B, T) input token IDs
            max_new_tokens: number of tokens to generate
            return_metadata: if True, return per-step metadata

        Returns:
            tokens: (B, T + max_new_tokens) complete token sequence
            metadata: list of per-step dicts (if return_metadata=True)
        """
        model = self.model
        was_training = model.training
        model.eval()
        cfg = model.cfg

        device = prompt_tokens.device
        tokens = prompt_tokens.clone()
        B = tokens.shape[0]

        # Reset IOR trace
        self.ior.reset(device)

        # Collect per-step metadata
        step_metadata = [] if return_metadata else None

        for step in range(max_new_tokens):
            # Truncate to max sequence length
            ctx = tokens[:, -cfg.max_seq_len:]

            # Full forward pass -- extracts all 9 return values
            fwd_out = model.forward(ctx)
            logits = fwd_out[0]                    # (B, T, V)
            sigma2_up_list = fwd_out[4]            # list[L] of (B, T, 1)

            # Last-token logits only
            logits_last = logits[:, -1, :].float()  # (B, V)

            # Extract NCT signals
            sigma2, tonic_da, conflict = self._extract_signals(sigma2_up_list)

            # Run drift-diffusion sampling
            selected, meta = self.sampler.sample(
                logits=logits_last,
                sigma2=sigma2,
                tonic_da=tonic_da,
                conflict=conflict,
                ior_trace=self.ior.trace,
            )  # selected: (B,)

```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
            # Update IOR trace (use first batch item -- for B=1 generation)
            self.ior.step(selected[0].item())

            # Append selected token
            tokens = torch.cat([tokens, selected.unsqueeze(1)], dim=1)

            if return_metadata:
                step_metadata.append(meta)

        if was_training:
            model.train()

        if return_metadata:
            return tokens, step_metadata
        return tokens, None

    @torch.no_grad()
    def generate_with_speculative(
        self,
        prompt_tokens: torch.Tensor,
        max_new_tokens: int = 100,
    ) -> Tuple[torch.Tensor, Dict]:
        """
        Speculative decoding using ProspectiveHPCHead (Thm 10).

        Uses the model's 8-step future predictions to draft candidate
        tokens, then verifies them against the full model. Accepted
        tokens are committed as a batch (faster generation).

        This mimics the brain's phrase-level planning: the articulatory
        system prepares entire phrases before execution, with
        self-monitoring catching errors before articulation (Levelt 1989).

        Args:
            prompt_tokens: (1, T) input token IDs (B=1 only for speculative)
            max_new_tokens: number of tokens to generate

        Returns:
            tokens: (1, T + generated) complete sequence
            stats: dict with acceptance rate, speculation depth, etc.
        """
        model = self.model
        was_training = model.training
        model.eval()
        cfg = model.cfg
        device = prompt_tokens.device

        tokens = prompt_tokens.clone()
        self.ior.reset(device)

        total_accepted = 0
        total_speculated = 0
        total_tokens_generated = 0
        spec_steps = self.cfg.speculative_steps

        while total_tokens_generated < max_new_tokens:
            ctx = tokens[:, -cfg.max_seq_len:]
            fwd_out = model.forward(ctx)

            logits = fwd_out[0]                 # (1, T, V)
            hpc_out = fwd_out[6]                # (1, T, d)
            sigma2_up_list = fwd_out[4]

            sigma2, tonic_da, conflict = self._extract_signals(sigma2_up_list)

            # Step 1: Draft candidates using ProspectiveHPC predictions
            # ProspectiveHPCHead is at model.prosp_hpc (V10+)
            # Final norm is at model.norm (RMSNorm)
            draft_tokens = []

            # Save IOR state before speculative drafting -- drafting updates
            # the trace, but rejected tokens should not contaminate it.
            ior_saved = self.ior.trace.clone() if self.ior.trace is not None else None

            if (hasattr(model, 'prosp_hpc') and hpc_out is not None
                    and hasattr(model, 'norm')):
                hpc_last = hpc_out[:, -1, :]  # (1, d)

                for k in range(min(spec_steps, 8)):
                    if k < len(model.prosp_hpc.W_prosp):
                        # ProspectiveHPC predicts future residual states
                        X_pred = model.prosp_hpc.W_prosp[k](
                            model.prosp_hpc.norm(hpc_last)
                        )  # (1, d)
                        draft_logits = model.lm_head(model.norm(X_pred))  # (1, V)

                        # Use brain decoding for draft selection too
                        selected, _ = self.sampler.sample(
                            logits=draft_logits.float(),
                            sigma2=sigma2,
                            tonic_da=tonic_da,
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
                            conflict=conflict,
                            ior_trace=self.ior.trace,
                        )
                        draft_tokens.append(selected[0].item())
                        self.ior.step(selected[0].item())

            # If no ProspectiveHPC, fallback to single-token generation
            if not draft_tokens:
                logits_last = logits[:, -1, :].float()
                selected, _ = self.sampler.sample(
                    logits=logits_last,
                    sigma2=sigma2,
                    tonic_da=tonic_da,
                    conflict=conflict,
                    ior_trace=self.ior.trace,
                )
                tokens = torch.cat([tokens, selected.unsqueeze(1)], dim=1)
                self.ior.step(selected[0].item())
                total_tokens_generated += 1
                continue

            # Restore IOR to pre-draft state -- only accepted tokens will be re-applied
            if ior_saved is not None:
                self.ior.trace = ior_saved

            # Step 2: Verify drafts against full model
            # Append all draft tokens to context
            draft_tensor = torch.tensor(
                draft_tokens, device=device
            ).unsqueeze(0)  # (1, spec_steps)
            extended = torch.cat([tokens, draft_tensor], dim=1)

            # Run full model on extended context
            ctx_verify = extended[:, -cfg.max_seq_len:]
            verify_out = model.forward(ctx_verify)
            verify_logits = verify_out[0]  # (1, T_ext, V)

            # Account for truncation offset: if extended was truncated,
            # positions shifted by the number of tokens removed from front
            offset = max(0, extended.shape[1] - cfg.max_seq_len)

            # Check each draft token against full model's predictions
            n_accepted = 0
            prompt_len = tokens.shape[1]
            for k in range(len(draft_tokens)):
                # Position in verify_logits: logits at pos predict token at pos+1
                # Draft token k sits at original position prompt_len + k
                # In verify_logits (after truncation): prompt_len + k - offset - 1
                pos = prompt_len + k - 1 - offset
                if pos < 0 or pos >= verify_logits.shape[1]:
                    break

                verify_probs = F.softmax(verify_logits[0, pos, :].float(), dim=-1)
                draft_prob = verify_probs[draft_tokens[k]].item()

                # Accept if probability is above threshold
                if draft_prob > self.cfg.speculative_threshold / cfg.V:
                    n_accepted += 1
                else:
                    break  # Reject from first mismatch onward

            total_speculated += len(draft_tokens)
            total_accepted += n_accepted

            # Re-apply IOR only for accepted tokens (not rejected drafts)
            if n_accepted > 0:
                for t in draft_tokens[:n_accepted]:
                    self.ior.step(t)
                accepted = torch.tensor(
                    draft_tokens[:n_accepted], device=device
                ).unsqueeze(0)
                tokens = torch.cat([tokens, accepted], dim=1)
                total_tokens_generated += n_accepted
            else:
                # All rejected -- fall back to brain-decoded single token
                logits_last = logits[:, -1, :].float()
                selected, _ = self.sampler.sample(
                    logits=logits_last,
                    sigma2=sigma2,
                    tonic_da=tonic_da,
                    conflict=conflict,
                    ior_trace=self.ior.trace,
                )
                tokens = torch.cat([tokens, selected.unsqueeze(1)], dim=1)
                self.ior.step(selected[0].item())
                total_tokens_generated += 1

        if was_training:
            model.train()

        stats = {
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
            'total_tokens': total_tokens_generated,
            'total_speculated': total_speculated,
            'total_accepted': total_accepted,
            'acceptance_rate': total_accepted / max(total_speculated, 1),
        }
        return tokens, stats


#

# Convenience: standard generate() for backward compatibility
#

def generate_standard(
    model: nn.Module,
    prompt_tokens: torch.Tensor,
    max_new_tokens: int = 50,
    temperature: float = 1.0,
    top_k: int = 40,
) -> list:
    """Standard autoregressive generation (temperature/top-k)."""
    was_training = model.training
    model.eval()
    tokens = prompt_tokens.clone()
    cfg = model.cfg

    with torch.no_grad():
        for _ in range(max_new_tokens):
            ctx = tokens[:, -cfg.max_seq_len:]
            logits, *_ = model.forward(ctx)
            logits_last = logits[:, -1, :] / max(temperature, 1e-6)
            if top_k > 0:
                topk_vals, _ = logits_last.topk(
                    min(top_k, logits_last.size(-1)), dim=-1
                )
                logits_last[logits_last < topk_vals[:, -1:]] = float('-inf')
            probs = F.softmax(logits_last.float(), dim=-1)
            next_tok = torch.multinomial(probs, 1)
            tokens = torch.cat([tokens, next_tok], dim=1)

    if was_training:
        model.train()
    return tokens[0].tolist()


#

# Analysis utilities
#

def summarize_generation_metadata(metadata_list: List[Dict]) -> Dict:
    """
    Summarize per-step generation metadata into aggregate statistics.

    Returns:
        dict with:
          - system1_frac: fraction of tokens decided by System 1
          - mean_deliberation: average deliberation steps (System 2 only)
          - mean_confidence: average confidence across all tokens
          - mean_theta: average effective threshold
          - mean_sigma_noise: average noise level
          - da_range: (min, max) tonic DA during generation
          - conflict_range: (min, max) ACC conflict during generation
    """
    if not metadata_list:
        return {}

    system1_fracs = [m['system1_frac'] for m in metadata_list]
    delib_steps = torch.cat([m['deliberation_steps'] for m in metadata_list])
    confidences = torch.cat([m['confidence'] for m in metadata_list])
    thetas = [m['theta_0'] for m in metadata_list]
    sigmas = [m['sigma_noise'] for m in metadata_list]
    das = [m['tonic_da'] for m in metadata_list]
    conflicts = [m['conflict'] for m in metadata_list]

    # System 2 deliberation steps only
    s2_steps = delib_steps[delib_steps > 0]

    return {
        'system1_frac': sum(system1_fracs) / len(system1_fracs),
        'mean_deliberation': s2_steps.mean().item() if len(s2_steps) > 0 else 0.0,
        'max_deliberation': s2_steps.max().item() if len(s2_steps) > 0 else 0,
        'mean_confidence': confidences.mean().item(),
        'mean_theta': sum(thetas) / len(thetas),
        'mean_sigma_noise': sum(sigmas) / len(sigmas),
        'da_range': (min(das), max(das)),
        'conflict_range': (min(conflicts), max(conflicts)),
        'n_tokens': len(metadata_list),
    }
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```python


def compare_decoders(
    model: nn.Module,
    prompt_tokens: torch.Tensor,
    tokenizer,
    max_new_tokens: int = 50,
    brain_config: BrainDecodingConfig = None,
    temperatures: List[float] = None,
) -> Dict:
    """
    Compare brain-inspired decoding against standard temperature sampling.

    Generates text with both methods and reports:
      - Generated text
      - Repetition rate (n-gram overlap)
      - Distinct-n scores (diversity)
      - Per-token statistics (for brain decoder)

    Args:
        model: NCT model
        prompt_tokens: (1, T) prompt
        tokenizer: for decoding tokens to text
        max_new_tokens: tokens to generate
        brain_config: BrainDecodingConfig
        temperatures: list of temperatures for standard sampling

    Returns:
        dict with results for each method
    """
    if temperatures is None:
        temperatures = [0.7, 1.0, 1.3]

    results = {}
    device = prompt_tokens.device

    #        Brain-inspired decoding

    decoder = BrainDecoder(model, config=brain_config)
    brain_tokens, brain_meta = decoder.generate(
        prompt_tokens, max_new_tokens=max_new_tokens, return_metadata=True
    )
    brain_text = tokenizer.decode(brain_tokens[0].tolist())
    brain_summary = summarize_generation_metadata(brain_meta)

    results['brain'] = {
        'text': brain_text,
        'tokens': brain_tokens[0].tolist(),
        'metadata': brain_summary,
        'repetition_rate': _repetition_rate(brain_tokens[0].tolist(), n=3),
        'distinct_1': _distinct_n(brain_tokens[0].tolist(), n=1),
        'distinct_2': _distinct_n(brain_tokens[0].tolist(), n=2),
    }

    #        Standard temperature sampling

    for temp in temperatures:
        std_tokens = generate_standard(
            model, prompt_tokens,
            max_new_tokens=max_new_tokens,
            temperature=temp, top_k=40,
        )
        std_text = tokenizer.decode(std_tokens)

        results[f'temp_{temp}'] = {
            'text': std_text,
            'tokens': std_tokens,
            'repetition_rate': _repetition_rate(std_tokens, n=3),
            'distinct_1': _distinct_n(std_tokens, n=1),
            'distinct_2': _distinct_n(std_tokens, n=2),
        }

    return results


def _repetition_rate(tokens: list, n: int = 3) -> float:
    """Fraction of n-grams that are repeated at least once."""
    if len(tokens) < n:
        return 0.0
    ngrams = [tuple(tokens[i:i+n]) for i in range(len(tokens) - n + 1)]
    unique = set(ngrams)
    return 1.0 - len(unique) / len(ngrams) if ngrams else 0.0


def _distinct_n(tokens: list, n: int = 1) -> float:
    """Distinct-n metric: unique n-grams / total n-grams."""
    if len(tokens) < n:
        return 0.0
    ngrams = [tuple(tokens[i:i+n]) for i in range(len(tokens) - n + 1)]
    return len(set(ngrams)) / len(ngrams) if ngrams else 0.0
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

Listing 8: Brain Decoding — Drift-Diffusion (brain_decoding.py, 914 lines)

10 Cortical Column — The Fundamental Compute Unit

Source: `nct_v2/v11_column.py` **Lines:** 1,435 **Classes:** 11

10.1 Neuroscience Foundation

The cortical column is the fundamental repeating computational unit of the mammalian neocortex, first identified by Vernon Mountcastle in his landmark 1957 study of the somatosensory cortex and later formalized in his 1997 review. Each column is a vertical cylinder spanning all six cortical layers, approximately 300–500μm in diameter, containing approximately 80–120 neurons organized in a stereotyped circuit with distinct layer functions. The same basic circuit is replicated across the entire cortical surface — an estimated 150,000 columns in the human neocortex — from primary visual cortex (V1) to prefrontal cortex, differing primarily in the density and connectivity of its components rather than its fundamental architecture.

Douglas and Martin (2004) described the **canonical cortical circuit**, identifying the key information flow: Layer 4 receives thalamic input (the "sensory" layer), Layer 2/3 performs lateral integration and sends projections to other cortical areas (the "association" layer), Layer 5 generates output to subcortical structures (the "motor" layer), and Layer 6 provides feedback to the thalamus (the "modulatory" layer). Harris and Shepherd (2015) updated this model with modern understanding of cell-type diversity, showing that each layer contains multiple distinct cell types (excitatory pyramidal cells, inhibitory interneurons of several subtypes) with specific connectivity patterns.

NCT implements this circuit as the `CorticalColumn` class, which replaces a single transformer block. Unlike a transformer block (which contains only self-attention + FFN + LayerNorm = 3 components), a cortical column contains six distinct computational layers, three interneuron types (VIP, SST, PV/Chandelier), three-timescale working memory (NMDA, silent trace, phonological loop), oscillatory dynamics with nested PAC, and homeostatic regulation (BCM + metaplasticity) — 11 submodules totaling ∼7.5M parameters per column in the default configuration. Each submodule is gated and identity-at-init, and the full column data flow mirrors the biological circuit described by Douglas, Martin, Harris, and Shepherd.

10.2 Data Flow

The biological data flow through a cortical column, with biologically correct placement of interneurons:

$$\underbrace{\text{L4}}_{\text{input}} \rightarrow \underbrace{\text{L1}}_{\text{feedback}} \rightarrow \underbrace{\text{SST+VIP}}_{\text{pre-L2/3}} \rightarrow \underbrace{\text{L2/3}}_{\text{association}} \rightarrow \underbrace{\text{WM}}_{\text{persistence}} \rightarrow \underbrace{\text{Osc}}_{\text{timing}} \rightarrow \underbrace{\text{PV}}_{\text{post-comp}} \rightarrow \underbrace{\text{L5}}_{\text{output}} \rightarrow \underbrace{\text{L6}}_{\text{modulation}} \tag{28}$$

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

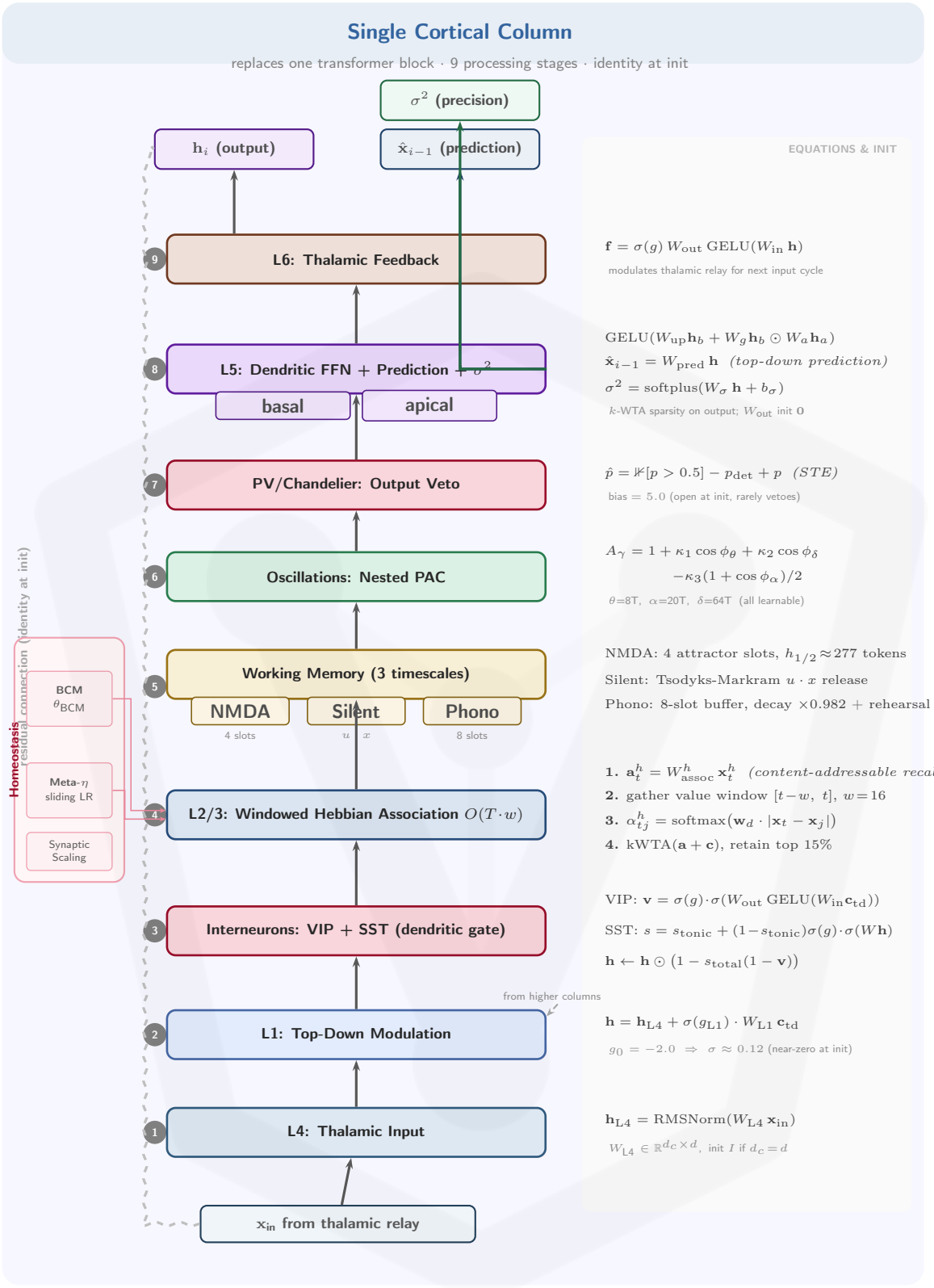


Figure 17: Complete data flow through a single cortical column. All nine processing stages are shown with their equations and initialization values. Stage numbers (1) indicate the biological processing order. The residual connection (left, dashed) bypasses all computation, ensuring identity-at-init when all gates $\sigma(g_0 = -2) \approx 0.12$. BCM homeostasis, metaplasticity, and synaptic scaling (left panel) regulate Hebbian learning in L2/3. L5 produces three outputs: the column representation $\mathbf{h}_i$, a top-down prediction $\hat{\mathbf{x}}_{i-1}$ for predictive coding, and precision σ^2 for uncertainty-aware processing.

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

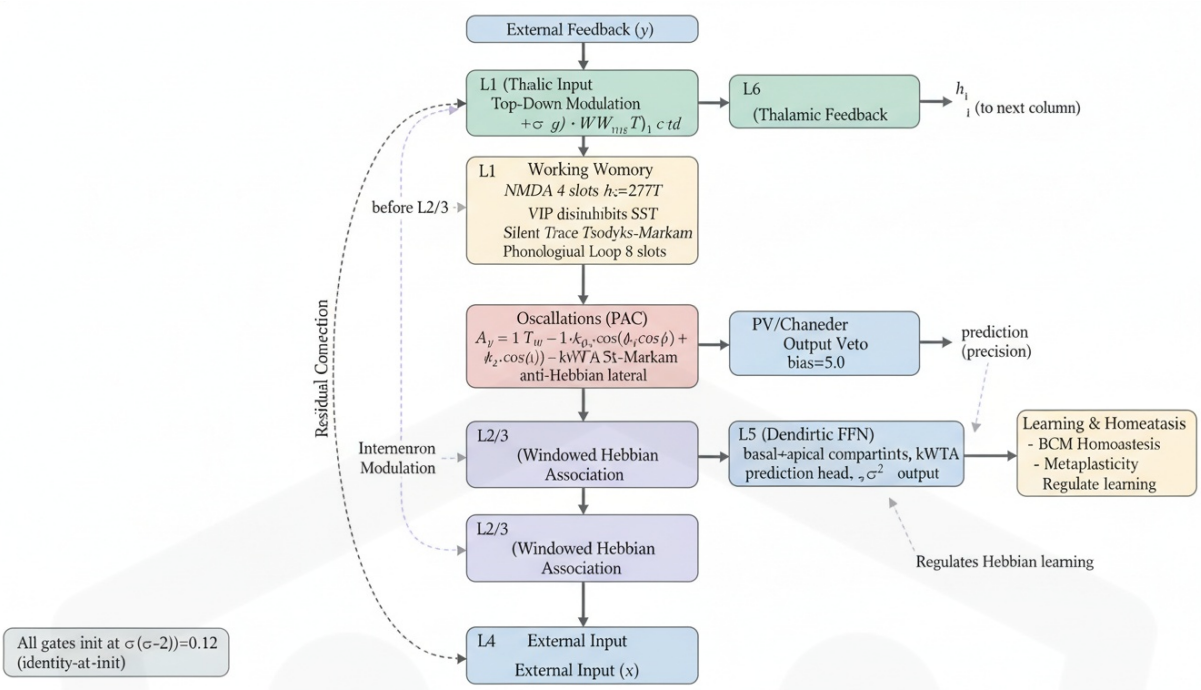


Figure 18: (Illustrated) Complete data flow through a single cortical column. All nine processing stages are shown with their equations and initialization values. The residual connection (dashed) bypasses all computation, ...

10.3 Layer 4 — Thalamic Input

Definition 10.1 (Layer 4)**.** *The thalamic input layer projects feedforward input from inter-column dimension to intra-column dimension:*

$$\mathbf{h}_{L4} = RMSNorm(W_{L4}\mathbf{x}_{in}), \qquad W_{L4} \in \mathbb{R}^{d_{col} \times d_{model}} \tag{29}$$

Initialization: If $d_{\text{col}} = d_{\text{model}}$ (default), $W_{\text{L4}} = I$ (identity matrix — pure pass-through). Otherwise, Kaiming normal initialization. This corresponds to the stellate cells in biological L4 that relay thalamic input with minimal transformation.

10.4 Layer 2/3 — Windowed Hebbian Association (Theorem B)

This is the core computational layer, replacing self-attention entirely. It implements Theorem B: Hebbian + anti-Hebbian learning converges to the PCA subspace, providing self-organizing representations without any supervised signal.

Theorem 10.1 (Windowed Hebbian Association Replaces Self-Attention)**.** *Let $\mathbf{X} \in \mathbb{R}^{T \times d}$ be a sequence of token representations. The L2/3 windowed Hebbian association computes, for each position t and head h:*

$$\mathbf{y}_t^h = W_{assoc}^h \mathbf{x}_t^h + \sum_{j=t-w}^{t} \frac{\exp(\mathbf{w}_d^h \cdot |\mathbf{x}_t^h - \mathbf{x}_j^h|)}{\sum_{k=t-w}^{t} \exp(\mathbf{w}_d^h \cdot |\mathbf{x}_t^h - \mathbf{x}_k^h|)} \cdot W_{val}\mathbf{x}_j^h \tag{30}$$

with complexity $O(T \cdot w \cdot d)$ compared to $O(T^2 \cdot d)$ for self-attention. For $T = 4096$ and $w = 16$, this is $256\times$ cheaper.

Complexity proof. Self-attention computes $QK^\top$ over all T^2 pairs. L2/3 computes associations only within a causal window of size $w + 1$. Per position: $w + 1$ comparisons $\times d_h$ dimensions $\times H$ heads $= O(w \cdot d)$. Over T positions: $O(T \cdot w \cdot d)$. □

The five steps of the L2/3 mechanism:

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

Step 1 — Content-addressable recall: $\mathbf{a}_t^h = W_{\text{assoc}}^h \mathbf{x}_t^h$ where $W_{\text{assoc}}^h \in \mathbb{R}^{d_h \times d_h}$ is initialized as identity (self-recall) and updated by Hebbian learning. This answers: "what does this pattern remind the column of?"

Step 2 — Local window context: Gather $\mathbf{v}_j^h = W_{\text{val}} \mathbf{x}_j^h$ for $j \in [t-w, t]$ (causal, zero-padded). Window size $w = 16$ tokens = 2 theta cycles.

Step 3 — Distance-weighted context: $\alpha_{tj}^h = \text{softmax}(\mathbf{w}_d^h \cdot |\mathbf{x}_t^h - \mathbf{x}_j^h|)$ where $\mathbf{w}_d^h \in \mathbb{R}^{d_h}$ initialized at $\mathbf{0}$ (uniform weighting at init).

Step 4 — k-WTA + anti-Hebbian: Retain only top 15% of activations (sparse coding). Anti-Hebbian lateral weights W_{lat}^h (init $\mathbf{0}$) enforce decorrelation.

Step 5 — Output: W_{out} (init $\mathbf{0}$) projects combined output. Gate $\sigma(g_{\text{L2/3}})$ with $g_0 = -2.0$.

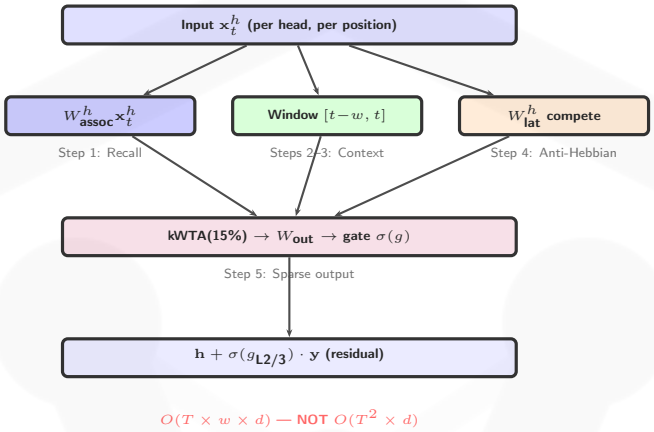


Figure 19: L2/3 Windowed Hebbian Association — 5-step mechanism replacing self-attention.

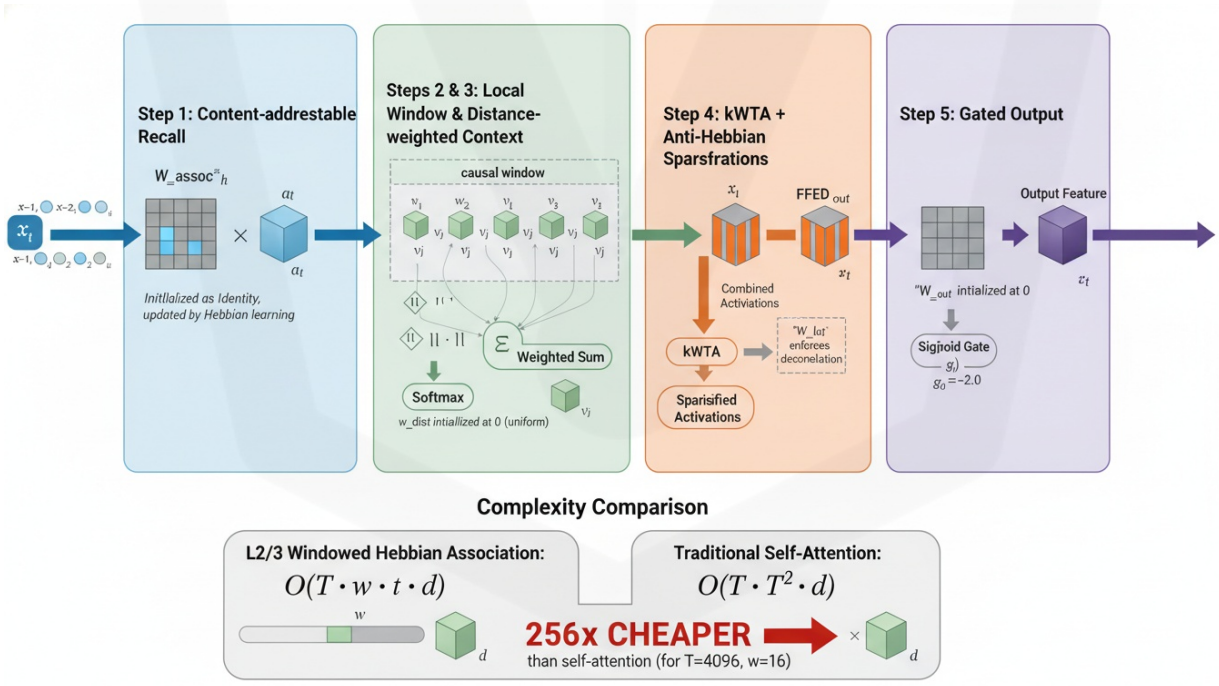


Figure 20: (Illustrated) L2/3 Windowed Hebbian Association — 5-step mechanism replacing self-attention.

Proposition 10.1 (Hebbian Update Rule)**.** *The association weights are updated by:*

$$\Delta W_{assoc}^h = \eta_H \cdot \mathbf{y}^h(\mathbf{y}^h - \theta_{BCM}) \cdot (\mathbf{x}^h)^\top \quad \textit{(Hebbian + BCM)} \tag{31}$$

$$\Delta W_{lat}^h = -\eta_H \cdot \mathbf{y}^h(\mathbf{y}^h)^\top \quad \textit{(anti-Hebbian)} \tag{32}$$

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

This combination converges to the PCA subspace (Pehlevan & Chklovskii, 2019), self-organizing representations without any supervised signal.

L2/3 also implements **structural plasticity** (Theorem E): synaptic density tracking, growth where PE is persistently high ($\|\epsilon\| > 2.0$), and pruning where gates approach zero ($\sigma(g) < 0.01$), checked every 1,000 steps.

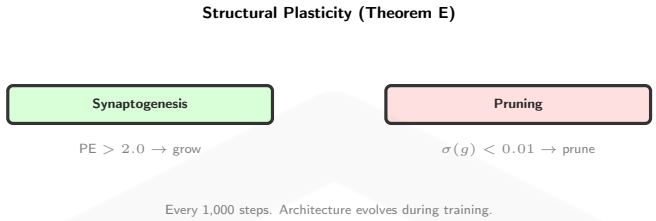


Figure 21: Structural plasticity: grow connections where PE is high, prune where gates are near zero.

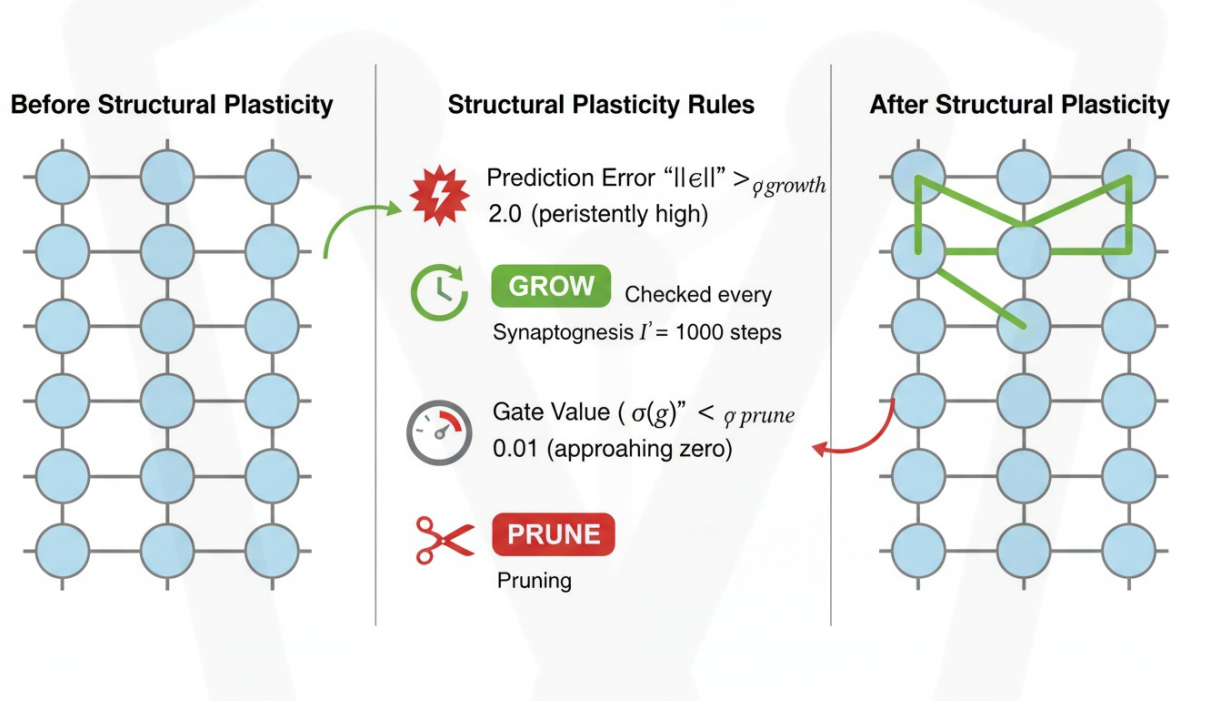


Figure 22: (Illustrated) Structural plasticity: grow connections where PE is high, prune where gates are near zero.

10.5 Working Memory — Three Timescales (NMDA, Silent, Phonological)

Theorem 10.2 (Multi-Timescale Working Memory)**.** *NCT implements three biologically-grounded WM systems operating at different temporal scales:*

1. NMDA Attractor Slots *(Luck & Vogel, 1997): $K = 4$ competitive slots with learned exponential decay:*

$$slots_{t+1} = \underbrace{\sigma(\beta_k)}_{decay} \cdot slots_t + \underbrace{softmax(W_s \mathbf{h}_t)}_{write\ weight} \cdot \underbrace{W_w \mathbf{h}_t}_{content} \tag{33}$$

Decay bias $\beta_{k,init} = 6.0 \Rightarrow \sigma(6) \approx 0.9975$, giving half-life ≈ 277 tokens.

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

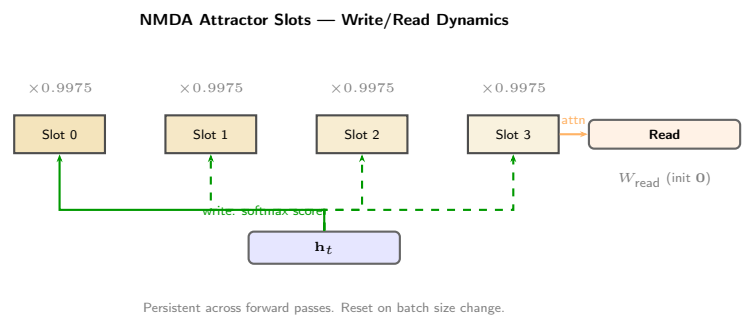


Figure 23: NMDA attractor slot dynamics: 4 competitive slots with exponential decay ($\alpha \approx 0.9975$, half-life ≈ 277 tokens). Write via softmax competition over slots. Read via query-based attention. State persists across sequential forward passes.

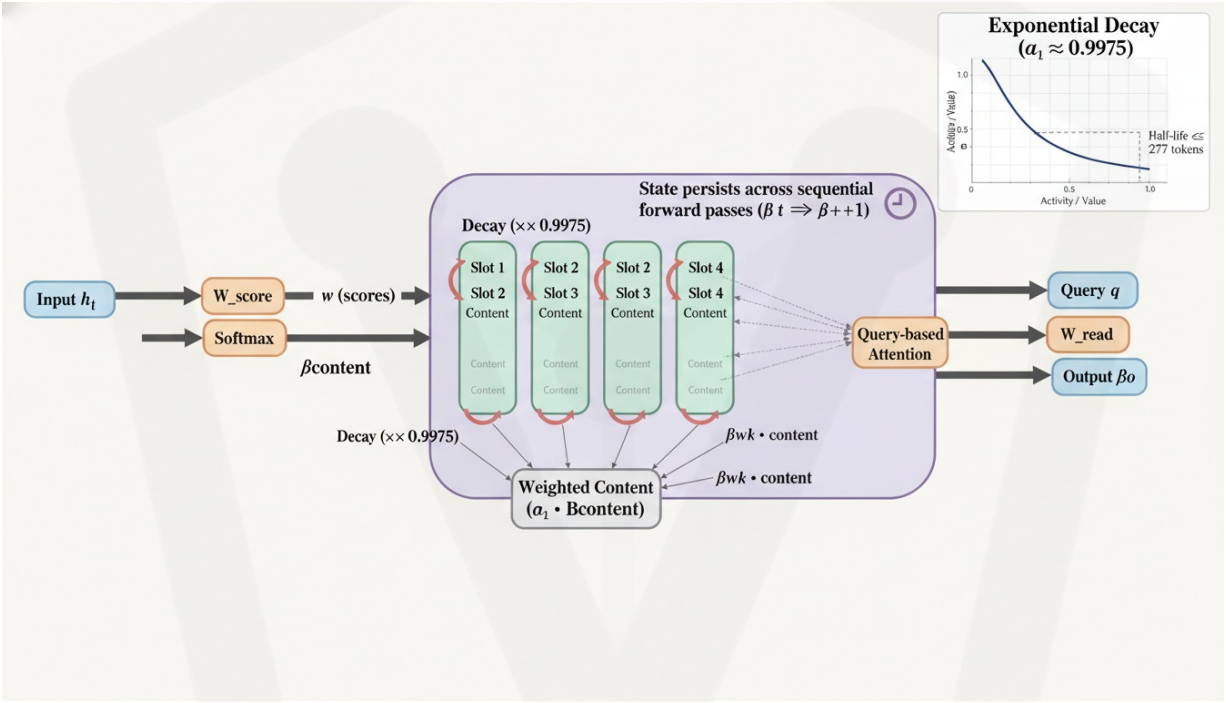


Figure 24: (Illustrated) NMDA attractor slot dynamics: 4 competitive slots with exponential decay ($\alpha \approx 0.9975$, half-life ≈ 277 tokens). Write via softmax competition over slots. Read via query-based at...

2. ***Activity-Silent Trace*** *(Mongillo et al., 2008): Tsodyks-Markram short-term plasticity dynamics:*

$$release_t = u_t \cdot x_t \tag{34}$$

$$u_{t+1} = \alpha_f u_t + (1 - u_t) \cdot w_g \cdot |content_t| \qquad (facilitation) \tag{35}$$

$$x_{t+1} = \alpha_d x_t + (1 - \alpha_d)(1 - x_t) - \delta_f u_t x_t \qquad (depression) \tag{36}$$

where $\alpha_f = \sigma(4.6) \approx 0.990$, $\alpha_d = \sigma(2.5) \approx 0.924$.

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

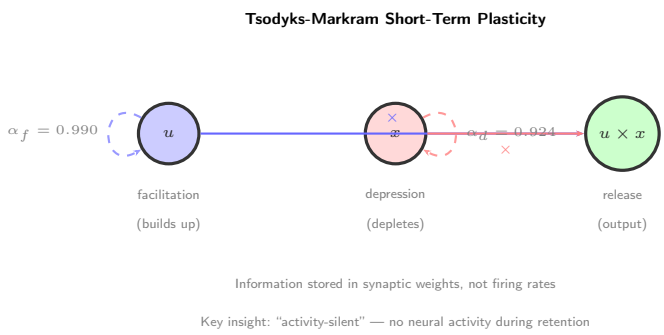


Figure 25: Tsodyks-Markram activity-silent trace: facilitation variable u builds up with input (slow decay $\alpha_f = 0.990$), depression variable x depletes (faster recovery $\alpha_d = 0.924$). The effective release $u \times x$ captures the interaction. Information is stored in synaptic weights rather than sustained firing — hence "activity-silent."

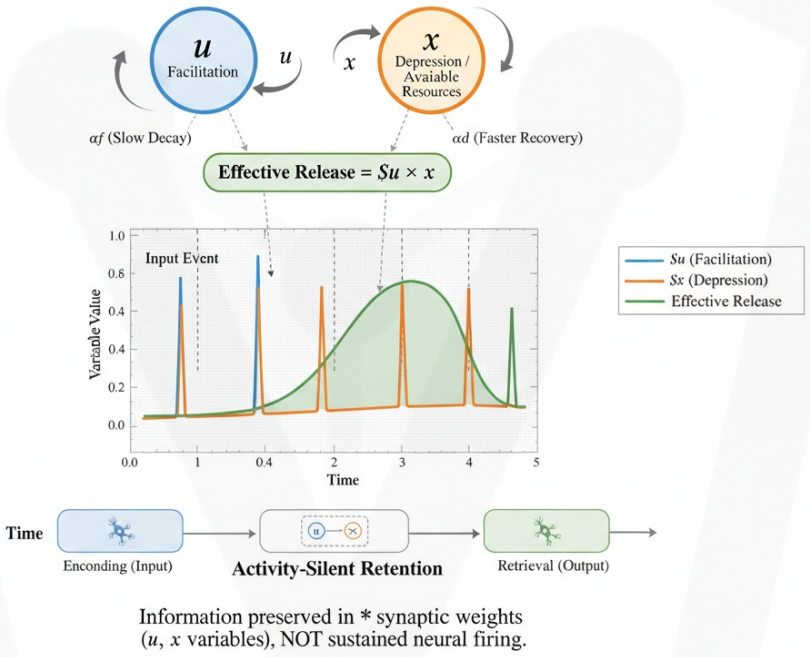


Figure 26: (Illustrated) Tsodyks-Markram activity-silent trace: facilitation variable u builds up with input (slow decay $\alpha_f = 0.990$), depression variable x depletes (faster recovery $\alpha_d = 0.924$). The eff...

3. ***Phonological Loop**: Circular buffer with passive decay and rehearsal:*

$$slots_{t+1} = p_{decay} \cdot slots_t + write(\mathbf{h}_t, ptr), \qquad p_{decay} = \sigma(4.0) \approx 0.982 \tag{37}$$

$K_p = 8$ slots (Miller's 7 ± 2), rehearsal strength $r = \sigma(-1.0) \approx 0.27$.

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

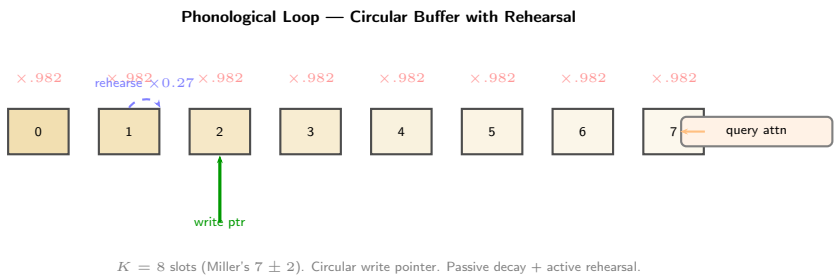


Figure 27: Phonological loop: 8-slot circular buffer with passive exponential decay (×0.982 per step) and active rehearsal of the most recent slot (×0.27). Write pointer advances modulo 8. Read via query-based attention over all slots. Models the brain's verbal working memory ("inner speech").

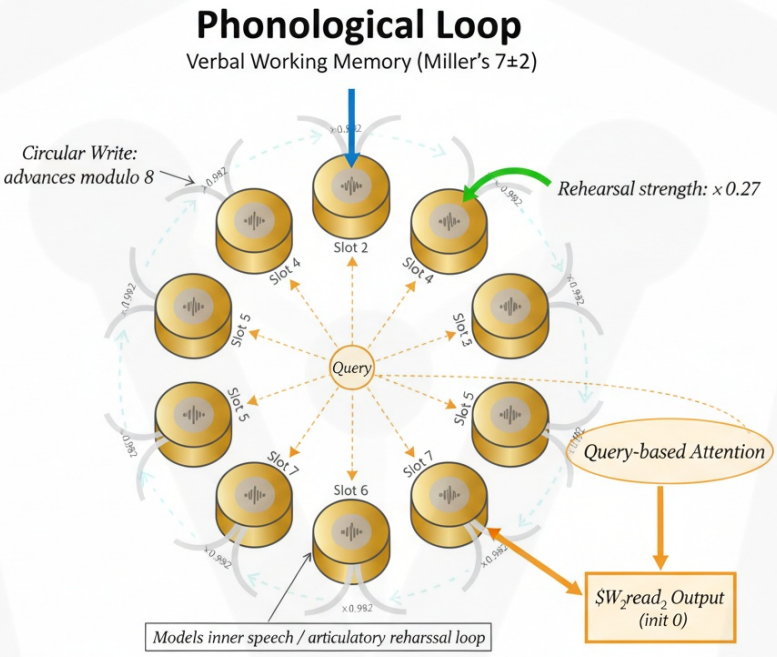


Figure 28: (Illustrated) Phonological loop: 8-slot circular buffer with passive exponential decay (×0.982 per step) and active rehearsal of the most recent slot (×0.27). Write pointer advances modulo 8. Rea...

All three subsystems are **gated** (gate init −2.0), **identity-at-init** ($W_{\text{read}} = \mathbf{0}$), and maintain **persistent state** across forward passes (unlike transformer layers, which are stateless).

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

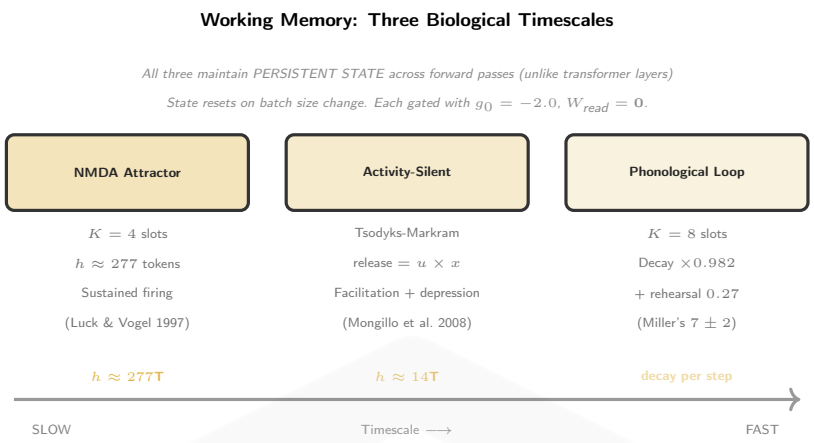


Figure 29: Working memory hierarchy: three biological timescales. NMDA attractor slots provide slow, sustained storage (half-life ≈277 tokens). Activity-silent traces use synaptic facilitation/depression for medium-term storage. The phonological loop provides fast, capacity-limited rehearsal (8 slots, matching Miller's 7 ± 2). All three are persistent, gated, and identity-at-init.

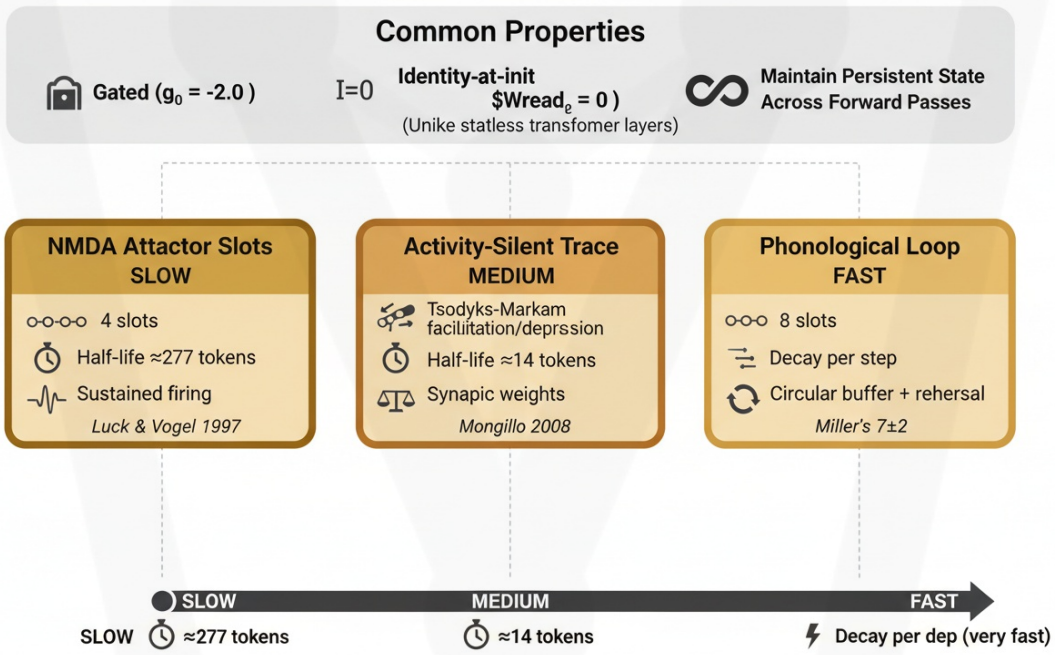


Figure 30: (Illustrated) Working memory hierarchy: three biological timescales. NMDA attractor slots provide slow, sustained storage (half-life ≈277 tokens). Activity-silent traces use synaptic facilitation/depress...

10.6 Oscillatory Dynamics — Nested PAC

Theorem 10.3 (Phase-Amplitude Coupling)**.** *Cross-frequency coupling modulates column activity at four nested timescales:*

$$A_\gamma(t) = 1 + \underbrace{\kappa_1 \cos\phi_\theta(t)}_{\textit{theta-gamma}} + \underbrace{\kappa_2 \cos\phi_\delta(t)}_{\textit{delta-gamma}} - \underbrace{\kappa_3 \frac{1+\cos\phi_\alpha(t)}{2}}_{\textit{alpha suppression}} \tag{38}$$

where $\phi_b(t) = 2\pi t / softplus(P_b^{raw})$ *with learnable periods:* $P_\theta = 8$ *tokens,* $P_\alpha = 20$*,* $P_\delta = 64$*.*

Coupling strengths: $\kappa_1 = \tanh(0.3) \approx 0.29$ *at init (theta-gamma PAC present at rest, Jensen & Colgin 2007),* $\kappa_2 = \kappa_3 = 0$ *at init.*

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

Phase-content interaction through rank-32 projection:

$$\text{mod} = W_{\text{out}} \tanh(W_{\text{content}}\mathbf{h} + W_{\text{phase}}[\sin\phi_\theta, \cos\phi_\theta, \sin\phi_\alpha, \cos\phi_\alpha, \sin\phi_\delta, \cos\phi_\delta]) \tag{39}$$

Output: $\mathbf{h} \leftarrow \mathbf{h} + \sigma(g_{\text{osc}}) \cdot A_\gamma \cdot \text{mod}$. W_{out} initialized $\mathcal{N}(0, 0.01)$ — small but nonzero so oscillations provide signal from the start.

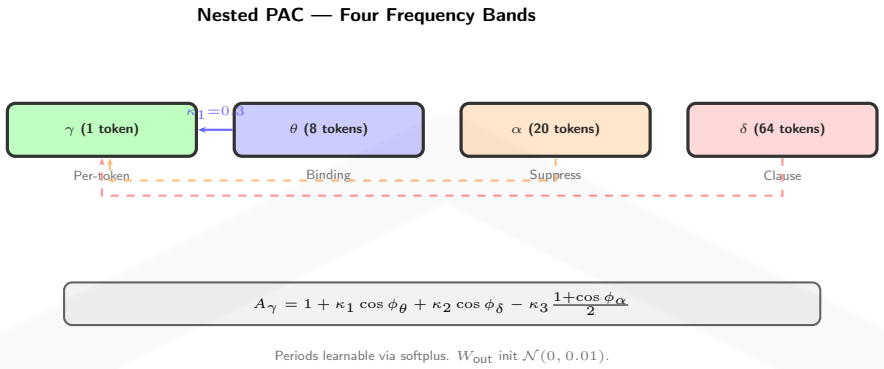


Figure 31: Nested PAC: gamma amplitude modulated by theta phase ($\kappa_1 = 0.3$), with optional delta and alpha coupling. All periods learnable.

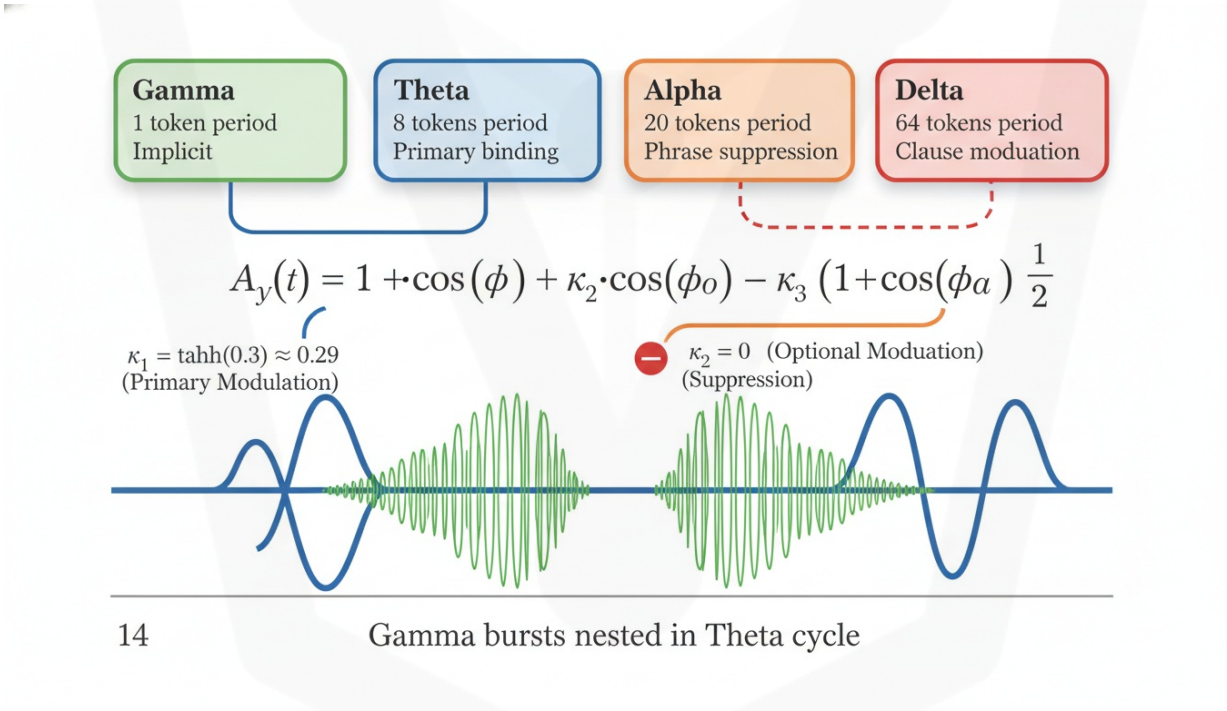


Figure 32: (Illustrated) Nested PAC: gamma amplitude modulated by theta phase ($\kappa_1 = 0.3$), with optional delta and alpha coupling. All periods learnable.

10.7 Interneurons — VIP, SST, PV/Chandelier

Three interneuron types implement distinct inhibitory functions at biologically correct positions:

Table 7: Interneuron types, placement, and function.

Type	Placement	Function	Disease if Missing
VIP	Before L2/3	Disinhibits SST (novelty gate)	Autism (ASD)
SST	Before L2/3	Dendritic inhibition ($s_{\text{tonic}} = 0.10$)	–
PV/Chandelier	After WM, before L5	Axo-axonic output veto	Epilepsy

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

VIP disinhibition: When top-down signals indicate novelty, VIP neurons *inhibit* SST neurons, which *disinhibits* the pyramidal cells — a double negative that opens the gate for new information. This is the brain's mechanism for attention: "this is new and important, let it through."

PV/Chandelier veto: A straight-through hard gate with bias initialized at 5.0 ($\sigma(5) \approx 0.993$, nearly always open). The Chandelier cell can completely silence a pyramidal cell's output by vetoing at the axon initial segment. This prevents erroneous outputs when the column is uncertain.

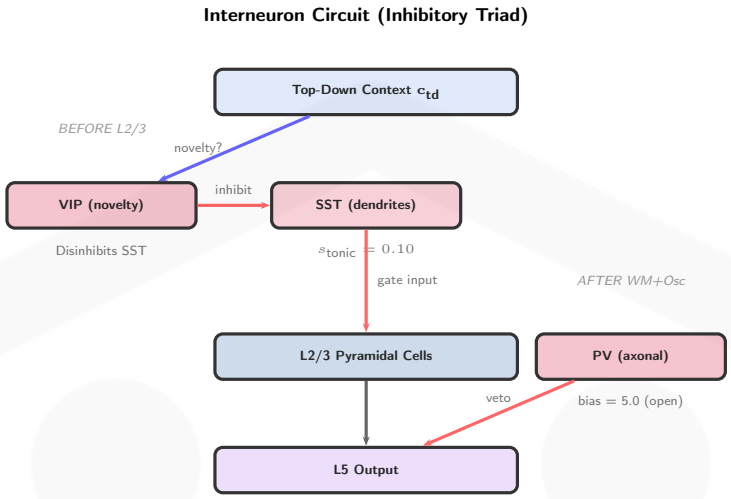


Figure 33: Interneuron circuit: VIP detects top-down novelty and disinhibits SST (double negative = opens gate). SST provides tonic + gated dendritic inhibition before L2/3. PV/Chandelier vetoes output after working memory and oscillations, before L5. Placement is biologically correct.

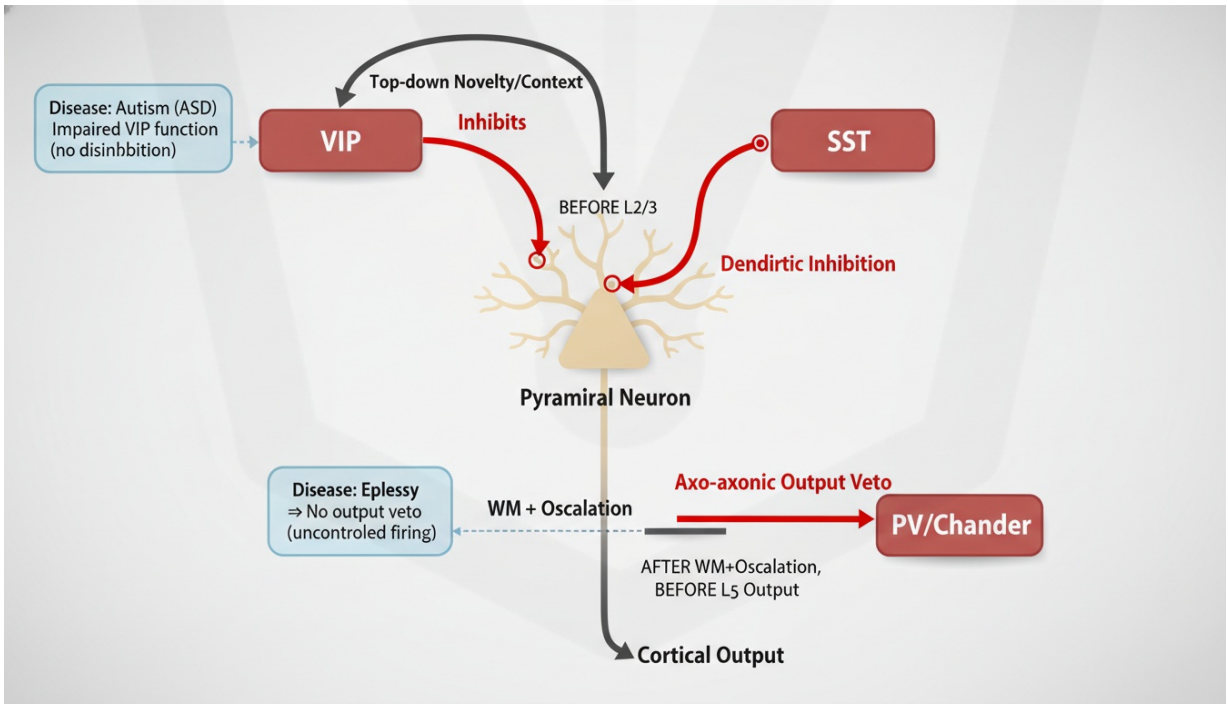


Figure 34: (Illustrated) Interneuron circuit: VIP detects top-down novelty and disinhibits SST (double negative = opens gate). SST provides tonic + gated dendritic inhibition before L2/3. PV/Chandelier vetoes output after ...

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

10.8 BCM Homeostasis and Metaplasticity

Theorem 10.4 (BCM Sliding Threshold (Bienenstock, Cooper, Munro 1982)). *The BCM threshold θ_{BCM} tracks mean squared postsynaptic activity:*

$$\theta_{BCM} \leftarrow \left(1 - \frac{1}{\tau_{BCM}}\right)\theta_{BCM} + \frac{1}{\tau_{BCM}}\overline{y^2}, \qquad \tau_{BCM} = 1000 \tag{40}$$

The Hebbian update is modulated: when $y > \theta_{BCM}$, potentiation occurs (strengthen active synapses). When $y < \theta_{BCM}$, depression occurs (weaken inactive synapses). This prevents runaway Hebbian learning — without BCM, strong synapses would grow without bound.

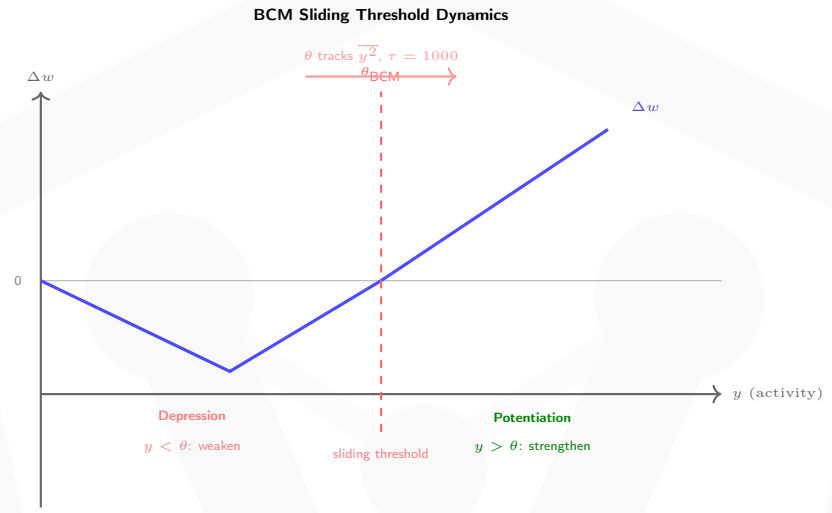


Figure 35: BCM sliding threshold: the threshold θ_{BCM} divides activity into depression (below) and potentiation (above) zones. The threshold itself slides to track mean squared activity ($\tau = 1000$), ensuring homeostatic balance. Without BCM, Hebbian learning would cause runaway potentiation.

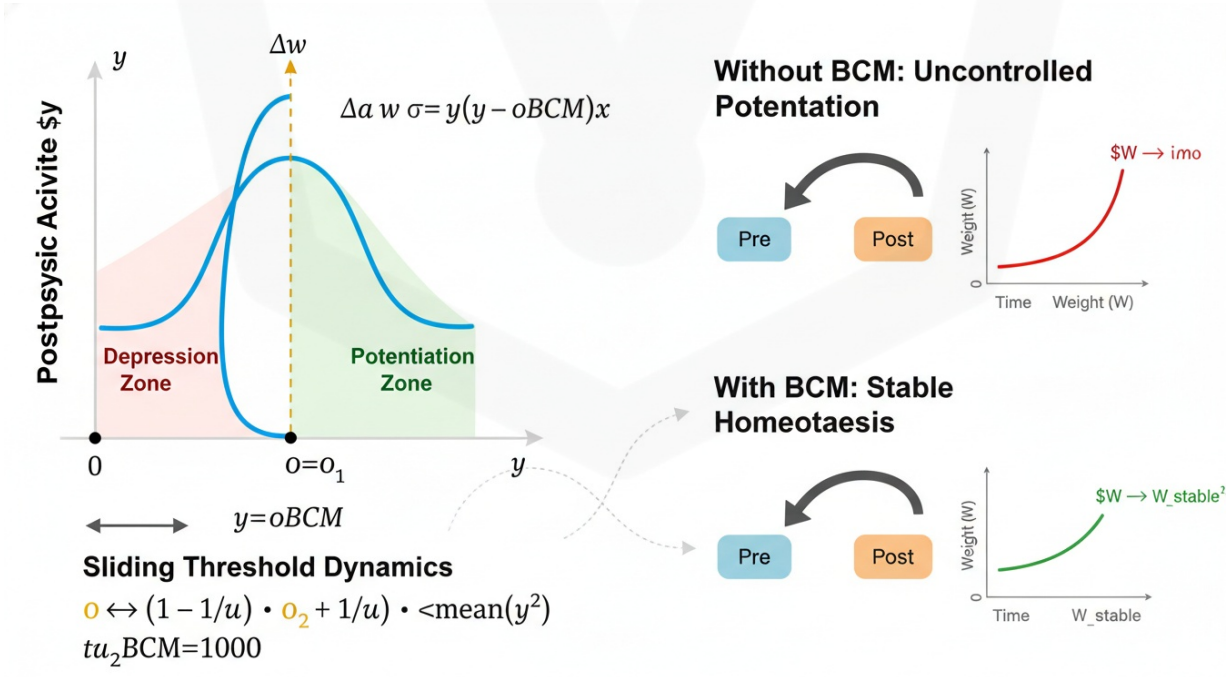


Figure 36: (Illustrated) BCM sliding threshold: the threshold θ_{BCM} divides activity into depression (below) and potentiation (above) zones. The threshold itself slides to track mean squared activity (τ...

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

Proposition 10.2 (Metaplasticity (Abraham & Bear, 1996))**.** *The effective learning rate adapts based on recent plasticity history:*

$$\eta_{eff} = \frac{\eta}{1 + plasticity_ema}, \qquad plasticity_ema \leftarrow \left(1 - \frac{1}{500}\right) plasticity_ema + \frac{1}{500}|\Delta W| \tag{41}$$

High recent plasticity reduces the learning rate (stabilization). Low recent plasticity increases it (exploration). This is the brain's mechanism for "learning to learn differently" — the plasticity of plasticity.

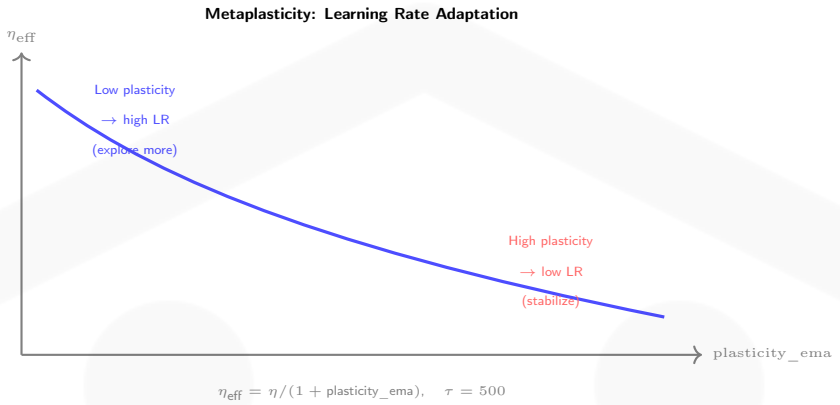


Figure 37: Metaplasticity: the effective learning rate is inversely proportional to recent plasticity. High recent weight changes → reduce LR (stabilize). Low recent changes → increase LR (explore). This prevents oscillation between over- and under-learning.

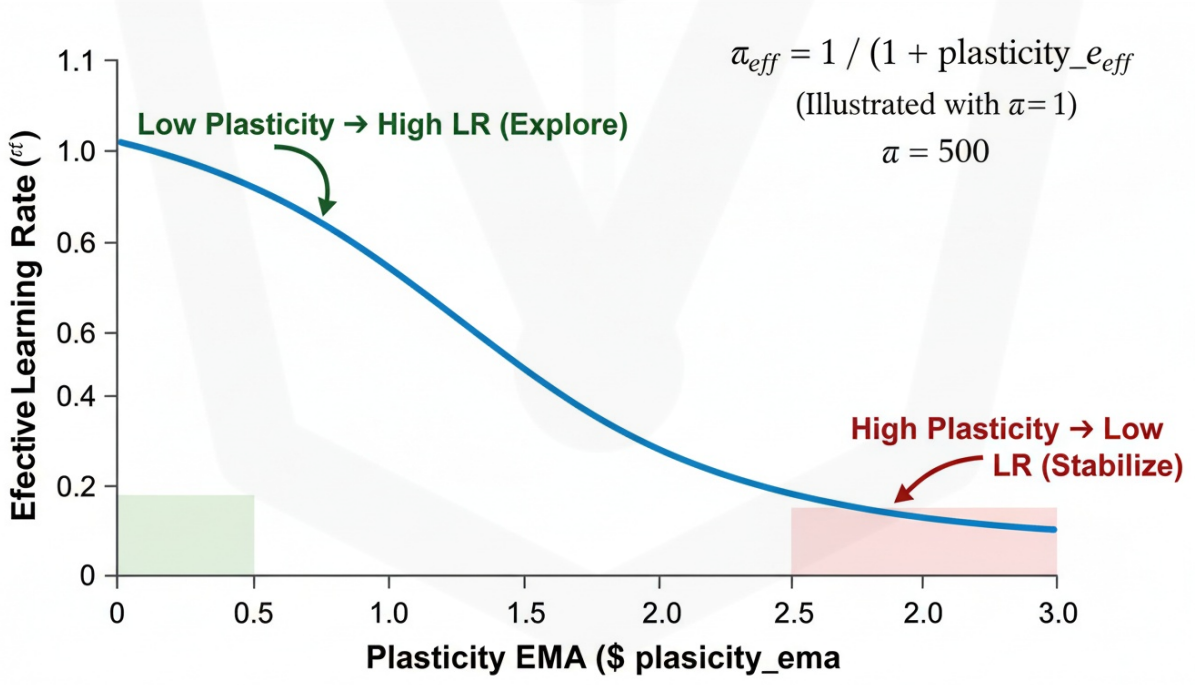


Figure 38: (Illustrated) Metaplasticity: the effective learning rate is inversely proportional to recent plasticity. High recent weight changes → reduce LR (stabilize). Low recent changes → increase...

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

10.9 Layer 5 — Dendritic FFN + Prediction + Precision

Theorem 10.5 (Dendritic Two-Compartment Computation)**.** *L5 implements gated computation with separate basal (feedforward) and apical (feedback) compartments:*

$$\mathbf{h}_{basal} = RMSNorm(\mathbf{h}) \tag{42}$$

$$\mathbf{h}_{apical} = \mathbf{h}_{L4} + \mathbf{c}_{td} \tag{43}$$

$$\mathbf{f} = GELU(W_{up}\mathbf{h}_b + W_{gate}\mathbf{h}_b \odot W_{apical}\mathbf{h}_a) \tag{44}$$

$$\mathbf{y}_{L5} = W_{out} \cdot kWTA(\mathbf{f}) \tag{45}$$

The gated linear unit (GLU) architecture means that the apical compartment (top-down feedback) can *gate* the basal compartment (bottom-up input). This is biologically faithful: calcium spikes in apical dendrites can gate somatic output, controlling whether a neuron responds to its feedforward input based on contextual feedback.

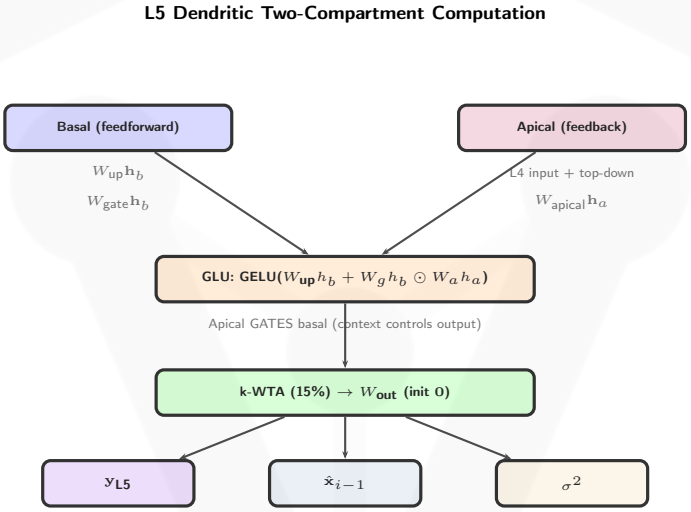


Figure 39: L5 dendritic two-compartment computation. The basal compartment (feedforward) is gated by the apical compartment (top-down feedback) via a GLU mechanism. This means top-down context can control whether a neuron responds to its feedforward input — matching calcium spike gating in biological apical dendrites. Output passes through k-WTA sparsity (15% active) before producing the column output, top-down prediction, and precision.

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

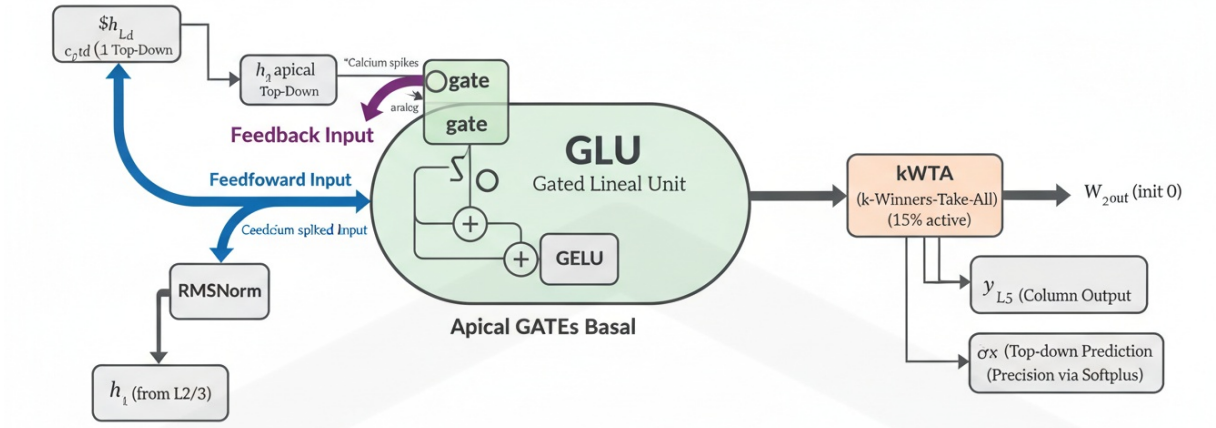


Figure 40: (Illustrated) L5 dendritic two-compartment computation. The basal compartment (feedforward) is gated by the apical compartment (top-down feedback) via a GLU mechanism. This means top-down context can control whe...

Prediction head: $\hat{\mathbf{x}}_{i-1} = W_{\text{pred}}\text{RMSNorm}(\mathbf{h})$ generates the top-down prediction for the column below.

Precision output: $\sigma^2 = \text{softplus}(W_\sigma \text{RMSNorm}(\mathbf{h}) + b_\sigma)$ with $b_{\sigma,\text{init}} = \log(e^{1.0} - 1)$ so that $\text{softplus}(b) = 1.0$.

10.10 Lateral Column Connections

Horizontal connections between adjacent columns ($i \pm 1$) via low-rank send/receive projections:

$$\mathbf{l}_i = \sigma(g_i) \cdot W_{\text{recv},i} \cdot \frac{1}{|\mathcal{N}(i)|} \sum_{j \in \mathcal{N}(i)} W_{\text{send},j}\mathbf{h}_j, \qquad r = \max(d/8, 32) \tag{46}$$

Disease mapping: schizophrenia (disconnection syndrome) → disrupted lateral connections → fragmented perception and thought disorder. W_{send} initialized at $\mathbf{0}$ (no lateral communication at init).

10.11 Column Output

The column returns a `ColumnOutput` dataclass containing seven fields, each serving a specific role in the NCT learning and communication framework:

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

Table 8: ColumnOutput fields and their roles in the NCT system.

Field	Shape	Role
`output`	$[B, T, d]$	Sent to next column via thalamic routing. Includes residual connection: $\mathbf{h}_{\text{out}} = \mathbf{x}_{\text{in}} + \mathbf{y}_{\text{L5}}$.
`prediction`	$[B, T, d]$	Top-down prediction $\hat{\mathbf{x}}_{i-1}$ for the column below. Used to compute PC error $\boldsymbol{\epsilon}_i$.
`thal_feedback`	$[B, T, d]$	L6 feedback signal sent to thalamic relay. Modulates routing for subsequent inputs.
`state`	$[B, T, d_c]$	Internal column state (post-oscillation, pre-L5). Used for monitoring and diagnostics.
`sigma2`	$[B, T, 1]$	Prediction precision from L5. Passed to next column's L5 as prior. Used for PC learning weight, surprise tagging, and σ^2-adaptive decoding.
`l23_input`	$[B, T, d_c]$	Input to L2/3 (after L1 modulation + interneuron gating). Needed for Hebbian update: $\mathbf{x}$ in $\Delta W = \eta \mathbf{y}\mathbf{x}^\top$.
`l23_output`	$[B, T, d_c]$	Output of L2/3 settling. Needed for Hebbian update: $\mathbf{y}$ in $\Delta W = \eta \mathbf{y}\mathbf{x}^\top$.

The residual connection ensures that even with all gates near zero (identity-at-init), the column passes information through unchanged. If $d_{\text{col}} \neq d_{\text{model}}$, a learned residual projection $W_{\text{residual}} \in \mathbb{R}^{d \times d}$ adapts the dimension.

10.12 Source Code

```
1
2  # ============================================================================
3  # Cortical Column -- The Fundamental Compute Unit
4  # Source: nct_v2/v11_column.py
5  # ============================================================================
6
7  """
8  NCT V11: Cortical Column -- The Fundamental Compute Unit
9  ========================================================
10
11 Replaces transformer blocks with a biologically-grounded 6-layer cortical column.
12
13 Architecture:
14   L1: Top-down modulation (apical dendrites, feedback from higher columns)
15   L2/3: Lateral association (self-organizing attention via Hebbian learning)
16   L4: Thalamic input (feedforward processing, the "sensory" layer)
17   L5: Output + prediction (per-position computation + top-down prediction)
18   L6: Thalamic feedback (modulates relay for future input)
19
20 Plus: Interneurons (VIP/SST/PV), Working Memory (NMDA/Silent/Phono),
21       Oscillatory dynamics, BCM homeostasis.
22
23 All modules are identity-at-init: gate_raw=-2.0, W_out=0.
24 """
25
26
27
28 # [bundled] from .v11_config import V11Config
29
30
31 #

32 # Column Output
33 #

34
35 @dataclass
36 class ColumnOutput:
37     """Output from a single cortical column."""
38     output: torch.Tensor           # [B, T, D_model] -- sent to next column
39     prediction: torch.Tensor       # [B, T, D_model] -- top-down prediction for column below
40     thal_feedback: torch.Tensor    # [B, T, D_model] -- sent to thalamus
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
    state: torch.Tensor            # [B, T, D_col] -- internal column state (for learning)
    sigma2: torch.Tensor           # [B, T, 1] -- prediction precision
    l23_input: torch.Tensor        # [B, T, D_col] -- L2/3 input (for Hebbian update)
    l23_output: torch.Tensor       # [B, T, D_col] -- L2/3 output (for Hebbian update)


#

# RMSNorm (lightweight normalization)
#


class RMSNorm(nn.Module):
    def __init__(self, d: int, eps: float = 1e-6):
        super().__init__()
        self.weight = nn.Parameter(torch.ones(d))
        self.eps = eps

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        rms = torch.sqrt(x.pow(2).mean(-1, keepdim=True) + self.eps)
        return x / rms * self.weight


#

# L4: Thalamic Input Layer
#


class L4_ThalamicInput(nn.Module):
    """
    Layer 4: Receives feedforward input from thalamus.
    The primary input layer of the cortical column.

    In V1: orientation-selective simple cells.
    In language: token-level feature extraction.
    """

    def __init__(self, cfg: V11Config):
        super().__init__()
        self.proj = nn.Linear(cfg.d_model, cfg.d_column, bias=False)
        self.norm = RMSNorm(cfg.d_column)
        # Identity at init: if d_model == d_column, start as identity
        if cfg.d_model == cfg.d_column:
            nn.init.eye_(self.proj.weight)
        else:
            nn.init.kaiming_normal_(self.proj.weight, nonlinearity='linear')

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """x: [B, T, D_model] → [B, T, D_col]"""
        return self.norm(self.proj(x))


#

# L2/3: Lateral Association Layer (Replaces Self-Attention)
#


class L23_LateralAssociation(nn.Module):
    """
    Layer 2/3: Windowed Hebbian Associative Recall -- NOT attention.

    Theorem B (Pehlevan & Chklovskii 2019):
      Hebbian feedforward + anti-Hebbian lateral → PCA subspace convergence.

    Mechanism (O(T×w×d), NOT O(T²)):
      1. Direct association: a_t = W_assoc @ x_t -- content-addressable recall
      2. Local window gather: each position t looks at [t-w, t] (theta period)
      3. Distance-weighted context: learned similarity within window
      4. Combine + k-WTA lateral inhibition (competitive coding)
      5. Association weights updated by Hebbian rule:  W  =   * (X^T X) / N

    No Q/K dot-product. No causal mask. No softmax attention.
    Causality is implicit: window only looks backward [t-w, t].
    W_assoc: content-addressable memory (Hebbian-learned, not trained by backprop).
    """

    def __init__(self, cfg: V11Config):
        super().__init__()
        self.cfg = cfg
        self.d_col = cfg.d_column
        self.n_heads = cfg.n_assoc_heads
        self.d_head = cfg.d_head
        self.window_size = cfg.assoc_window_size

        # Per-head association matrix: content-addressable recall
        # a_t = W_assoc[h] @ x_t -- what this pattern reminds the column of
        # Initialized as identity → recalls the input itself
        self.W_assoc = nn.Parameter(torch.zeros(self.n_heads, self.d_head, self.d_head))
        for h in range(self.n_heads):
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
            nn.init.eye_(self.W_assoc[h])

        # Value projection (what information to extract from window context)
        self.W_val = nn.Linear(cfg.d_column, cfg.d_column, bias=False)
        nn.init.eye_(self.W_val.weight)

        # Distance weighting for local window: per-head [H, D_h]
        # At init (zeros): all window positions weighted equally → uniform average
        self.W_dist = nn.Parameter(torch.zeros(self.n_heads, self.d_head))

        # Output projection (combines heads)
        self.W_out = nn.Linear(cfg.d_column, cfg.d_column, bias=False)
        nn.init.zeros_(self.W_out.weight)  # Identity at init via residual

        # Anti-Hebbian lateral weights for feature competition
        self.W_lateral = nn.Parameter(torch.zeros(self.n_heads, self.d_head, self.d_head))

        # k-WTA parameters
        self.k_wta_ratio = cfg.k_wta_ratio

        # Gate
        self.gate_raw = nn.Parameter(torch.tensor(cfg.gate_init))

        self.norm = RMSNorm(cfg.d_column)
        self.dropout = nn.Dropout(cfg.assoc_dropout)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """
        Windowed Hebbian Associative Recall -- O(T × w × d), NOT O(T²).

        x: [B, T, D_col]
        Returns: [B, T, D_col] association output (to be added as residual)

        Mechanism:
          1. Direct association: a_t = W_assoc @ x_t (content-addressable recall)
          2. Local window gather: each position looks back w positions (theta period)
          3. Distance-weighted context: learned similarity within window
          4. Combine: W_out(association + context)
          5. k-WTA + lateral inhibition (competitive coding)

        No Q/K dot-product. No causal mask. No RoPE.
        Causality is implicit: window only looks backward [t-w, t].
        """
        B, T, D = x.shape
        gate = torch.sigmoid(self.gate_raw)
        w = self.window_size

        x_norm = self.norm(x)

        # Reshape to multi-head: [B, T, H, D_h] → [B, H, T, D_h]
        x_heads = x_norm.view(B, T, self.n_heads, self.d_head).transpose(1, 2)

        # 1. Direct association (content-addressable recall per head)
        assoc = torch.matmul(x_heads, self.W_assoc)  # [B, H, T, D_h]

        # 2. Local window context (theta-period temporal binding)
        v = self.W_val(x_norm)
        v_heads = v.view(B, T, self.n_heads, self.d_head).transpose(1, 2)  # [B, H, T, D_h]

        # Pad on left for causal window: position t sees [t-w, ..., t]
        x_pad = F.pad(x_heads, (0, 0, w, 0))  # [B, H, T+w, D_h]
        v_pad = F.pad(v_heads, (0, 0, w, 0))  # [B, H, T+w, D_h]

        # Unfold to get windows: [B, H, T, D_h, w+1]
        x_win = x_pad.unfold(2, w + 1, 1)  # [B, H, T, D_h, w+1]
        v_win = v_pad.unfold(2, w + 1, 1)  # [B, H, T, D_h, w+1]

        # Permute to [B, H, T, w+1, D_h]
        x_win = x_win.permute(0, 1, 2, 4, 3)
        v_win = v_win.permute(0, 1, 2, 4, 3)

        # 3. Distance-weighted context
        # Compute feature-wise difference between current position and each window position
        x_curr = x_heads.unsqueeze(3)  # [B, H, T, 1, D_h]
        diff = x_curr - x_win  # [B, H, T, w+1, D_h]

        # Learned distance weighting: dot(diff, W_dist[h]) → scalar per window position
        # W_dist[h] shape: [D_h], broadcast to [1, H, 1, 1, D_h]
        w_dist = self.W_dist[None, :, None, None, :]  # [1, H, 1, 1, D_h]
        scores = (diff * w_dist).sum(dim=-1)  # [B, H, T, w+1]

        # Softplus → non-negative weights, then normalize
        weights = F.softplus(scores)  # [B, H, T, w+1]
        weights = weights / (weights.sum(dim=-1, keepdim=True) + 1e-8)
        weights = self.dropout(weights)

        # Weighted sum of values in window
        context = (weights.unsqueeze(-1) * v_win).sum(dim=3)  # [B, H, T, D_h]

        # 4. Combine association + windowed context
        combined = assoc + context  # [B, H, T, D_h]
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        # 5. Lateral inhibition (anti-Hebbian decorrelation)
        if self.W_lateral.abs().max() > 1e-8:  # Skip if zero (init)
            lateral = torch.matmul(combined, self.W_lateral)  # [B, H, T, D_h]
            combined = combined - lateral

        # 6. k-WTA sparsity on features per head (competitive activation coding)
        k = max(1, int(self.d_head * self.k_wta_ratio))
        topk_vals, topk_idx = combined.topk(k, dim=-1)
        mask = torch.zeros_like(combined)
        mask.scatter_(-1, topk_idx, 1.0)
        combined = combined * mask

        # 7. Reshape back: [B, H, T, D_h] → [B, T, D]
        combined = combined.transpose(1, 2).contiguous().view(B, T, D)

        # Output projection
        out = self.W_out(combined)

        return gate * out

    #        Structural Plasticity (Theorem E)
    # Synaptogenesis + pruning on lateral weights.

    def init_structural_plasticity(self):
        """Initialize structural plasticity masks and counters.
        Called once after construction if structural_plasticity=True.
        """
        d = self.d_head
        n = self.n_heads
        # Binary mask: 1 = active connection, 0 = pruned
        self.register_buffer(
            'assoc_mask', torch.ones(n, d, d)
        )
        # Counter: steps since weight was below pruning threshold
        self.register_buffer(
            'below_thresh_count', torch.zeros(n, d, d, dtype=torch.long)
        )
        self._structural_init = True

    @torch.no_grad()
    def structural_plasticity_step(
        self,
        error_magnitude: float,
        prune_threshold: float = 0.01,
        growth_threshold: float = 2.0,
        prune_interval: int = 1000,
    ):
        """
        Structural plasticity: prune weak connections, grow new ones where needed.

        Theorem E: The brain continuously prunes and grows synapses.
        - Pruning: connections with |w| < threshold for sustained periods die
        - Synaptogenesis: high prediction error triggers new random connections

        Args:
            error_magnitude: mean PE across tokens (drives synaptogenesis)
            prune_threshold: weight magnitude below which connections can be pruned
            growth_threshold: error above which new connections sprout
            prune_interval: steps below threshold before permanent pruning
        """
        if not hasattr(self, '_structural_init'):
            return

        mask = self.assoc_mask
        counts = self.below_thresh_count

        for h in range(self.n_heads):
            w = self.W_assoc.data[h]

            # Track how long each weight has been below threshold
            below = (w.abs() < prune_threshold) & (mask[h] > 0)
            counts[h][below] += 1
            counts[h][~below] = 0

            # Prune: connections below threshold for prune_interval steps
            to_prune = counts[h] >= prune_interval
            if to_prune.any():
                mask[h][to_prune] = 0
                w[to_prune] = 0
                counts[h][to_prune] = 0

            # Apply mask to ensure pruned connections stay dead
            w *= mask[h]
            self.W_assoc.data[h] = w

        # Synaptogenesis: high error → grow new connections
        if error_magnitude > growth_threshold:
            for h in range(self.n_heads):
                pruned = mask[h] == 0
                n_pruned = pruned.sum().item()
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
                if n_pruned == 0:
                    continue
                # Restore a fraction of pruned connections with small random weights
                n_restore = max(1, int(n_pruned * 0.05))  # 5% of pruned per event
                pruned_indices = pruned.nonzero()
                perm = torch.randperm(len(pruned_indices), device=mask.device)[:n_restore]
                for idx in pruned_indices[perm]:
                    r, c = idx[0].item(), idx[1].item()
                    mask[h, r, c] = 1
                    self.W_assoc.data[h, r, c] = torch.randn(1, device=mask.device).item() * 0.01

    def get_connection_density(self) -> float:
        """Return fraction of active connections (1.0 = fully connected)."""
        if not hasattr(self, '_structural_init'):
            return 1.0
        return self.assoc_mask.float().mean().item()

    @torch.no_grad()
    def hebbian_update(self, x: torch.Tensor, lr: Optional[float] = None):
        """
        Hebbian learning update for W_assoc and W_lateral.
        Called after forward pass with the same input.

          W_assoc   =    * (X^T X) / N  (strengthen correlated features)
          W_lateral   =    * (Y^T Y) / N  (decorrelate outputs, anti-Hebbian)
        """
        if lr is None:
            lr = self.cfg.hebbian_lr

        B, T, D = x.shape
        N = B * T
        x_heads = x.view(B, T, self.n_heads, self.d_head).transpose(1, 2)

        for h in range(self.n_heads):
            x_h = x_heads[:, h].reshape(-1, self.d_head)  # [BT, D_h]

            # Hebbian:  W  =    * correlation matrix
            corr = (x_h.T @ x_h) / N
            # Remove diagonal (self-correlations add instability)
            corr.fill_diagonal_(0)
            self.W_assoc.data[h] += lr * corr

            # Respect structural plasticity mask
            if hasattr(self, '_structural_init'):
                self.W_assoc.data[h] *= self.assoc_mask[h]

            # Normalize to prevent explosion
            norm = self.W_assoc.data[h].norm()
            if norm > 5.0:
                self.W_assoc.data[h] *= 5.0 / norm

    @torch.no_grad()
    def anti_hebbian_update(self, y: torch.Tensor, lr: Optional[float] = None):
        """
        Anti-Hebbian update for lateral weights (decorrelation).
        y is the output of the association layer.

          W_lat   =    * (Y^T Y) / N -    * I  (decorrelate, preserve norm)
        """
        if lr is None:
            lr = self.cfg.hebbian_lr

        B, T, D = y.shape
        N = B * T
        y_heads = y.view(B, T, self.n_heads, self.d_head).transpose(1, 2)

        for h in range(self.n_heads):
            y_h = y_heads[:, h].reshape(-1, self.d_head)
            decorr = (y_h.T @ y_h) / N
            # Target: identity covariance (decorrelated outputs)
            decorr -= torch.eye(self.d_head, device=y.device)
            self.W_lateral.data[h] += lr * decorr

            norm = self.W_lateral.data[h].norm()
            if norm > 3.0:
                self.W_lateral.data[h] *= 3.0 / norm

    @torch.no_grad()
    def hebbian_update_bcm(self, pre_input: torch.Tensor, post_output: torch.Tensor,
                           bcm_theta: torch.Tensor, lr: Optional[float] = None):
        """
        BCM-modulated Hebbian update for W_assoc.
         w      y(y -  _BCM)x -- potentiate above threshold, depress below.

        pre_input: [B, T, D_col] -- input to L2/3 (pre-synaptic)
        post_output: [B, T, D_col] -- output of L2/3 (post-synaptic)
        bcm_theta: scalar tensor -- BCM sliding threshold
        """
        if lr is None:
            lr = self.cfg.hebbian_lr

```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        B, T, D = pre_input.shape
        N = B * T
        pre_heads = pre_input.view(B, T, self.n_heads, self.d_head).transpose(1, 2)
        post_heads = post_output.view(B, T, self.n_heads, self.d_head).transpose(1, 2)

        for h in range(self.n_heads):
            pre_h = pre_heads[:, h].reshape(-1, self.d_head)   # [BT, D_h]
            post_h = post_heads[:, h].reshape(-1, self.d_head)  # [BT, D_h]

            # BCM modulation: y(y - sqrt( ))x
            bcm_mod = post_h * (post_h - bcm_theta.sqrt())  # [BT, D_h]
            update = (bcm_mod.T @ pre_h) / N  # [D_h, D_h]
            update.fill_diagonal_(0)

            self.W_assoc.data[h] += lr * update

            # Respect structural plasticity mask
            if hasattr(self, '_structural_init'):
                self.W_assoc.data[h] *= self.assoc_mask[h]

            norm = self.W_assoc.data[h].norm()
            if norm > 5.0:
                self.W_assoc.data[h] *= 5.0 / norm

    @torch.no_grad()
    def hebbian_update_error_modulated(
        self,
        pre_input: torch.Tensor,
        post_output: torch.Tensor,
        error: torch.Tensor,
        bcm_theta: torch.Tensor,
        lr: Optional[float] = None,
        error_weight: float = 0.5,
    ):
        """
        Error-driven Hebbian: BCM × prediction_error_magnitude × (pre^T × post).

        The brain's Hebbian learning is NOT blind correlation -- it's modulated
        by prediction error. High error = more plasticity (learn from mistakes).

        error_weight: 0 = pure BCM (blind), 1 = fully error-driven
        """
        if lr is None:
            lr = self.cfg.hebbian_lr

        B, T, D = pre_input.shape
        N = B * T

        # Per-token error magnitude [B, T, 1]
        error_mag = error.detach().norm(dim=-1, keepdim=True)  # [B, T, 1]
        # Normalize to mean 1 to avoid scaling issues
        error_scale = error_mag / (error_mag.mean() + 1e-8)

        pre_heads = pre_input.view(B, T, self.n_heads, self.d_head).transpose(1, 2)
        post_heads = post_output.view(B, T, self.n_heads, self.d_head).transpose(1, 2)
        # Broadcast error scale to heads: [B, 1, T, 1]
        err_heads = error_scale.unsqueeze(1)

        for h in range(self.n_heads):
            pre_h = pre_heads[:, h]   # [B, T, D_h]
            post_h = post_heads[:, h]  # [B, T, D_h]
            err_h = err_heads[:, 0]    # [B, T, 1]

            # BCM modulation: y(y - sqrt( ))
            bcm_mod = post_h * (post_h - bcm_theta.sqrt())  # [B, T, D_h]

            # Error modulation: blend blind BCM with error-weighted BCM
            # When error_weight=0: pure BCM. When error_weight=1: error × BCM
            modulated = bcm_mod * ((1.0 - error_weight) + error_weight * err_h)

            # Hebbian update
            modulated_flat = modulated.reshape(-1, self.d_head)  # [BT, D_h]
            pre_flat = pre_h.reshape(-1, self.d_head)
            update = (modulated_flat.T @ pre_flat) / N
            update.fill_diagonal_(0)

            self.W_assoc.data[h] += lr * update

            # Respect structural plasticity mask
            if hasattr(self, '_structural_init'):
                self.W_assoc.data[h] *= self.assoc_mask[h]

            norm = self.W_assoc.data[h].norm()
            if norm > 5.0:
                self.W_assoc.data[h] *= 5.0 / norm


#

# L1: Top-Down Modulation Layer
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```python
#


class L1_TopDownModulation(nn.Module):
    """
    Layer 1: Receives top-down feedback from higher cortical areas.
    Modulates column processing via apical dendrites.

    In the brain: L1 contains apical dendrite tufts from L5 pyramidal cells
    of the same column, plus axons from higher areas. Top-down signals
    modulate (but don't drive) the column's processing.
    """

    def __init__(self, cfg: V11Config):
        super().__init__()
        self.proj = nn.Linear(cfg.d_model, cfg.d_column, bias=False)
        nn.init.zeros_(self.proj.weight)  # No top-down initially
        self.gate_raw = nn.Parameter(torch.tensor(cfg.gate_init))
        self.norm = RMSNorm(cfg.d_column)

    def forward(self, td_context: torch.Tensor) -> torch.Tensor:
        """
        td_context: [B, T, D_model] -- top-down from higher column
        Returns: [B, T, D_col] -- modulatory signal (added to L4 output)
        """
        gate = torch.sigmoid(self.gate_raw)
        return gate * self.norm(self.proj(td_context))


#

# L5: Output + Prediction Generation
#


class L5_OutputPrediction(nn.Module):
    """
    Layer 5: Output computation + top-down prediction generation.

    Two functions:
    1. Compute output for next column (like FFN: up-project → SwiGLU → down-project)
    2. Generate top-down prediction for the column below (predictive coding)

    The prediction is the KEY to local learning:
       _l = actual_input_l - prediction_from_{l+1}
      This error drives local weight updates (Theorem A).
    """

    def __init__(self, cfg: V11Config):
        super().__init__()
        self.dendritic = cfg.l5_dendritic
        self.kwta = cfg.l5_kwta
        self.k_wta_ratio = cfg.k_wta_ratio

        # Output computation (SwiGLU FFN -- SiLU ~ biological f-I curve)
        self.W_up = nn.Linear(cfg.d_column, cfg.d_ff, bias=False)
        self.W_gate = nn.Linear(cfg.d_column, cfg.d_ff, bias=False)
        self.W_down = nn.Linear(cfg.d_ff, cfg.d_model, bias=False)
        nn.init.zeros_(self.W_down.weight)  # Identity at init via residual

        # Dendritic computation (basal + apical gating)
        if self.dendritic:
            # Basal dendritic gate: bottom-up context from L4 input
            # bias=3.0 → sigmoid(3)~0.95 → nearly open at init
            self.W_dend = nn.Linear(cfg.d_column, cfg.d_ff, bias=True)
            nn.init.zeros_(self.W_dend.weight)
            nn.init.constant_(self.W_dend.bias, 3.0)

            # Apical dendritic gate: top-down context from L1
            self.W_apical = nn.Linear(cfg.d_column, cfg.d_ff, bias=True)
            nn.init.zeros_(self.W_apical.weight)
            nn.init.constant_(self.W_apical.bias, 3.0)

            #  ²-adaptive modulation: prediction uncertainty modulates dendrites
            self.W_sigma_adapt = nn.Linear(1, cfg.d_ff, bias=False)
            nn.init.zeros_(self.W_sigma_adapt.weight)

        # Prediction head: generates prediction of input to column below
        self.W_predict = nn.Linear(cfg.d_column, cfg.d_model, bias=False)
        nn.init.zeros_(self.W_predict.weight)  # No predictions initially → max error → max learning

        # Precision estimation ( ² for this column's predictions)
        self.W_sigma = nn.Linear(cfg.d_column, 1, bias=True)
        nn.init.zeros_(self.W_sigma.weight)
        _sigma2_bias = math.log(math.exp(cfg.sigma2_init) - 1) if cfg.sigma2_init > 0.01 else 0.0
        nn.init.constant_(self.W_sigma.bias, _sigma2_bias)

        self.norm = RMSNorm(cfg.d_column)
        self.dropout = nn.Dropout(cfg.dropout)

    def forward(self, h: torch.Tensor,
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
                l4_input: Optional[torch.Tensor] = None,
                td_context: Optional[torch.Tensor] = None,
                sigma2_prev: Optional[torch.Tensor] = None,
                ) -> Tuple[torch.Tensor, torch.Tensor, torch.Tensor]:
        """
        Dendritic FFN: SwiGLU core + basal/apical dendritic gating + k-WTA sparsity.

        h: [B, T, D_col] -- column state after L2/3
        l4_input: [B, T, D_col] -- bottom-up from L4 (basal dendrite context)
        td_context: [B, T, D_col] -- top-down from L1 (apical dendrite context)
        sigma2_prev: [B, T, 1] -- previous column's prediction precision

        Returns:
          output: [B, T, D_model] -- sent to next column
          prediction: [B, T, D_model] -- top-down prediction for column below
          sigma2: [B, T, 1] -- prediction precision
        """
        h_norm = self.norm(h)

        # SwiGLU core (SiLU ~ biological frequency-current curve)
        ff = F.silu(self.W_gate(h_norm)) * self.W_up(h_norm)  # [B, T, d_ff]

        # Dendritic gating
        if self.dendritic:
            # Basal dendritic gate (bottom-up: what the input says to compute)
            if l4_input is not None:
                dend_logit = self.W_dend(l4_input)  # [B, T, d_ff]
            else:
                # Use bias only -- sigmoid(3.0)~0.95 → nearly open
                dend_logit = self.W_dend.bias.unsqueeze(0).unsqueeze(0).expand_as(ff)

            #  ²-adaptive: high uncertainty → modulate dendritic gate
            if sigma2_prev is not None:
                dend_logit = dend_logit + self.W_sigma_adapt(sigma2_prev)

            ff = ff * torch.sigmoid(dend_logit)

            # Apical dendritic gate (top-down: what higher areas expect)
            if td_context is not None:
                ff = ff * torch.sigmoid(self.W_apical(td_context))

        # k-WTA sparsity (cortical-level sparse coding)
        if self.kwta:
            k = max(1, int(ff.shape[-1] * self.k_wta_ratio))
            topk_vals, topk_idx = ff.topk(k, dim=-1)
            mask = torch.zeros_like(ff)
            mask.scatter_(-1, topk_idx, 1.0)
            ff = ff * mask

        output = self.W_down(self.dropout(ff))

        # Top-down prediction
        prediction = self.W_predict(h_norm)

        # Per-token precision:  ²(h) = softplus(W_sigma(h)) +
        sigma2 = F.softplus(self.W_sigma(h_norm)) + 1e-6  # [B, T, 1]

        return output, prediction, sigma2


#

# L6: Thalamic Feedback
#


class L6_ThalamicFeedback(nn.Module):
    """
    Layer 6: Projects back to thalamus.
    Modulates thalamic relay for future input processing.

    In the brain: L6 corticothalamic neurons sharpen thalamic responses
    and control the "gain" of thalamic relay for the next processing cycle.
    """

    def __init__(self, cfg: V11Config):
        super().__init__()
        self.proj = nn.Linear(cfg.d_column, cfg.d_model, bias=False)
        nn.init.zeros_(self.proj.weight)  # No feedback initially
        self.gate_raw = nn.Parameter(torch.tensor(cfg.gate_init))

    def forward(self, h: torch.Tensor) -> torch.Tensor:
        """
        h: [B, T, D_col] -- column state
        Returns: [B, T, D_model] -- thalamic feedback signal
        """
        gate = torch.sigmoid(self.gate_raw)
        return gate * self.proj(h)
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```python
#

# Column Interneurons (VIP → SST → PV/Chandelier)
#


class ColumnInterneurons(nn.Module):
    """
    Interneurons within a cortical column.

    Circuit: VIP        SST        Pyramidal (apical)
             PV        Pyramidal (axon initial segment)

    VIP (Thm 38): Disinhibition -- releases SST suppression when input is novel
    SST (Thm 46): Input gate -- suppresses apical dendrites (top-down influence)
    PV  (Thm 41): Output veto -- axon initial segment gating

    All identity-at-init: gate_raw = -2.0, output weights = 0.
    """

    def __init__(self, cfg: V11Config):
        super().__init__()
        d = cfg.d_column
        d_i = cfg.d_inhibitory

        # VIP: Vasoactive Intestinal Peptide interneuron (disinhibition)
        self.vip_enabled = cfg.vip_enabled
        if cfg.vip_enabled:
            self.vip_in = nn.Linear(d, d_i, bias=False)
            self.vip_out = nn.Linear(d_i, d, bias=False)
            nn.init.zeros_(self.vip_out.weight)
            self.vip_gate_raw = nn.Parameter(torch.tensor(cfg.gate_init))
            self.vip_norm = RMSNorm(d)

        # SST: Somatostatin interneuron (Martinotti cell, input gate)
        self.sst_enabled = cfg.sst_enabled
        if cfg.sst_enabled:
            self.sst_in = nn.Linear(d, d_i, bias=False)
            self.sst_out = nn.Linear(d_i, d, bias=False)
            nn.init.zeros_(self.sst_out.weight)
            self.sst_gate_raw = nn.Parameter(torch.tensor(cfg.gate_init))
            self.sst_tonic = 0.10  # Intrinsic Na+ currents (constant)
            self.sst_norm = RMSNorm(d)

        # PV: Parvalbumin interneuron (Chandelier cell, output veto)
        self.pv_enabled = cfg.pv_enabled
        if cfg.pv_enabled:
            self.pv_in = nn.Linear(d, d_i, bias=False)
            self.pv_score = nn.Linear(d_i, 1, bias=True)
            nn.init.zeros_(self.pv_score.weight)
            nn.init.constant_(self.pv_score.bias, 5.0)  # Nearly open at init
            self.pv_gate_raw = nn.Parameter(torch.tensor(cfg.gate_init))
            self.pv_norm = RMSNorm(d)

    def apply_dendritic_inhibition(self, h: torch.Tensor,
                                   td_signal: Optional[torch.Tensor] = None) -> torch.Tensor:
        """
        SST + VIP: Pre-L2/3 apical dendrite gating.
        VIP detects novelty in top-down signal → disinhibits SST.
        SST suppresses apical dendrites (input gating).

        Applied BEFORE L2/3 (correct biological placement).
        """
        # VIP disinhibition (reads top-down novelty)
        vip_disinhibition = None
        if self.vip_enabled and td_signal is not None:
            vip_g = torch.sigmoid(self.vip_gate_raw)
            vip_act = torch.sigmoid(self.vip_out(F.gelu(self.vip_in(self.vip_norm(td_signal)))))
            vip_disinhibition = vip_g * vip_act  # [B, T, D]

        # SST dendritic inhibition
        if self.sst_enabled:
            sst_g = torch.sigmoid(self.sst_gate_raw)
            sst_act = torch.sigmoid(self.sst_out(F.gelu(self.sst_in(self.sst_norm(h)))))
            sst_total = self.sst_tonic + (1.0 - self.sst_tonic) * sst_g * sst_act
            # VIP → SST disinhibition
            if vip_disinhibition is not None:
                sst_effective = sst_total * (1.0 - vip_disinhibition)
            else:
                sst_effective = sst_total
            h = h * (1.0 - sst_effective)

        return h

    def apply_output_veto(self, h: torch.Tensor) -> torch.Tensor:
        """
        PV/Chandelier: Post-computation axon initial segment veto.
        Applied AFTER working memory/oscillations, BEFORE L5 output.

        Correct biological placement: PV gates whether the pyramidal cell fires.
        """
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        if self.pv_enabled:
            pv_g = torch.sigmoid(self.pv_gate_raw)
            pv_logits = self.pv_score(F.gelu(self.pv_in(self.pv_norm(h))))  # [B, T, 1]
            pass_prob = torch.sigmoid(pv_logits)  # Near 1.0 at init (bias=5.0)
            # Straight-through: hard gate forward, soft gradient backward
            hard_pass = (pass_prob > 0.5).float()
            pass_gate = hard_pass - pass_prob.detach() + pass_prob
            h = h * ((1.0 - pv_g) + pv_g * pass_gate)

        return h

    def forward(self, h: torch.Tensor, td_signal: Optional[torch.Tensor] = None) -> torch.Tensor:
        """Backward-compatible: applies both stages. Use split methods for correct placement."""
        h = self.apply_dendritic_inhibition(h, td_signal)
        h = self.apply_output_veto(h)
        return h


#

# Column Working Memory
#


class ColumnWorkingMemory(nn.Module):
    """
    Working memory within a cortical column.
    Three timescales (brain WM hierarchy):

    1. NMDA attractor slots (Thm 31): Slow decay (~277 tokens), 4 competitive slots
    2. Activity-silent trace (Thm 35): Tsodyks-Markram facilitation/depression
    3. Phonological loop (Thm 36): Decay + rehearsal, 8 slots

    All gated and identity-at-init.
    """

    def __init__(self, cfg: V11Config):
        super().__init__()
        self.cfg = cfg
        d = cfg.d_column

        # Persistent WM state (volatile -- like real WM, not saved to disk)
        # Persists across sequential forward calls, resets on batch size change.
        self._nmda_state: Optional[torch.Tensor] = None  # [B, K, d_slot]
        self._silent_u: Optional[torch.Tensor] = None     # [B, d_trace]
        self._silent_x: Optional[torch.Tensor] = None     # [B, d_trace]
        self._phono_state: Optional[torch.Tensor] = None   # [B, K, d_slot]
        self._phono_ptr: int = 0
        self._last_batch_size: int = -1

        if not cfg.wm_enabled:
            return

        #         NMDA Attractor Slots

        K = cfg.nmda_slots
        d_slot = cfg.nmda_d_slot
        self.nmda_K = K
        self.nmda_d_slot = d_slot

        self.nmda_write = nn.Linear(d, d_slot, bias=False)
        self.nmda_read = nn.Linear(d_slot, d, bias=False)
        nn.init.zeros_(self.nmda_read.weight)
        self.nmda_score = nn.Linear(d, K, bias=False)
        self.nmda_decay_bias = nn.Parameter(torch.full((K,), cfg.nmda_decay_init))
        self.nmda_gate_raw = nn.Parameter(torch.tensor(cfg.gate_init))
        self.nmda_norm = RMSNorm(d)

        #         Activity-Silent Trace

        d_trace = cfg.silent_d_trace
        self.silent_d_trace = d_trace

        self.silent_store = nn.Linear(d, d_trace, bias=False)
        self.silent_retrieve = nn.Linear(d_trace, d, bias=False)
        nn.init.zeros_(self.silent_retrieve.weight)
        self.silent_write_gate = nn.Linear(d, 1, bias=True)
        nn.init.constant_(self.silent_write_gate.bias, -1.0)
        self.silent_facil_decay = nn.Parameter(torch.tensor(cfg.silent_facil_init))
        self.silent_depress_decay = nn.Parameter(torch.tensor(cfg.silent_depress_init))
        self.silent_depress_frac = nn.Parameter(torch.tensor(-2.0))
        self.silent_gate_raw = nn.Parameter(torch.tensor(cfg.gate_init))
        self.silent_norm = RMSNorm(d)

        #         Phonological Loop

        K_p = cfg.phono_capacity
        d_p = cfg.phono_d_slot
        self.phono_K = K_p
        self.phono_d_slot = d_p

```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        self.phono_write = nn.Linear(d, d_p, bias=False)
        self.phono_query = nn.Linear(d, d_p, bias=False)
        self.phono_read = nn.Linear(d_p, d, bias=False)
        nn.init.zeros_(self.phono_read.weight)
        self.phono_write_gate = nn.Linear(d, 1, bias=True)
        nn.init.constant_(self.phono_write_gate.bias, -2.0)
        self.phono_decay = nn.Parameter(torch.tensor(4.0))  # sigmoid(4)~0.982
        self.phono_rehearsal_str = nn.Parameter(torch.tensor(-1.0))  # sigmoid(-1)~0.27
        self.phono_gate_raw = nn.Parameter(torch.tensor(cfg.gate_init))
        self.phono_norm = RMSNorm(d)

    def reset_wm(self):
        """Reset working memory state. Call on new context/document."""
        self._nmda_state = None
        self._silent_u = None
        self._silent_x = None
        self._phono_state = None
        self._phono_ptr = 0
        self._last_batch_size = -1

    @staticmethod
    def _scatter_slot(slots: torch.Tensor, idx: int, new_val: torch.Tensor) -> torch.Tensor:
        """Functionally update slots[:, idx] = new_val without in-place ops.
        slots: [B, K, D], new_val: [B, D] → [B, K, D]"""
        K = slots.shape[1]
        mask = torch.zeros(K, device=slots.device, dtype=slots.dtype)
        mask[idx] = 1.0
        mask = mask.view(1, K, 1)  # [1, K, 1]
        return slots * (1.0 - mask) + new_val.unsqueeze(1) * mask

    def forward(self, h: torch.Tensor) -> torch.Tensor:
        """
        h: [B, T, D_col]
        Returns: [B, T, D_col] -- h with WM contributions added

        All operations are functional (no in-place) to support autograd backward.
        """
        if not self.cfg.wm_enabled:
            return h

        B, T, D = h.shape

        # Reset persistent state if batch size changed (new context)
        if B != self._last_batch_size:
            self.reset_wm()
            self._last_batch_size = B

        #        NMDA Attractor Slots

        nmda_g = torch.sigmoid(self.nmda_gate_raw)
        if nmda_g > 0.01:
            decay = torch.sigmoid(self.nmda_decay_bias)  # [K]
            # Persistent: use stored state if available (cross-sequence WM)
            if self._nmda_state is not None and self._nmda_state.shape[0] == B:
                slots = self._nmda_state.detach().to(h.device)
            else:
                slots = torch.zeros(B, self.nmda_K, self.nmda_d_slot, device=h.device)

            wm_outputs = []
            for t in range(T):
                h_t = self.nmda_norm(h[:, t])  # [B, D]
                # Slot competition
                scores = self.nmda_score(h_t)  # [B, K]
                write_prob = F.softmax(scores, dim=-1)  # [B, K]
                # Write -- vectorized over K (each slot update is independent)
                content = self.nmda_write(h_t)  # [B, d_slot]
                w = write_prob.unsqueeze(-1)  # [B, K, 1]
                slots = decay.view(1, -1, 1) * slots + w * content.unsqueeze(1)
                # Read (attention over slots)
                read_scores = torch.bmm(
                    slots, content.unsqueeze(-1)
                ).squeeze(-1)  # [B, K]
                read_weights = F.softmax(read_scores, dim=-1)  # [B, K]
                read = torch.bmm(read_weights.unsqueeze(1), slots).squeeze(1)  # [B, d_slot]
                wm_outputs.append(self.nmda_read(read))

            wm_out = torch.stack(wm_outputs, dim=1)  # [B, T, D]
            h = h + nmda_g * wm_out
            # Persist NMDA state for next forward call
            self._nmda_state = slots.detach()

        #        Activity-Silent Trace

        silent_g = torch.sigmoid(self.silent_gate_raw)
        if silent_g > 0.01:
            facil_a = torch.sigmoid(self.silent_facil_decay)
            depress_a = torch.sigmoid(self.silent_depress_decay)
            depress_f = torch.sigmoid(self.silent_depress_frac) * 0.5

            # Persistent: use stored state if available
            if self._silent_u is not None and self._silent_u.shape[0] == B:
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
                u = self._silent_u.detach().to(h.device)
                x = self._silent_x.detach().to(h.device)
            else:
                u = torch.zeros(B, self.silent_d_trace, device=h.device)  # Facilitation
                x = torch.ones(B, self.silent_d_trace, device=h.device)   # Depression (available)

            trace_outputs = []
            for t in range(T):
                h_t = self.silent_norm(h[:, t])
                write_g = torch.sigmoid(self.silent_write_gate(h_t))  # [B, 1]
                content = self.silent_store(h_t)  # [B, d_trace]

                # Tsodyks-Markram dynamics
                release = u * x  # Effective release
                trace_outputs.append(self.silent_retrieve(release))

                # Update facilitation: u → facil_a * u + (1-u) * write_g * |content|
                u = facil_a * u + (1.0 - u) * write_g * torch.abs(content)
                # Update depression: x → depress_a * x + (1-x)/tau - depress_f * u * x
                x = depress_a * x + (1.0 - depress_a) * (1.0 - x) - depress_f * u * x
                x = x.clamp(min=0.0, max=1.0)

            trace_out = torch.stack(trace_outputs, dim=1)  # [B, T, D]
            h = h + silent_g * trace_out
            # Persist Silent state for next forward call
            self._silent_u = u.detach()
            self._silent_x = x.detach()

        #        Phonological Loop

        phono_g = torch.sigmoid(self.phono_gate_raw)
        if phono_g > 0.01:
            p_decay = torch.sigmoid(self.phono_decay)
            r_str = torch.sigmoid(self.phono_rehearsal_str)

            # Persistent: use stored state if available
            if self._phono_state is not None and self._phono_state.shape[0] == B:
                slots = self._phono_state.detach().to(h.device)
                write_ptr = self._phono_ptr
            else:
                slots = torch.zeros(B, self.phono_K, self.phono_d_slot, device=h.device)
                write_ptr = 0

            phono_outputs = []
            for t in range(T):
                h_t = self.phono_norm(h[:, t])
                w_gate = torch.sigmoid(self.phono_write_gate(h_t))  # [B, 1]

                # Passive decay (functional -- creates new tensor)
                slots = p_decay * slots

                # Rehearsal (functional scatter update)
                if t > 0:
                    most_recent = write_ptr - 1 if write_ptr > 0 else self.phono_K - 1
                    rehearsed = slots[:, most_recent] + r_str * slots[:, most_recent]
                    slots = self._scatter_slot(slots, most_recent, rehearsed)

                # Write (functional scatter update)
                content = self.phono_write(h_t)  # [B, d_slot]
                idx = write_ptr % self.phono_K
                new_slot = slots[:, idx] * (1.0 - w_gate) + w_gate * content
                slots = self._scatter_slot(slots, idx, new_slot)
                write_ptr = (write_ptr + 1) % self.phono_K

                # Read (query-based attention)
                query = self.phono_query(h_t)  # [B, d_slot]
                read_scores = torch.bmm(
                    slots, query.unsqueeze(-1)
                ).squeeze(-1)  # [B, K]
                read_weights = F.softmax(read_scores, dim=-1)
                read = torch.bmm(read_weights.unsqueeze(1), slots).squeeze(1)
                phono_outputs.append(self.phono_read(read))

            phono_out = torch.stack(phono_outputs, dim=1)  # [B, T, D]
            h = h + phono_g * phono_out
            # Persist Phono state for next forward call
            self._phono_state = slots.detach()
            self._phono_ptr = write_ptr

        return h


#

# Column Oscillatory Dynamics
#


class ColumnOscillations(nn.Module):
    """
    Oscillatory dynamics within a cortical column (Thm 53-55).
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
    Nested oscillations: gamma → theta → alpha → delta
    Cross-frequency phase-amplitude coupling (PAC):
      A_gamma = 1 +  1*cos( _ ) +  2*cos( _ ) -  3*(1+cos( _ ))/2

    Periods are learnable via bare softplus (positive guarantee).
    """

    def __init__(self, cfg: V11Config):
        super().__init__()
        self.enabled = cfg.oscillations_enabled
        if not self.enabled:
            return

        d = cfg.d_column
        r = 32  # Rank-reduced interaction

        # Learnable periods
        self.period_theta_raw = nn.Parameter(torch.tensor(math.log(math.exp(cfg.theta_period) - 1)))
        self.period_alpha_raw = nn.Parameter(torch.tensor(math.log(math.exp(cfg.alpha_period) - 1)))
        self.period_delta_raw = nn.Parameter(torch.tensor(math.log(math.exp(cfg.delta_period) - 1)))

        # PAC coupling strengths -- theta-gamma coupling starts active (small)
        # Jensen & Colgin 2007: theta-gamma PAC is present even at rest
        self.kappa1_raw = nn.Parameter(torch.tensor(0.3))   # theta-gamma (primary)
        self.kappa2_raw = nn.Parameter(torch.tensor(0.0))   # delta-gamma
        self.kappa3_raw = nn.Parameter(torch.tensor(0.0))   # alpha suppression

        # Phase-content interaction
        self.W_content = nn.Linear(d, r, bias=False)
        self.W_phase = nn.Linear(6, r, bias=False)  # sin/cos for theta, alpha, delta
        self.W_out = nn.Linear(r, d, bias=False)
        nn.init.normal_(self.W_out.weight, std=0.01)  # Small non-zero → oscillations provide signal

        self.gate_raw = nn.Parameter(torch.tensor(cfg.gate_init))

    def forward(self, h: torch.Tensor) -> torch.Tensor:
        """h: [B, T, D_col] → [B, T, D_col]"""
        if not self.enabled:
            return h

        B, T, D = h.shape
        gate = torch.sigmoid(self.gate_raw)

        # Compute periods
        P_theta = F.softplus(self.period_theta_raw)
        P_alpha = F.softplus(self.period_alpha_raw)
        P_delta = F.softplus(self.period_delta_raw)

        # Compute phases for each position
        t = torch.arange(T, device=h.device, dtype=h.dtype)
        phi_theta = 2 * math.pi * t / P_theta
        phi_alpha = 2 * math.pi * t / P_alpha
        phi_delta = 2 * math.pi * t / P_delta

        # Phase features [T, 6]
        phase_features = torch.stack([
            phi_theta.sin(), phi_theta.cos(),
            phi_alpha.sin(), phi_alpha.cos(),
            phi_delta.sin(), phi_delta.cos(),
        ], dim=-1)

        # PAC amplitude modulation
        kappa1 = torch.tanh(self.kappa1_raw)
        kappa2 = torch.tanh(self.kappa2_raw)
        kappa3 = torch.tanh(self.kappa3_raw)
        A_gamma = 1.0 + kappa1 * phi_theta.cos() + kappa2 * phi_delta.cos() \
                  - kappa3 * (1.0 + phi_alpha.cos()) / 2.0  # [T]

        # Phase-content interaction
        content = self.W_content(h)  # [B, T, r]
        phase_proj = self.W_phase(phase_features).unsqueeze(0)  # [1, T, r]
        interaction = torch.tanh(content + phase_proj)  # [B, T, r]
        modulation = self.W_out(interaction)  # [B, T, D]

        # Apply with PAC amplitude
        A = A_gamma.unsqueeze(0).unsqueeze(-1)  # [1, T, 1]
        return h + gate * A * modulation


#

# BCM Homeostasis
#


class BCMHomeostasis(nn.Module):
    """
    Bienenstock-Cooper-Munro homeostatic plasticity (Thm 7).

    Sliding threshold:  _BCM tracks mean postsynaptic activity.
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
    Learning rule:  w      y(y -  _BCM)x
      - y >  _BCM: potentiation (strengthen active synapses)
      - y <  _BCM: depression (weaken inactive synapses)

    Prevents runaway Hebbian learning. Essential for stability.
    """

    def __init__(self, cfg: V11Config):
        super().__init__()
        self.enabled = cfg.bcm_enabled
        if not self.enabled:
            return
        self.tau = cfg.bcm_tau
        # Sliding threshold (tracked, not learned)
        self.register_buffer('theta_bcm', torch.tensor(1.0))
        self.register_buffer('_n_updates', torch.tensor(0, dtype=torch.long))

    @torch.no_grad()
    def update_threshold(self, activity: torch.Tensor):
        """Update BCM threshold from mean squared activity."""
        if not self.enabled:
            return
        mean_sq = activity.pow(2).mean()
        if torch.isfinite(mean_sq):
            alpha = 1.0 / self.tau
            self.theta_bcm.lerp_(mean_sq, alpha)
            self._n_updates += 1

    # BCM modulation is applied directly in L23.hebbian_update_bcm()
    # which uses self.theta_bcm as the sliding threshold.


#

# Metaplasticity
#


class Metaplasticity(nn.Module):
    """
    Metaplasticity -- learning to learn (Thm 42).

    Adjusts learning rates based on recent plasticity history.
    High recent plasticity → lower learning rate (stabilize).
    Low recent plasticity → higher learning rate (explore).

    Bienenstock 1982, Abraham & Bear 1996.
    """

    def __init__(self, cfg: V11Config):
        super().__init__()
        self.enabled = cfg.metaplasticity_enabled
        if not self.enabled:
            return
        self.register_buffer('plasticity_ema', torch.tensor(1.0))
        self.tau = 500.0  # Slow timescale

    @torch.no_grad()
    def update(self, weight_change_magnitude: float):
        """Track recent plasticity."""
        if not self.enabled:
            return
        val = torch.tensor(weight_change_magnitude, device=self.plasticity_ema.device)
        if torch.isfinite(val):
            self.plasticity_ema.lerp_(val, 1.0 / self.tau)

    def get_lr_multiplier(self) -> float:
        """Return learning rate multiplier based on recent plasticity."""
        if not self.enabled:
            return 1.0
        # Inverse relationship: high plasticity → low LR
        return 1.0 / (1.0 + self.plasticity_ema.item())


#

# Lateral Column Connections (Horizontal Connections)
#


class LateralColumnConnection(nn.Module):
    """
    Horizontal connections between cortical columns at the same hierarchical level.

    In the brain, columns don't ONLY communicate top-down/bottom-up.
    Dense horizontal connections in L2/3 span 2-3mm, connecting columns
    processing related features. These enable:
      - Feature binding across columns (perceptual grouping)
      - Surround suppression (competition between similar representations)
      - Synchronization (coordinated gamma oscillations)

```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
    Disease mapping:
      Schizophrenia (disconnection syndrome) → disrupted lateral connections
        → inability to bind features → fragmented perception / thought disorder
      Autism → altered lateral connectivity → weak central coherence

    Architecture:
      Each column sends a message to its neighbors (adjacent columns).
      Messages are low-rank projections gated by gamma oscillations.
      Receiving column integrates lateral input via L2/3.

    Identity at init: W_send=0 → no lateral communication initially.
    After training: columns coordinate via learned lateral signals.
    """

    def __init__(self, cfg: V11Config):
        super().__init__()
        d = cfg.d_model
        n = cfg.n_columns
        r = max(d // 8, 32)  # Low-rank bottleneck (efficient)

        # Each column has a send and receive projection
        # Only connect to immediate neighbors (sparse, not fully connected)
        self.W_send = nn.ModuleList([nn.Linear(d, r, bias=False) for _ in range(n)])
        self.W_recv = nn.ModuleList([nn.Linear(r, d, bias=False) for _ in range(n)])

        # Identity at init: no lateral signal
        for w in self.W_send:
            nn.init.zeros_(w.weight)
        for w in self.W_recv:
            nn.init.normal_(w.weight, std=0.01)

        # Per-column gate
        self.gate_raws = nn.ParameterList([
            nn.Parameter(torch.tensor(cfg.gate_init)) for _ in range(n)
        ])

    def forward(self, column_outputs: list) -> List[torch.Tensor]:
        """
        Compute lateral signals between adjacent columns.

        Args:
            column_outputs: list of ColumnOutput from all columns

        Returns:
            lateral_signals: list of [B, T, D] -- lateral modulation per column
        """
        n = len(column_outputs)
        lateral_signals = []

        for i in range(n):
            gate = torch.sigmoid(self.gate_raws[i])

            # Collect messages from neighbors (i-1, i+1)
            neighbor_msgs = []
            if i > 0:
                msg = self.W_send[i - 1](column_outputs[i - 1].output)
                neighbor_msgs.append(msg)
            if i < n - 1:
                msg = self.W_send[i + 1](column_outputs[i + 1].output)
                neighbor_msgs.append(msg)

            if not neighbor_msgs:
                lateral_signals.append(torch.zeros_like(column_outputs[i].output))
                continue

            # Average neighbor messages (could be 1 or 2 neighbors)
            avg_msg = sum(neighbor_msgs) / len(neighbor_msgs)

            # Receive and gate
            lateral = self.W_recv[i](avg_msg)
            lateral_signals.append(gate * lateral)

        return lateral_signals


#

# Full Cortical Column
#


class CorticalColumn(nn.Module):
    """
    A single cortical column -- the fundamental compute unit of NCT V11.
    Replaces a transformer block.

    6 layers with distinct neuroscience-grounded functions:
      L4: Thalamic input (feedforward)
      L1: Top-down modulation (apical context)
      L2/3: Lateral association (windowed Hebbian recall, O(T×w))
      Interneurons: VIP → SST → PV
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
      Working memory: NMDA + Silent + Phono
      Oscillations: Nested PAC
      L5: Output + prediction generation
      L6: Thalamic feedback

    All components are identity-at-init.

    Data flow:
      ff_input → L4 → (+L1 top-down) → L2/3 → Interneurons → WM → Osc → L5 → output
                                                                          → prediction
                                                                  L6 → thal_feedback
    """

    def __init__(self, cfg: V11Config, column_idx: int):
        super().__init__()
        self.cfg = cfg
        self.column_idx = column_idx

        # Core layers
        self.L4 = L4_ThalamicInput(cfg)
        self.L1 = L1_TopDownModulation(cfg)
        self.L23 = L23_LateralAssociation(cfg)
        self.L5 = L5_OutputPrediction(cfg)
        self.L6 = L6_ThalamicFeedback(cfg)

        # Interneurons
        self.interneurons = ColumnInterneurons(cfg)

        # Working memory
        self.wm = ColumnWorkingMemory(cfg)

        # Oscillations
        self.oscillations = ColumnOscillations(cfg)

        # BCM homeostasis (per-column)
        self.bcm = BCMHomeostasis(cfg)

        # Metaplasticity (per-column)
        self.metaplasticity = Metaplasticity(cfg)

        # Structural plasticity (Theorem E)
        if cfg.structural_plasticity:
            self.L23.init_structural_plasticity()

        # Residual projection (if d_column != d_model)
        if cfg.d_column != cfg.d_model:
            self.residual_proj = nn.Linear(cfg.d_model, cfg.d_model, bias=False)
        else:
            self.residual_proj = None

    def forward(
        self,
        ff_input: torch.Tensor,
        td_context: Optional[torch.Tensor] = None,
        sigma2_prev: Optional[torch.Tensor] = None,
    ) -> ColumnOutput:
        """
        Forward pass through the cortical column.

        Biological data flow (corrected interneuron placement):
          L4 → L1 → SST+VIP → L2/3(settle) → WM → Osc → PV → L5 → output

        Args:
          ff_input: [B, T, D_model] -- feedforward from lower column (via thalamus)
          td_context: [B, T, D_model] -- top-down from higher column's prediction
          sigma2_prev: [B, T, 1] -- previous column's prediction precision (for L5)

        No RoPE: L2/3 uses Windowed Hebbian Association (O(T×w), not attention).
        Causality is implicit in the backward-looking window.
        """
        # L4: Thalamic input processing
        l4_out = self.L4(ff_input)  # [B, T, D_col]
        h = l4_out

        # L1: Top-down modulation (if available)
        td_signal = None
        if td_context is not None:
            td_signal = self.L1(td_context)
            h = h + td_signal  # Apical modulation

        # SST + VIP: Dendritic inhibition (BEFORE L2/3 -- apical gate)
        h = self.interneurons.apply_dendritic_inhibition(h, td_signal=td_signal)

        # L2/3: Windowed Hebbian Association (no RoPE, no O(T²) attention)
        l23_input = h
        for _ in range(self.cfg.n_settle_steps):
            l23_out = self.L23(h)
            h = l23_input + l23_out  # Re-anchor to input each step (settling)
        l23_output = h

        # Working memory
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        h = self.wm(h)

        # Oscillatory dynamics
        h = self.oscillations(h)

        # PV/Chandelier: Output veto (AFTER computation, BEFORE L5 -- axon initial segment)
        h = self.interneurons.apply_output_veto(h)

        # L5: Dendritic FFN + prediction (receives L4 bottom-up + L1 top-down +  ²)
        output_delta, prediction, sigma2 = self.L5(
            h, l4_input=l4_out, td_context=td_signal, sigma2_prev=sigma2_prev
        )
        # Store h_norm for analytical PC gradient (accessed by PredictiveCodingOptimizer)
        self._last_l5_h_norm = self.L5.norm(h).detach()

        # Output with residual connection (+ residual_proj if dimensions differ)
        if self.residual_proj is not None:
            output = self.residual_proj(ff_input) + output_delta
        else:
            output = ff_input + output_delta

        # L6: Thalamic feedback
        thal_feedback = self.L6(h)

        return ColumnOutput(
            output=output,
            prediction=prediction,
            thal_feedback=thal_feedback,
            state=h,
            sigma2=sigma2,
            l23_input=l23_input,
            l23_output=l23_output,
        )

    def get_gate_values(self) -> dict:
        """Return all gate values for monitoring."""
        i = self.column_idx
        gates = {
            f'col{i}_l23': torch.sigmoid(self.L23.gate_raw).item(),
            f'col{i}_l1': torch.sigmoid(self.L1.gate_raw).item(),
            f'col{i}_l6': torch.sigmoid(self.L6.gate_raw).item(),
        }
        if self.cfg.vip_enabled:
            gates[f'col{i}_vip'] = torch.sigmoid(self.interneurons.vip_gate_raw).item()
        if self.cfg.sst_enabled:
            gates[f'col{i}_sst'] = torch.sigmoid(self.interneurons.sst_gate_raw).item()
        if self.cfg.pv_enabled:
            gates[f'col{i}_pv'] = torch.sigmoid(self.interneurons.pv_gate_raw).item()
        # WM gates (were missing -- now monitored)
        if self.cfg.wm_enabled:
            gates[f'col{i}_nmda'] = torch.sigmoid(self.wm.nmda_gate_raw).item()
            gates[f'col{i}_silent'] = torch.sigmoid(self.wm.silent_gate_raw).item()
            gates[f'col{i}_phono'] = torch.sigmoid(self.wm.phono_gate_raw).item()
        # Oscillation gate (was missing)
        if self.cfg.oscillations_enabled:
            gates[f'col{i}_osc'] = torch.sigmoid(self.oscillations.gate_raw).item()
        # L5 dendritic gates
        if self.cfg.l5_dendritic:
            # Report mean dendritic gate activation (bias-only at init)
            gates[f'col{i}_dend_bias'] = torch.sigmoid(self.L5.W_dend.bias.mean()).item()
            gates[f'col{i}_apical_bias'] = torch.sigmoid(self.L5.W_apical.bias.mean()).item()
        # Structural plasticity density
        if self.cfg.structural_plasticity:
            gates[f'col{i}_density'] = self.L23.get_connection_density()
        return gates
```

Listing 9: Cortical Column — complete source code (v11_column.py, 1,435 lines)

11 Subcortical Systems

Source: `nct_v2/v11_systems.py` **Lines:** 938 **Classes:** 9

11.1 Neuroscience Foundation

The neocortex does not operate in isolation. Every cortical area has reciprocal connections with subcortical structures that provide routing, memory consolidation, reward computation, temporal prediction, and global state regulation. In the human brain, these subcortical structures contain only ∼15% of all neurons but are essential for cognition — damage to any single structure produces a specific neurological syndrome.

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

NCT implements seven subcortical systems, each validated by its disease-mapping: removing the module produces a computational deficit analogous to the corresponding neurological disorder.

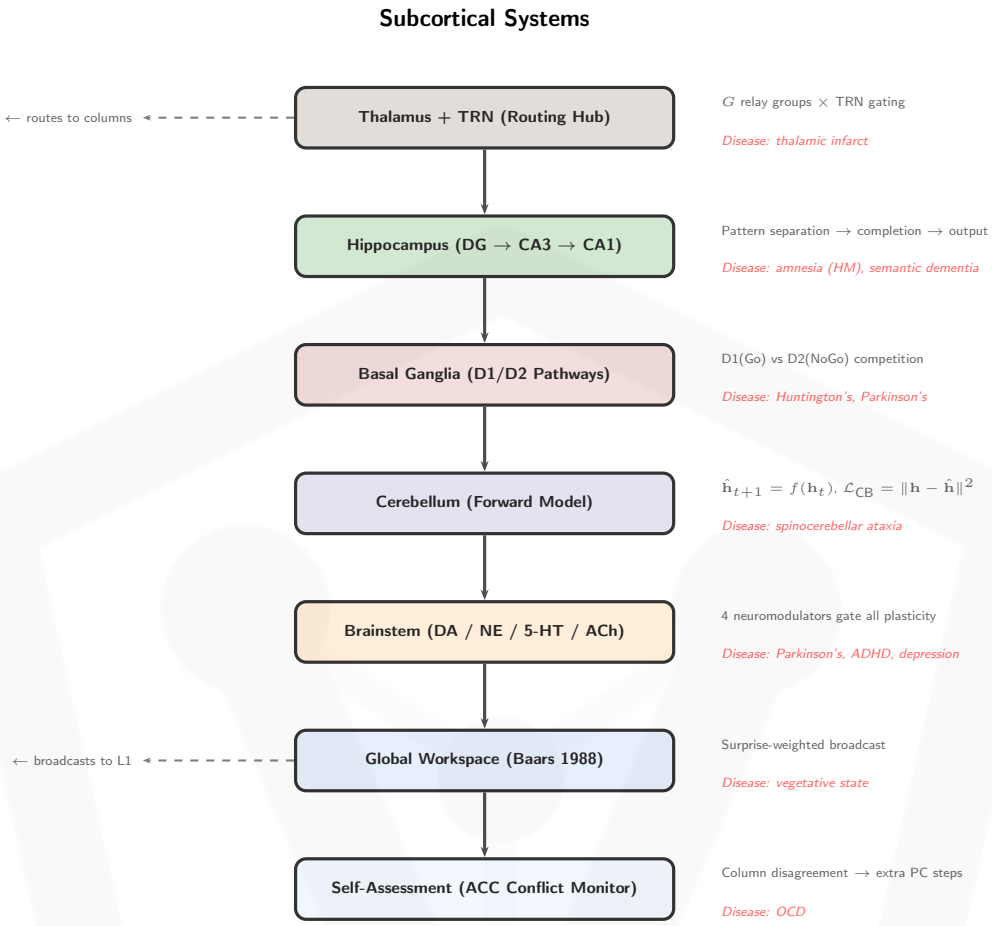


Figure 41: NCT subcortical systems hierarchy. Each system is validated by its disease-mapping: removing the module produces the corresponding neurological deficit.

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

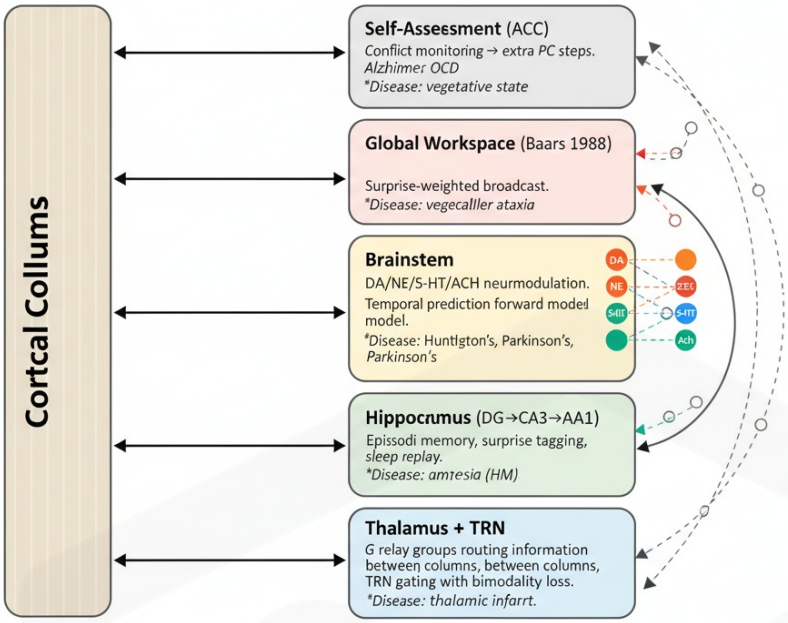


Figure 42: (Illustrated) NCT subcortical systems hierarchy. Each system is validated by its disease-mapping: removing the module produces the corresponding neurological deficit.

11.2 Thalamus + TRN

The thalamus is the brain's routing hub — virtually all sensory information passes through it before reaching cortex. NCT implements G relay groups (default $G = \lceil N/3 \rceil = 4$ for 12 columns), each shared by 3 adjacent columns.

Theorem 11.1 (Thalamic Relay with TRN Gating)**.** *Each relay group computes a gated residual:*

$$\mathbf{x}' = \mathbf{x} + \underbrace{\sigma(g_r)}_{\text{relay gate}} \cdot \underbrace{r_{TRN}}_{\text{reticular gate}} \cdot \underbrace{W_{out} GELU(W_{in} RMSNorm(\mathbf{x}))}_{\text{relay computation}} \tag{47}$$

where $W_{in} \in \mathbb{R}^{d_r \times d}$, $W_{out} \in \mathbb{R}^{d \times d_r}$, and L6 feedback modulates the relay: $+0.1 \cdot \mathbf{f}_{thal}$.

Definition 11.1 (Thalamic Reticular Nucleus (TRN))**.** *The TRN implements low-rank gating ($d_{trn} = 24$):*

$$r_{TRN} = \sigma\big(W_{trn,out} \tanh(W_{trn,in}[\mathbf{x}; relay_delta])\big) \tag{48}$$

A bimodality loss encourages binary (open/closed) routing decisions:

$$\mathcal{L}_{TRN} = \frac{1}{G} \sum_{g=1}^{G} r_g(1 - r_g) \tag{49}$$

This loss is minimized when $r \approx 0$ (gate closed) or $r \approx 1$ (gate open), driving the thalamus toward discrete routing decisions rather than continuous mixing.

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

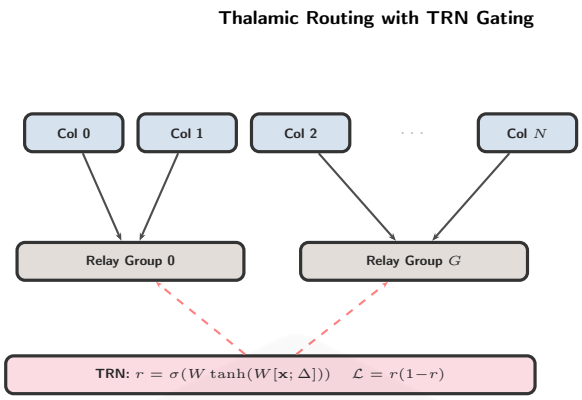


Figure 43: Thalamic routing: G relay groups serve ∼3 columns each. TRN gating enforces binary routing via bimodality loss.

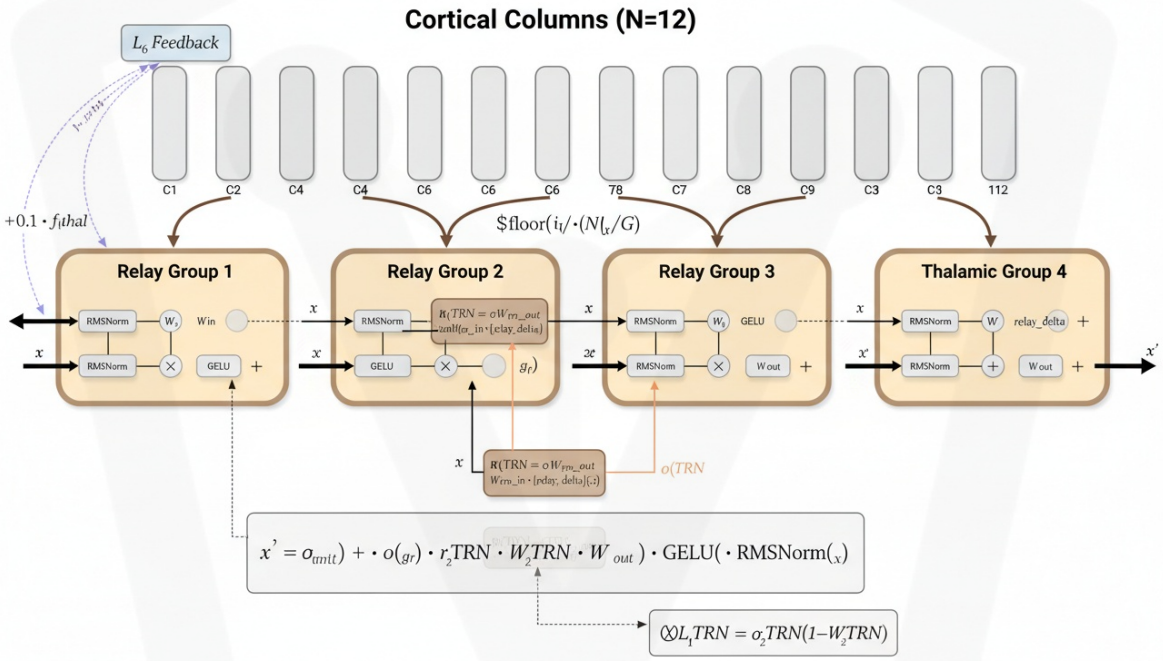


Figure 44: (Illustrated) Thalamic routing: G relay groups serve ∼3 columns each. TRN gating enforces binary routing via bimodality loss.

11.3 Hippocampus

The hippocampus is the brain's episodic memory system, essential for forming new memories and consolidating them into long-term cortical storage. Patient HM, who had bilateral hippocampal removal, could not form new declarative memories — demonstrating the hippocampus's critical role.

Theorem 11.2 (Hippocampal Trisynaptic Circuit)**.** *NCT's hippocampus implements the three-stage circuit:*

1. Dentate Gyrus (DG) — Pattern Separation:

$$\mathbf{h}_{DG} = ReLU(W_{DG}\mathbf{h}), \qquad W_{DG} \in \mathbb{R}^{d_{sparse} \times d}, \quad d_{sparse} = 4d \tag{50}$$

The $4\times$ *expansion ratio with ReLU produces sparse, orthogonal representations that minimize interference between stored memories.*

2. CA3 — Autoassociative Pattern Completion:

$$\mathbf{h}_{CA3} = W_{CA3}\mathbf{h}_{DG}, \qquad W_{CA3} \in \mathbb{R}^{d_{sparse} \times d_{sparse}} \tag{51}$$

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

Given a partial cue, CA3 retrieves the full stored pattern via associative recall.

3. CA1 — Output to Cortex:

$$\mathbf{h}_{out} = W_{CA1}\mathbf{h}_{CA3}, \qquad W_{CA1} \in \mathbb{R}^{d \times d_{sparse}} \tag{52}$$

Definition 11.2 (Surprise Tagging for Replay)**.** *During training, episodes are tagged for sleep replay when the prediction error exceeds the surprise threshold:*

$$tag_episode(\mathbf{h}, \boldsymbol{\epsilon}, \sigma^2) \quad if \quad \frac{\|\boldsymbol{\epsilon}\|}{\sigma^2} > \theta_{surprise} = 0.5 \tag{53}$$

Tagged episodes are stored in a circular buffer (max 2,048 memories) with exponential decay: $\alpha_{decay} = \sigma(4.95) \approx 0.993$*. During sleep, high-surprise episodes are preferentially replayed, strengthening the cortical representations of unexpected events.*

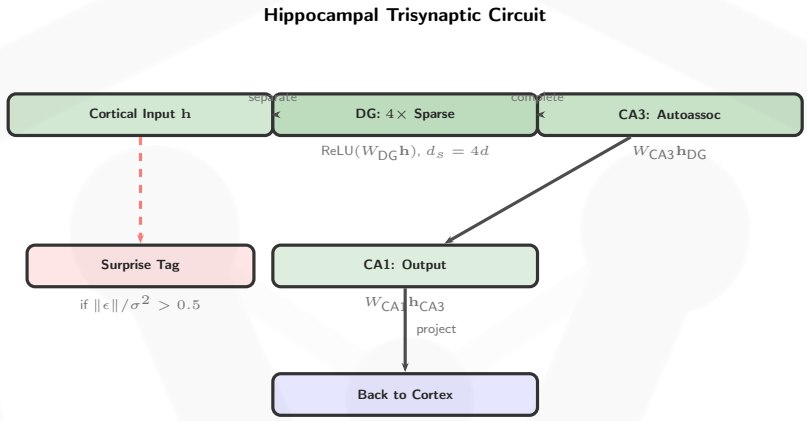


Figure 45: Hippocampal trisynaptic circuit: DG sparse expansion (4×) provides pattern separation, CA3 autoassociation provides pattern completion, CA1 projects back to cortex. Surprise tagging stores episodes for sleep replay.

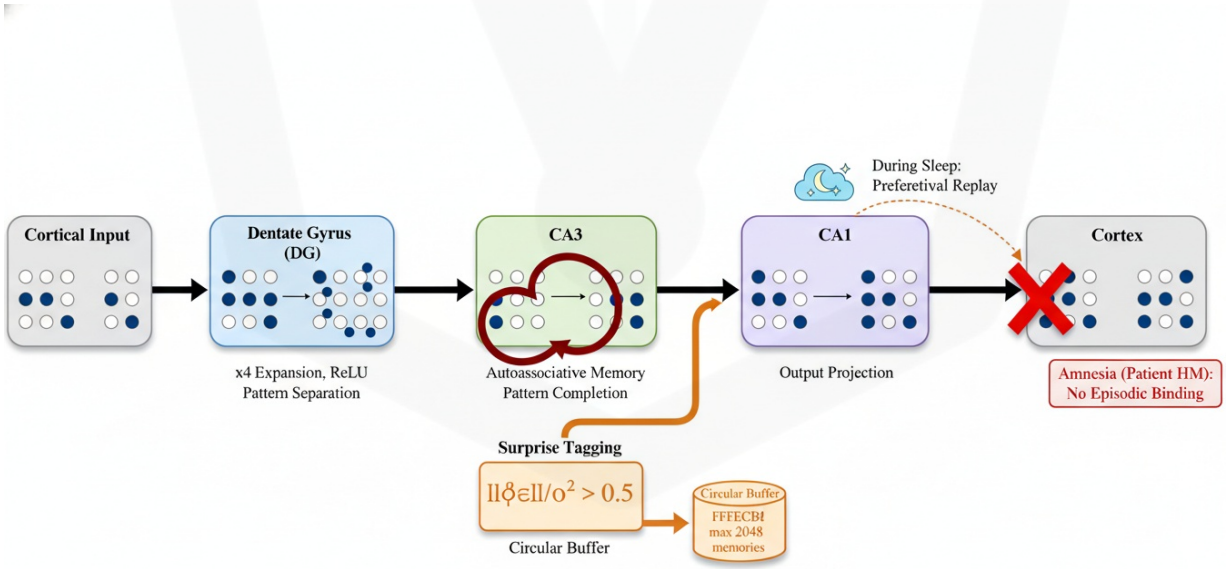


Figure 46: (Illustrated) Hippocampal trisynaptic circuit: DG sparse expansion (4×) provides pattern separation, CA3 autoassociation provides pattern completion, CA1 projects back to cortex. Surprise tagging stores e...

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

11.4 Basal Ganglia

The basal ganglia implement reward-modulated action selection through two competing pathways with pharmacologically distinct dopamine receptor types.

Theorem 11.3 (D1/D2 Inverted-U Dopamine Response)**.** *The direct (D1/Go) and indirect (D2/NoGo) pathways have different dopamine sensitivities:*

$$D1(DA) = \sigma(slope_1 \cdot (DA - EC50_1)), \qquad EC50_1 = 0.65, \quad slope_1 = 8.0 \tag{54}$$

$$D2(DA) = \sigma(slope_2 \cdot (DA - EC50_2)), \qquad EC50_2 = 0.25, \quad slope_2 = 8.0 \tag{55}$$

The output is the competition between pathways:

$$\mathbf{h}_{BG} = \mathbf{h} + \sigma(g_{BG}) \cdot (\underbrace{D1(DA) \cdot W_{go}\mathbf{h}}_{direct/Go} - \underbrace{D2(DA) \cdot W_{nogo}\mathbf{h}}_{indirect/NoGo}) \tag{56}$$

At moderate DA (∼0.5), both pathways are partially active — balanced selection. At high DA (>0.65), D1 dominates — impulsive, Go-biased (mania analog). At low DA (<0.25), D2 dominates — conservative, NoGo-biased (Parkinson's analog). The inverted-U emerges naturally from the different EC50 values.

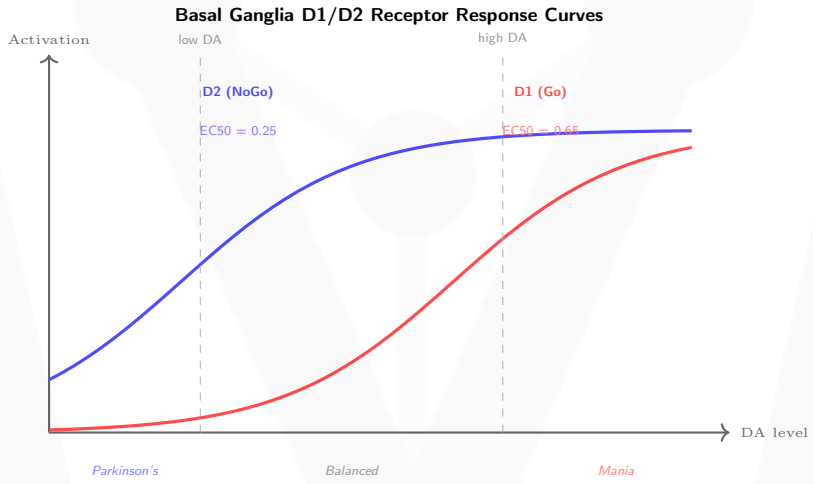


Figure 47: Basal ganglia D1/D2 response curves. D2 (indirect/NoGo) activates at low DA (EC50=0.25), D1 (direct/Go) at high DA (EC50=0.65). Low DA (Parkinson's): conservative. High DA (mania): impulsive. The inverted-U performance curve emerges from the intersection region.

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

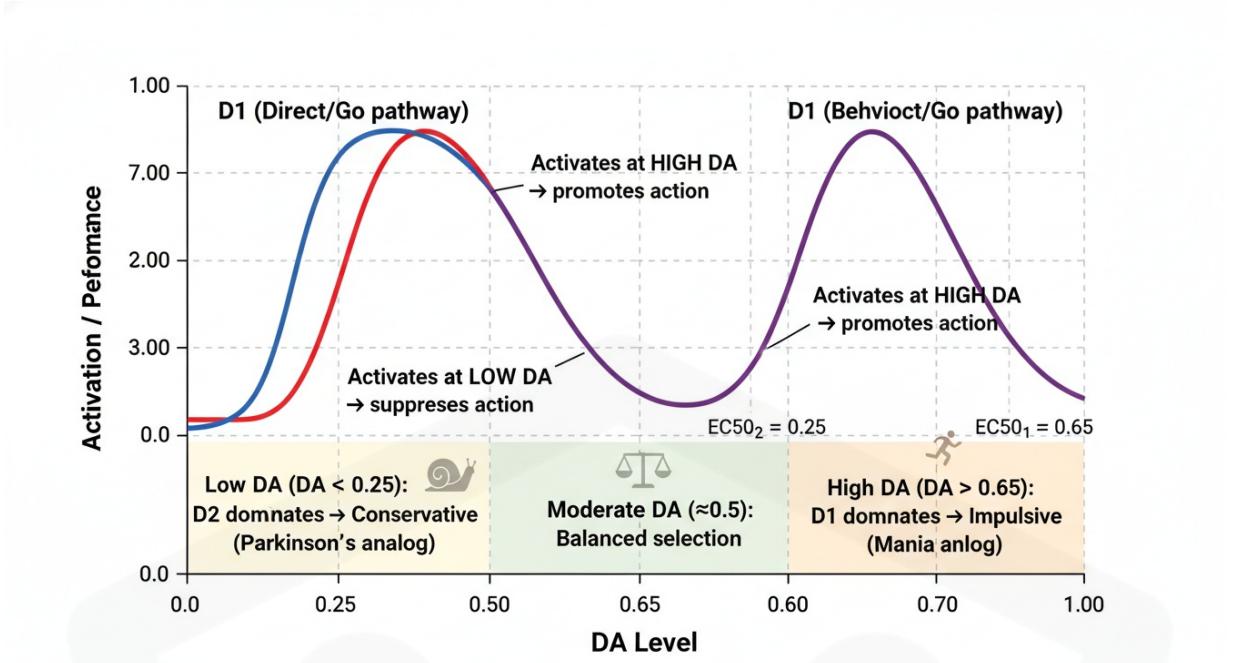


Figure 48: (Illustrated) Basal ganglia D1/D2 response curves. D2 (indirect/NoGo) activates at low DA (EC50=0.25), D1 (direct/Go) at high DA (EC50=0.65). Low DA (Parkinson's): conservative. High DA (mania): impulsive. The i...

11.5 Cerebellum

Theorem 11.4 (Cerebellar Forward Model)**.** *The cerebellum learns to predict the next hidden state:*

$$\hat{\mathbf{h}}_{t+1} = W_{cb,out} GELU(W_{cb,in} RMSNorm(\mathbf{h}_t)) \tag{57}$$

Prediction error provides an auxiliary training signal:

$$\mathcal{L}_{CB} = \|\mathbf{h}_t - \hat{\mathbf{h}}_t\|^2 \tag{58}$$

This enables the system to anticipate upcoming processing demands and pre-activate relevant representations.

11.6 Brainstem — Neuromodulation (Theorem C)

Theorem 11.5 (Four Neuromodulatory Systems)**.** *The brainstem computes four global modulatory signals from the top column's output:*

$$DA = \sigma(W_{da} RMSNorm(\mathbf{h}_{top})) \quad \textit{(dopamine: reward prediction error)} \tag{59}$$

$$NE = \sigma(W_{ne} RMSNorm(\mathbf{h}_{top})) \quad \textit{(norepinephrine: arousal/exploration)} \tag{60}$$

$$5\text{-}HT = \sigma(W_{5ht} RMSNorm(\mathbf{h}_{top})) \quad \textit{(serotonin: time horizon/patience)} \tag{61}$$

$$ACh = \sigma(W_{ach} RMSNorm(\mathbf{h}_{top})) \quad \textit{(acetylcholine: attention/plasticity gate)} \tag{62}$$

These multiplicatively gate all local learning (Theorem C):

$$\Delta\theta = DA \cdot ACh \cdot f(NE, 5\text{-}HT) \cdot \Delta\theta_{local} \tag{63}$$

Tonic DA is tracked as an EMA ($\alpha = 0.999$) for baseline subtraction, implementing the difference between phasic (event-related) and tonic (background) dopamine signaling.

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

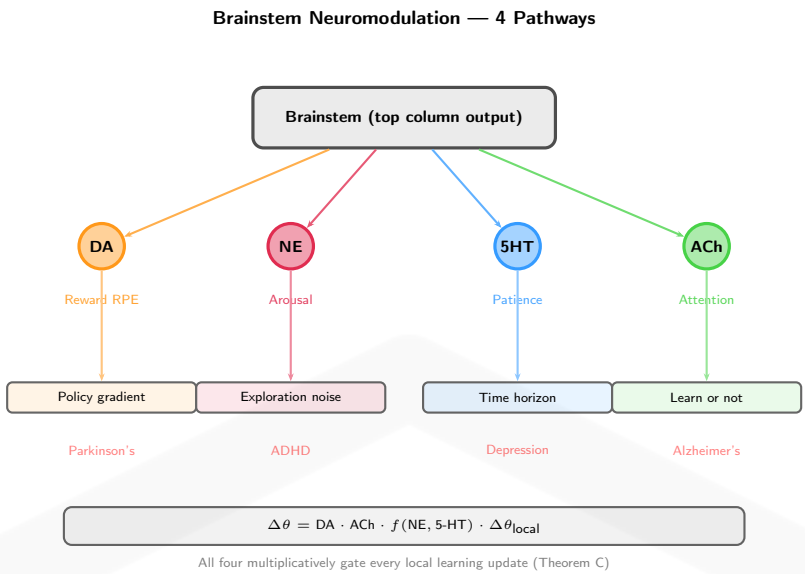


Figure 49: Four neuromodulatory pathways from the brainstem. Each modulator is computed from the top column's output via a learned sigmoid projection. Together, they multiplicatively gate all local learning: DA provides reward signal, ACh gates whether learning occurs, NE controls exploration magnitude, 5-HT modulates time horizon. Each maps to a specific neurological disorder when disrupted.

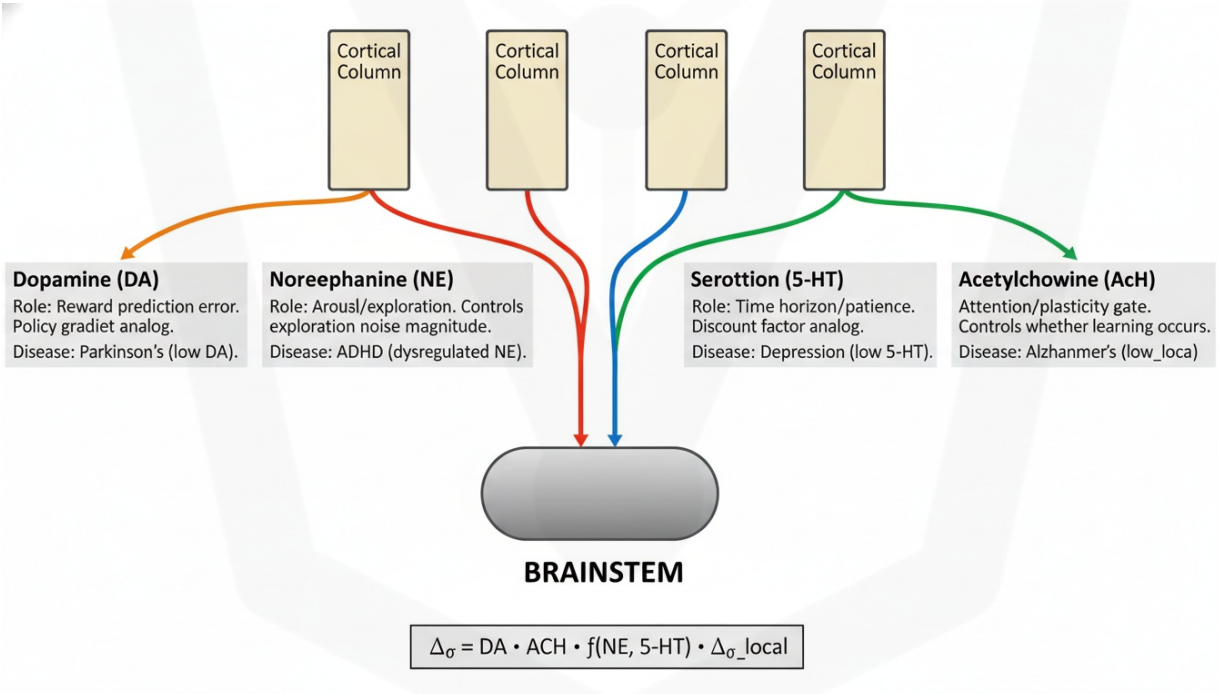


Figure 50: (Illustrated) Four neuromodulatory pathways from the brainstem. Each modulator is computed from the top column's output via a learned sigmoid projection. Together, they multiplicatively gate all local learning: ...

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

Table 9: Neuromodulatory systems and their computational roles.

System	Signal	Computational Role	Disease if Disrupted
Dopamine (DA)	Reward prediction error	Policy gradient, Go/NoGo	Parkinson's (low), mania (high)
Norepinephrine (NE)	Arousal level	Exploration noise, gain control	ADHD (dysregulated)
Serotonin (5-HT)	Time horizon	Discount factor, patience	Depression (low)
Acetylcholine (ACh)	Attentional state	Plasticity gate (learn or not)	Alzheimer's (low)

11.7 Global Workspace (Baars 1988)

Theorem 11.6 (Surprise-Weighted Broadcast)**.** *Following Global Workspace Theory (Baars 1988, Dehaene & Naccache 2001), columns compete for broadcast access weighted by their prediction error (surprise):*

$$w_i = \frac{\exp(\|\boldsymbol{\epsilon}_i\|/\tau)}{\sum_{j=0}^{N-1} \exp(\|\boldsymbol{\epsilon}_j\|/\tau)}, \qquad \mathbf{b} = \sum_{i=0}^{N-1} w_i \cdot W_{write,i} \cdot \mathbf{h}_i \tag{64}$$

Each column reads the broadcast via its own projection:

$$\mathbf{c}_i^{ws} = W_{read,i} \cdot \mathbf{b} \tag{65}$$

The broadcast is added to L1 top-down context on the next PC iteration, enabling cross-column coordination beyond local lateral connections. Bottleneck dimension: $d_{ws} = d/4$.

$W_{\text{write},i}$ initialized at $\mathbf{0}$ — no broadcast at initialization. The highest-surprise column "wins" access to the workspace, broadcasting its state to all other columns. This is the computational analog of conscious access: only information that is sufficiently surprising reaches global broadcast.

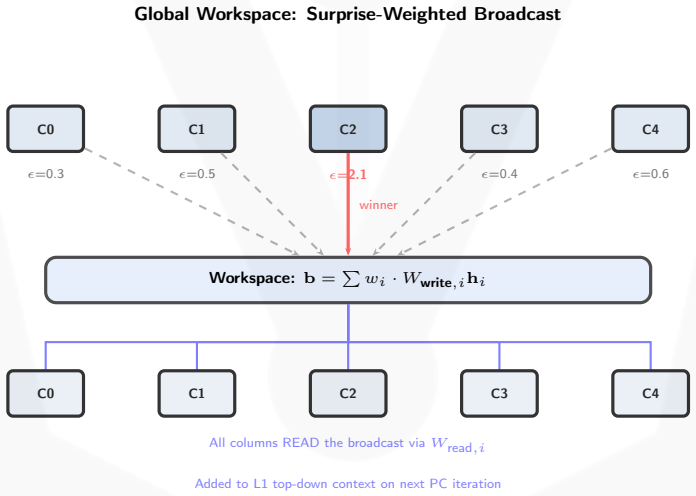


Figure 51: Global workspace broadcast: columns compete for broadcast access weighted by surprise ($\|\boldsymbol{\epsilon}\|$). Column 2 has the highest prediction error ($\epsilon = 2.1$) and "wins" — its state is broadcast to all columns via the workspace. Other columns read this broadcast as L1 top-down context, enabling cross-column coordination beyond local lateral connections.

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

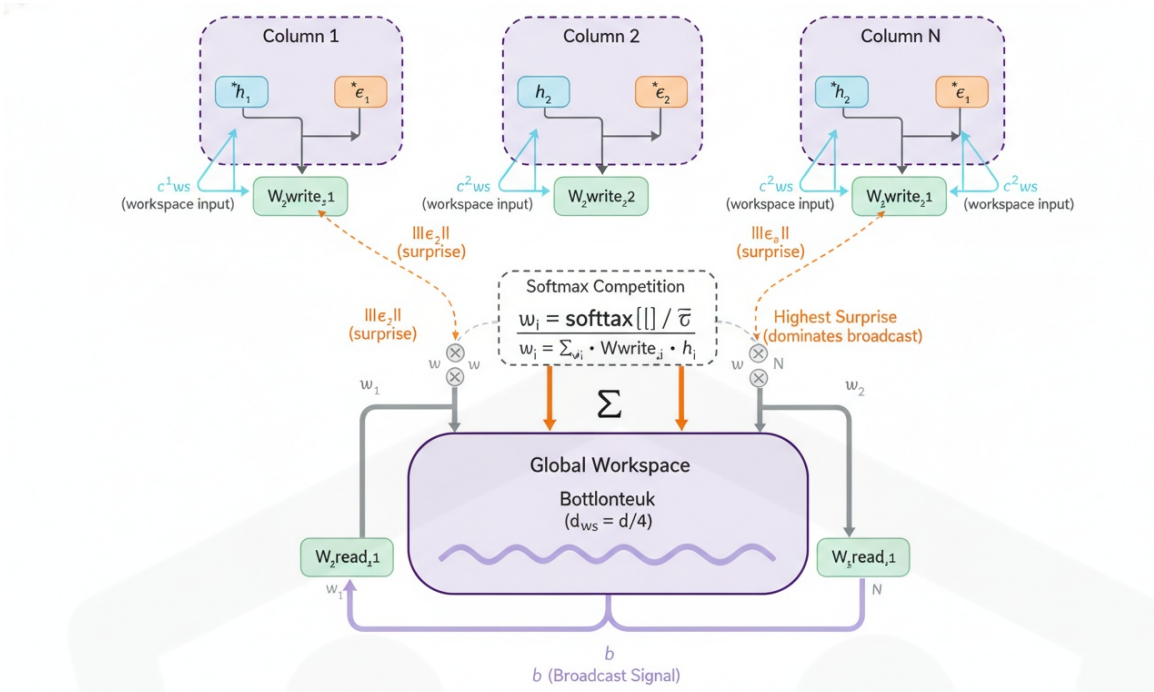


Figure 52: (Illustrated) Global workspace broadcast: columns compete for broadcast access weighted by surprise ($\|\boldsymbol{\epsilon}\|$). Column 2 has the highest prediction error ($\epsilon = 2.1$) and "wins" — it...

11.8 Self-Assessment (ACC Conflict Monitor)

Definition 11.3 (Metacognitive Conflict Detection)**.** *The anterior cingulate cortex (ACC) monitors for conflict between competing representations:*

$$conflict = Var(\{\mathbf{h}_0, \mathbf{h}_1, \ldots, \mathbf{h}_{N-1}\}) \tag{66}$$

High conflict triggers additional PC inference steps (up to +2 extra, capped), implementing metacognitive "thinking harder when uncertain." This is the computational basis for the feeling of effort during difficult cognitive tasks.

The Self-Assessment module also tracks model health: loss history, divergence detection ($> 2\times$ best loss triggers rollback), and performance trends. This enables self-repair — a form of metacognitive monitoring absent in transformers.

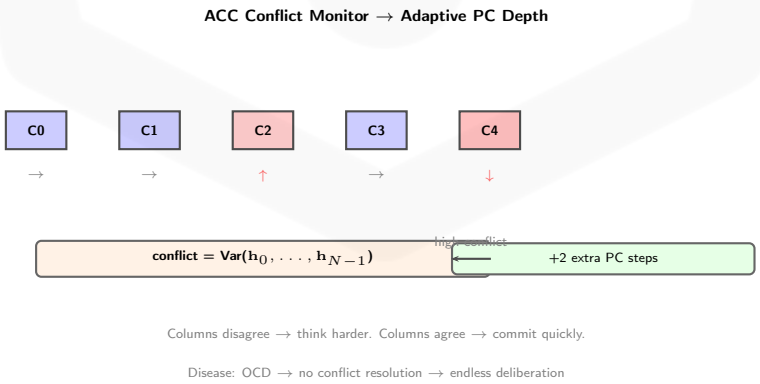


Figure 53: ACC conflict monitor: computes variance of column outputs. High variance (columns disagree) triggers up to 2 extra PC inference steps — the model "thinks harder" when uncertain. Low variance (consensus) allows early commitment. This is metacognitive self-regulation.

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

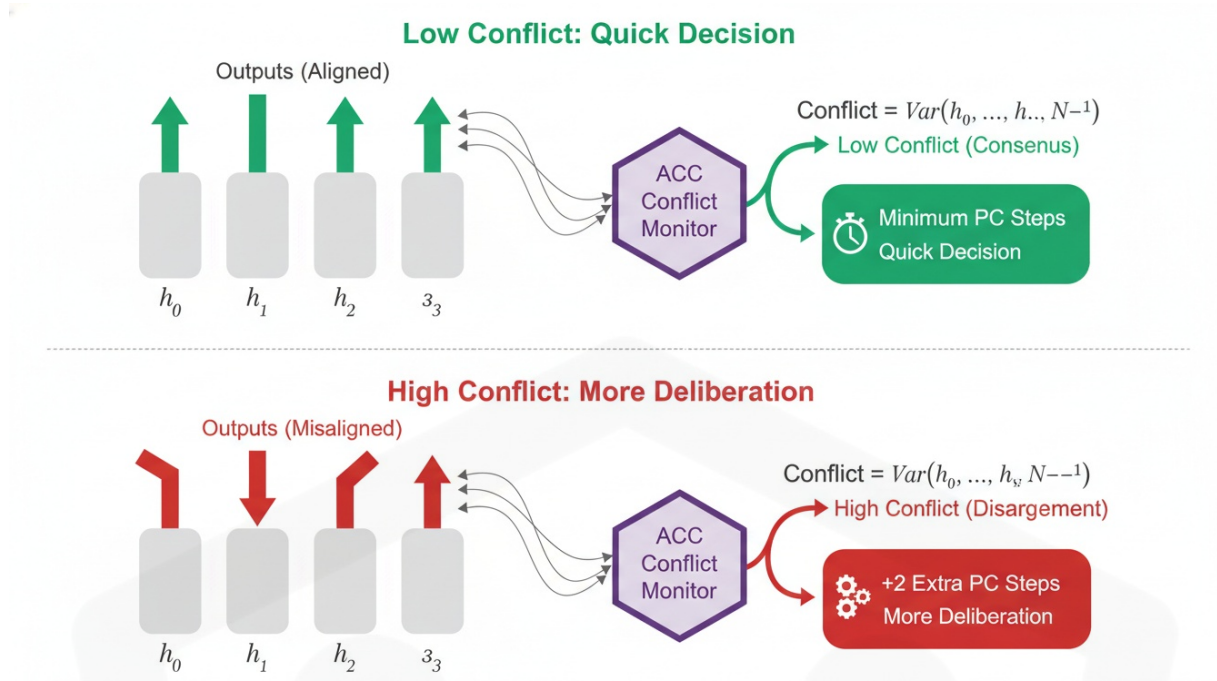


Figure 54: (Illustrated) ACC conflict monitor: computes variance of column outputs. High variance (columns disagree) triggers up to 2 extra PC inference steps — the model "thinks harder" when uncertain. Low variance (c...

11.9 Source Code

```
1
2  # ======================================================================
3  # Subcortical Systems (Thalamus, HPC, BG, Cerebellum, Brainstem)
4  # Source: nct_v2/v11_systems.py
5  # ======================================================================
6
7  """
8  NCT V11: Subcortical Systems
9  ============================
10
11 The cortex doesn't operate alone. These subcortical systems provide:
12   - Thalamus: Routing, gating, synchronization between columns
13   - Hippocampus: Episodic memory, surprise tagging, sleep replay
14   - Basal Ganglia: DA-based reward learning, action selection
15   - Cerebellum: Temporal prediction, forward models
16   - Brainstem: NE/5-HT/ACh global state regulation
17
18 All systems interact with cortical columns via defined interfaces.
19 """
20
21
22
23 # [bundled] from .v11_config import V11Config
24 # [bundled] from .v11_column import RMSNorm
25
26
27 #
28 # Neuromodulatory State
29 #
30
31 @dataclass
32 class NeuromodState:
33     """Global neuromodulatory state broadcast to all columns."""
34     dopamine: torch.Tensor       # [B, T, 1] -- prediction error / reward
35     norepinephrine: torch.Tensor  # [B, T, 1] -- arousal / exploration
36     serotonin: torch.Tensor      # [B, T, 1] -- time horizon / patience
37     acetylcholine: torch.Tensor  # [B, T, 1] -- attention / plasticity gate
38     tonic_da: float              # Scalar -- slow DA average (exploration/exploitation)
39
40
41 #
42 # Thalamus (Routing Hub)
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```python
#


class ThalamicRelay(nn.Module):
    """
    A single thalamic relay group.
    Gates and routes information between columns.

    Relay = W_out(gelu(W_in(x))) gated by reticular nucleus.
    """

    def __init__(self, cfg: V11Config):
        super().__init__()
        self.W_in = nn.Linear(cfg.d_model, cfg.d_relay, bias=False)
        self.W_out = nn.Linear(cfg.d_relay, cfg.d_model, bias=False)
        nn.init.zeros_(self.W_out.weight)  # Identity at init
        self.norm = RMSNorm(cfg.d_model)
        self.gate_raw = nn.Parameter(torch.tensor(cfg.gate_init))

    # forward() is not used -- Thalamus.route() drives relays directly
    # to support L6 feedback injection and TRN loss accumulation.


class ThalamicReticularNucleus(nn.Module):
    """
    Thalamic Reticular Nucleus (TRN).
    Thin shell around thalamus that gates relay neurons.

    Computes relevance between pre-relay and post-relay signals.
    TRN loss: bimodality loss r*(1-r) for binary decisions.
    """

    def __init__(self, cfg: V11Config):
        super().__init__()
        self.W_q = nn.Linear(cfg.d_model, cfg.d_trn, bias=False)
        self.W_k = nn.Linear(cfg.d_model, cfg.d_trn, bias=False)
        self.scale = cfg.d_trn ** -0.5
        self.gate_raw = nn.Parameter(torch.tensor(cfg.gate_init))

    def forward(self, x_pre: torch.Tensor, x_delta: torch.Tensor) -> torch.Tensor:
        """
        x_pre: [B, T, D_model] -- before relay
        x_delta: [B, T, D_model] -- relay output delta
        Returns: [B, T, 1] -- relevance gate (0=block, 1=pass)
        """
        gate = torch.sigmoid(self.gate_raw)
        q = self.W_q(x_pre)  # [B, T, d_trn]
        k = self.W_k(x_delta)
        # Per-token relevance
        relevance = torch.sigmoid(
            (q * k).sum(dim=-1, keepdim=True) * self.scale
        )  # [B, T, 1]
        # Lerp with gate: at init, relevance → 1 (pass everything)
        return (1.0 - gate) * 1.0 + gate * relevance

    # TRN bimodality loss is accumulated in Thalamus.route() → Thalamus.compute_loss().


class Thalamus(nn.Module):
    """
    Complete thalamic system: relay groups + TRN.
    Controls information flow between cortical columns.
    """

    def __init__(self, cfg: V11Config):
        super().__init__()
        self.cfg = cfg
        self.enabled = cfg.thalamic_enabled
        if not self.enabled:
            return

        self.n_groups = cfg.n_relay_groups
        self.relays = nn.ModuleList([ThalamicRelay(cfg) for _ in range(self.n_groups)])
        self.trn = ThalamicReticularNucleus(cfg)

        # Accumulated TRN loss during forward pass
        self._trn_loss_accum: List[torch.Tensor] = []

    def reset_loss(self):
        """Reset TRN loss accumulator for a new forward pass."""
        self._trn_loss_accum = []

    def route(self, x: torch.Tensor, column_idx: int,
              thal_feedback: Optional[torch.Tensor] = None) -> torch.Tensor:
        """
        Route input through thalamic relay for a specific column.

        x: [B, T, D_model] -- input to be relayed
        column_idx: which column is receiving
        thal_feedback: [B, T, D_model] -- L6 feedback from the column (modulates relay)
        """
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        if not self.enabled:
            return x

        group_idx = min(column_idx // self.cfg.columns_per_relay, self.n_groups - 1)
        relay = self.relays[group_idx]

        # Compute relay delta
        relay_delta = relay.W_out(F.gelu(relay.W_in(relay.norm(x))))

        # Modulate with L6 feedback (if available)
        if thal_feedback is not None:
            relay_delta = relay_delta + 0.1 * thal_feedback

        # TRN gating
        ret_gate = self.trn(x, relay_delta)

        # Accumulate TRN bimodality loss: r*(1-r) encourages binary gating
        trn_loss = (ret_gate * (1.0 - ret_gate)).mean()
        self._trn_loss_accum.append(trn_loss)

        # Apply gated relay
        gate = torch.sigmoid(relay.gate_raw)
        return x + gate * ret_gate * relay_delta

    def compute_loss(self) -> torch.Tensor:
        """TRN bimodality loss accumulated during route() calls."""
        if not self.enabled or len(self._trn_loss_accum) == 0:
            return torch.tensor(0.0)
        loss = sum(self._trn_loss_accum) / len(self._trn_loss_accum)
        self._trn_loss_accum = []  # Reset for next forward pass
        return loss


#

# Hippocampus (Episodic Memory + Replay)
#


class Hippocampus(nn.Module):
    """
    Hippocampal system for episodic memory, surprise tagging, and sleep replay.

    Components:
      Dentate Gyrus (DG): Sparse expansion for pattern separation
      CA3: Autoassociative memory for pattern completion
      CA1: Output → cortex (relayed via subiculum)

    Functions:
      1. Encode: Tag high-surprise events during wake phase
      2. Replay: Sample stored memories during sleep phase
      3. Consolidate: Transfer episodic → semantic (cortical) memory

    Theorem D: Wake-Sleep Cycle = Autonomous Self-Improvement
    """

    def __init__(self, cfg: V11Config):
        super().__init__()
        self.cfg = cfg
        self.enabled = cfg.hpc_enabled
        if not self.enabled:
            return

        d = cfg.d_model
        d_sparse = cfg.hpc_d_sparse

        # Dentate Gyrus: sparse expansion (pattern separation)
        self.dg = nn.Linear(d, d_sparse, bias=False)
        # CA3: autoassociative (pattern completion)
        self.ca3_encode = nn.Linear(d_sparse, d, bias=False)
        self.ca3_retrieve = nn.Linear(d, d_sparse, bias=False)
        # CA1: output projection
        self.ca1 = nn.Linear(d, d, bias=False)
        nn.init.zeros_(self.ca1.weight)

        # Write gate
        self.write_gate = nn.Linear(d + 1, 1, bias=True)  # +1 for sigma2
        nn.init.constant_(self.write_gate.bias, 1.0)  # Start open ( (1)~0.73), learn to be selective

        # Decay (ultra-slow)
        self.decay_bias = nn.Parameter(torch.tensor(cfg.hpc_decay_init))

        # Normalization
        self.norm = RMSNorm(d)

        self.gate_raw = nn.Parameter(torch.tensor(cfg.gate_init))

        # Episodic memory buffer (not a parameter, managed manually)
        self._memory_buffer: List[Dict[str, torch.Tensor]] = []
        self._surprise_buffer: List[float] = []

```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```python
    def forward(self, x: torch.Tensor, sigma2: torch.Tensor) -> torch.Tensor:
        """
        Forward pass: DG → CA3 encode → CA3 retrieve (pattern completion) → CA1.

        x: [B, T, D_model] -- current representation
        sigma2: [B, T, 1] -- precision (used for write gating)
        Returns: [B, T, D_model] -- x with hippocampal contribution
        """
        if not self.enabled:
            return x

        gate = torch.sigmoid(self.gate_raw)
        h = self.norm(x)

        # Sparse encoding via Dentate Gyrus (pattern separation)
        sparse = F.relu(self.dg(h))  # [B, T, d_sparse]

        # CA3 encode: sparse → dense
        encoded = self.ca3_encode(sparse)  # [B, T, D]

        # CA3 retrieve: dense → sparse (pattern completion loop)
        # Apply decay to retrieval (decay_bias controls memory persistence)
        decay = torch.sigmoid(self.decay_bias)  # Ultra-slow decay
        retrieved_sparse = F.relu(self.ca3_retrieve(encoded)) * decay  # [B, T, d_sparse]

        # Re-encode completed pattern
        completed = self.ca3_encode(retrieved_sparse)  # [B, T, D]

        # Output via CA1 (uses completed pattern, not raw encoding)
        hpc_out = self.ca1(completed)

        return x + gate * hpc_out

    @torch.no_grad()
    def encode_episode(self, x: torch.Tensor, prediction_error: torch.Tensor,
                       sigma2: torch.Tensor):
        """
        Tag and store surprising events during wake phase.
        Write-gated: adaptive write gate modulates storage probability.

        x: [B, T, D_model] -- representations to remember
        prediction_error: [B, T, D_model] -- prediction errors
        sigma2: [B, T, 1] -- precision
        """
        if not self.enabled:
            return

        # Surprise = precision-weighted prediction error magnitude
        surprise = (prediction_error.pow(2) / (sigma2 + 1e-6)).mean(dim=-1)  # [B, T]
        mean_surprise = surprise.mean().item()

        # Adaptive write gate:  ²-modulated storage probability
        write_input = torch.cat([x.mean(dim=1), sigma2.mean(dim=1)], dim=-1)  # [B, D+1]
        write_prob = torch.sigmoid(self.write_gate(write_input)).mean().item()  # scalar

        # Only store if surprising enough AND write gate permits
        if mean_surprise * write_prob > self.cfg.hpc_surprise_threshold:
            # Mean-pool over batch dim → [1, T, D] so memories are batch-size-agnostic.
            # Prevents shape mismatches when batch size changes between
            # encode (wake) and replay (sleep/dream/eval).
            self._memory_buffer.append({
                'x': x.detach().mean(dim=0, keepdim=True).cpu(),
                'error': prediction_error.detach().mean(dim=0, keepdim=True).cpu(),
                'sigma2': sigma2.detach().mean(dim=0, keepdim=True).cpu(),
            })
            self._surprise_buffer.append(mean_surprise)

            # Evict oldest if buffer full
            if len(self._memory_buffer) > self.cfg.hpc_memory_size:
                self._memory_buffer.pop(0)
                self._surprise_buffer.pop(0)

    def replay(self, n_samples: int, device: torch.device) -> List[Dict[str, torch.Tensor]]:
        """
        Sample from episodic memory for sleep replay.
        Priority sampling: higher surprise → more likely replayed.

        Returns: List of (x, error, sigma2) tuples on the specified device.
        """
        if not self.enabled or len(self._memory_buffer) == 0:
            return []

        n_samples = min(n_samples, len(self._memory_buffer))

        if self.cfg.replay_priority == 'surprise':
            # Priority sampling by surprise
            surprises = torch.tensor(self._surprise_buffer)
            probs = F.softmax(surprises, dim=0)
            indices = torch.multinomial(probs, n_samples, replacement=True)
        elif self.cfg.replay_priority == 'recency':
            # Recency bias
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
            weights = torch.arange(len(self._memory_buffer), dtype=torch.float)
            probs = F.softmax(weights, dim=0)
            indices = torch.multinomial(probs, n_samples, replacement=True)
        else:
            # Uniform
            indices = torch.randint(0, len(self._memory_buffer), (n_samples,))

        results = []
        for idx in indices:
            mem = self._memory_buffer[idx.item()]
            results.append({
                'x': mem['x'].to(device),
                'error': mem['error'].to(device),
                'sigma2': mem['sigma2'].to(device),
            })
        return results

    def clear_memory(self):
        """Clear episodic buffer (e.g., after consolidation)."""
        self._memory_buffer.clear()
        self._surprise_buffer.clear()

    @torch.no_grad()
    def consolidate(self, n_clusters: int = 8, min_cluster: int = 3) -> List[Dict[str, torch.Tensor]]:
        """
        Episodic → Semantic consolidation (McClelland et al. 1995).

        Complementary Learning Systems theory: hippocampus stores episodes fast,
        cortex learns slowly from patterns across episodes. Consolidation extracts
        the pattern (centroid) from clusters of similar memories, returns centroids
        for cortical training, then removes the consolidated episodes.

        This is what happens during deep sleep (SWS stage 3-4).

        Returns: list of centroid patterns (semantic memories) for cortical training.
        """
        if not self.enabled or len(self._memory_buffer) < n_clusters:
            return []

        # Stack all memories
        all_x = torch.stack([m['x'].mean(dim=(0, 1)) for m in self._memory_buffer])  # [N, D]

        # Simple k-means clustering (no sklearn dependency)
        N, D = all_x.shape
        k = min(n_clusters, N // min_cluster)
        if k < 2:
            return []

        # Initialize centroids via k-means++
        centroids = [all_x[torch.randint(N, (1,)).item()]]
        for _ in range(k - 1):
            dists = torch.stack([((all_x - c.unsqueeze(0)).pow(2)).sum(-1) for c in centroids])
            min_dists = dists.min(dim=0).values
            probs = min_dists / (min_dists.sum() + 1e-8)
            idx = torch.multinomial(probs, 1).item()
            centroids.append(all_x[idx])
        centroids = torch.stack(centroids)  # [k, D]

        # Run k-means iterations
        for _ in range(10):
            # Assign
            dists = torch.cdist(all_x, centroids)  # [N, k]
            assignments = dists.argmin(dim=-1)  # [N]
            # Update
            new_centroids = torch.zeros_like(centroids)
            for c in range(k):
                members = all_x[assignments == c]
                if len(members) > 0:
                    new_centroids[c] = members.mean(dim=0)
                else:
                    new_centroids[c] = centroids[c]
            centroids = new_centroids

        # Extract centroids from clusters with enough members
        semantic_memories = []
        consolidated_indices = set()
        for c in range(k):
            member_mask = assignments == c
            n_members = member_mask.sum().item()
            if n_members >= min_cluster:
                # This cluster is stable enough to extract a pattern
                centroid = centroids[c]
                # Store as a "semantic memory" (pattern without episode details)
                device = self._memory_buffer[0]['x'].device if self._memory_buffer else 'cpu'
                semantic_memories.append({
                    'pattern': centroid,
                    'n_episodes': n_members,
                    'mean_surprise': sum(
                        self._surprise_buffer[i] for i in range(N) if member_mask[i]
                    ) / max(n_members, 1),
                })
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
                # Mark episodes for removal
                for i in range(N):
                    if member_mask[i]:
                        consolidated_indices.add(i)

        # Remove consolidated episodes (keep unconsolidated ones)
        new_buffer = []
        new_surprise = []
        for i in range(len(self._memory_buffer)):
            if i not in consolidated_indices:
                new_buffer.append(self._memory_buffer[i])
                new_surprise.append(self._surprise_buffer[i])
        self._memory_buffer = new_buffer
        self._surprise_buffer = new_surprise

        return semantic_memories

    @property
    def memory_size(self) -> int:
        return len(self._memory_buffer)


#

# Basal Ganglia (Action Selection + DA Reward)
#


class BasalGanglia(nn.Module):
    """
    Basal Ganglia: DA-based reward learning and action selection.

    Dual-pathway D1/D2 competition (Theorem C):
      D1 (direct): High DA threshold → stabilize current representations
      D2 (indirect): Low DA threshold → enable flexibility/switching

    Boltzmann receptor kinetics:  (slope * (DA - EC50))
    The D1/D2 balance creates an emergent inverted-U for WM performance.

    DA RPE (reward prediction error) is Phase 2 (RL integration).
    Currently DA comes from Brainstem (prediction error → phasic DA).
    """

    def __init__(self, cfg: V11Config):
        super().__init__()
        self.cfg = cfg
        self.enabled = cfg.bg_enabled
        if not self.enabled:
            return

        d = cfg.d_model
        d_bg = cfg.d_bg

        # D1 pathway (direct, stabilizing)
        self.d1_in = nn.Linear(d, d_bg, bias=False)
        self.d1_out = nn.Linear(d_bg, d, bias=False)
        nn.init.zeros_(self.d1_out.weight)

        # D2 pathway (indirect, flexible)
        self.d2_in = nn.Linear(d, d_bg, bias=False)
        self.d2_out = nn.Linear(d_bg, d, bias=False)
        nn.init.zeros_(self.d2_out.weight)

        # Receptor parameters
        self.d1_ec50 = nn.Parameter(torch.tensor(cfg.d1_ec50))
        self.d1_slope = nn.Parameter(torch.tensor(cfg.d1_slope))
        self.d2_ec50 = nn.Parameter(torch.tensor(cfg.d2_ec50))
        self.d2_slope = nn.Parameter(torch.tensor(cfg.d2_slope))

        # Gate
        self.gate_raw = nn.Parameter(torch.tensor(cfg.gate_init))
        self.norm = RMSNorm(d)

        # Tonic DA EMA
        self.register_buffer('tonic_da', torch.tensor(0.5))

    def forward(self, x: torch.Tensor, da: torch.Tensor) -> torch.Tensor:
        """
        x: [B, T, D_model]
        da: [B, T, 1] -- phasic dopamine (prediction error)
        Returns: [B, T, D_model] -- D1/D2 modulated representation
        """
        if not self.enabled:
            return x

        gate = torch.sigmoid(self.gate_raw)
        h = self.norm(x)

        # D1 receptor activation (high threshold, stabilizing)
        d1_act = torch.sigmoid(self.d1_slope * (da - self.d1_ec50))
        d1_mod = self.d1_out(F.gelu(self.d1_in(h))) * d1_act
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
493
494         # D2 receptor activation (low threshold, flexible)
495         d2_act = torch.sigmoid(self.d2_slope * (da - self.d2_ec50))
496         d2_mod = self.d2_out(F.gelu(self.d2_in(h))) * d2_act
497
498         # D1 - D2 competition (direct - indirect pathways)
499         modulation = d1_mod - d2_mod
500
501         return x + gate * modulation
502
503     @torch.no_grad()
504     def update_tonic_da(self, phasic_da: torch.Tensor):
505         """Update tonic DA EMA."""
506         if not self.enabled:
507             return
508         mean_da = phasic_da.mean()
509         if torch.isfinite(mean_da):
510             self.tonic_da.lerp_(mean_da, 1.0 - self.cfg.tonic_da_ema)
511
512     # RPE + learned value estimation is Phase 2 (RL integration).
513
514
515 #
516 # Cerebellum (Temporal Prediction)
517 #
518
519 class Cerebellum(nn.Module):
520     """
521     Cerebellum: Temporal prediction and error correction.
522
523     Forward model: predicts X[t+1] from X[t].
524     When prediction fails, applies correction signal.
525
526     The cerebellum is the brain's "error correction chip" -- it learns
527     precise temporal patterns and corrects deviations in real-time.
528     """
529
530     def __init__(self, cfg: V11Config):
531         super().__init__()
532         self.enabled = cfg.cerebellum_enabled
533         if not self.enabled:
534             return
535
536         d = cfg.d_model
537         d_cb = cfg.d_cerebellar
538
539         # Forward model (predict next timestep)
540         self.W_in = nn.Linear(d, d_cb, bias=False)
541         self.W_predict = nn.Linear(d_cb, d, bias=False)
542
543         # Error correction
544         self.W_error = nn.Linear(d, d, bias=False)
545         nn.init.zeros_(self.W_error.weight)
546
547         self.norm = RMSNorm(d)
548         self.gate_raw = nn.Parameter(torch.tensor(cfg.gate_init))
549
550     def forward(self, x: torch.Tensor) -> Tuple[torch.Tensor, torch.Tensor]:
551         """
552         x: [B, T, D_model]
553         Returns:
554           corrected: [B, T, D_model] -- x with cerebellar correction
555           cb_loss: scalar -- cerebellar prediction error (for monitoring)
556         """
557         if not self.enabled:
558             return x, torch.tensor(0.0, device=x.device)
559
560         gate = torch.sigmoid(self.gate_raw)
561         h = self.norm(x)
562
563         # Predict next timestep
564         cb_hidden = F.gelu(self.W_in(h))
565         prediction = self.W_predict(cb_hidden)  # [B, T, D]
566
567         # Temporal error: predicted X[t+1] vs actual X[t+1]
568         if x.shape[1] > 1:
569             # prediction[:, :-1] should match x[:, 1:]
570             error = x[:, 1:] - prediction[:, :-1]
571             cb_loss = error.pow(2).mean()
572
573             # Apply correction to current timestep
574             # Pad error to match T dimension
575             padded_error = F.pad(error, (0, 0, 1, 0))  # [B, T, D]
576             correction = self.W_error(padded_error)
577             return x + gate * correction, cb_loss
578         else:
579             return x, torch.tensor(0.0, device=x.device)
580
581
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```python
#

# Brainstem (Global State Regulation)
#


class Brainstem(nn.Module):
    """
    Brainstem: Global neuromodulatory state regulation.

    Computes all four neuromodulatory signals (Theorem C):
      DA (dopamine): Prediction error, reward (VTA/SNc)
      NE (norepinephrine): Arousal, exploration (Locus Coeruleus)
      5-HT (serotonin): Time horizon, patience (Raphe nuclei)
      ACh (acetylcholine): Attention, plasticity gating (Basal forebrain)

    These signals are broadcast globally to all cortical columns
    and modulate learning, not just activation.
    """

    def __init__(self, cfg: V11Config):
        super().__init__()
        self.cfg = cfg
        d = cfg.d_model

        # DA: Dopamine (prediction error / reward)
        if cfg.da_enabled:
            self.da_proj = nn.Linear(d, 1, bias=True)
            nn.init.zeros_(self.da_proj.weight)
            nn.init.zeros_(self.da_proj.bias)

        # NE: Norepinephrine (arousal / exploration)
        if cfg.ne_enabled:
            self.ne_proj = nn.Linear(d, 1, bias=True)
            nn.init.zeros_(self.ne_proj.weight)
            nn.init.constant_(self.ne_proj.bias, 0.5)  # Moderate arousal baseline

        # 5-HT: Serotonin (time horizon / patience)
        if cfg.serotonin_enabled:
            self.serotonin_proj = nn.Linear(d, 1, bias=True)
            nn.init.zeros_(self.serotonin_proj.weight)
            nn.init.constant_(self.serotonin_proj.bias, 0.5)  # Moderate patience

        # ACh: Acetylcholine (attention / plasticity gate)
        if cfg.ach_enabled:
            self.ach_proj = nn.Linear(d, 1, bias=True)
            nn.init.zeros_(self.ach_proj.weight)
            nn.init.constant_(self.ach_proj.bias, 0.5)  # Moderate attention

        self.norm = RMSNorm(d)

        # Tonic DA EMA
        self.register_buffer('tonic_da', torch.tensor(0.5))

    def compute_state(
        self,
        x: torch.Tensor,
        prediction_error: Optional[torch.Tensor] = None,
    ) -> NeuromodState:
        """
        Compute global neuromodulatory state from current representations.

        x: [B, T, D_model] -- aggregate representation
        prediction_error: [B, T, D_model] -- current prediction errors (for DA)
        Returns: NeuromodState with all four signals
        """
        h = self.norm(x)
        B, T, D = h.shape

        # DA: based on prediction error magnitude
        if self.cfg.da_enabled:
            if prediction_error is not None:
                pe_mag = prediction_error.pow(2).mean(dim=-1, keepdim=True)  # [B, T, 1]
                da = torch.sigmoid(self.da_proj(h) + pe_mag.sqrt())
            else:
                da = torch.sigmoid(self.da_proj(h))
        else:
            da = torch.full((B, T, 1), 0.5, device=x.device)

        # NE: arousal from representation variance
        if self.cfg.ne_enabled:
            ne = torch.sigmoid(self.ne_proj(h))
        else:
            ne = torch.full((B, T, 1), 0.5, device=x.device)

        # 5-HT: patience / time horizon
        if self.cfg.serotonin_enabled:
            serotonin = torch.sigmoid(self.serotonin_proj(h))
        else:
            serotonin = torch.full((B, T, 1), 0.5, device=x.device)

```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        # ACh: attention depth
        if self.cfg.ach_enabled:
            ach = torch.sigmoid(self.ach_proj(h))
        else:
            ach = torch.full((B, T, 1), 0.5, device=x.device)

        # Update tonic DA
        tonic_da_val = self.tonic_da.item()
        with torch.no_grad():
            mean_da = da.mean()
            if torch.isfinite(mean_da):
                self.tonic_da.lerp_(mean_da, 1.0 - self.cfg.tonic_da_ema)

        return NeuromodState(
            dopamine=da,
            norepinephrine=ne,
            serotonin=serotonin,
            acetylcholine=ach,
            tonic_da=tonic_da_val,
        )

    # Plasticity gating is handled by NeuromodulatedPlasticity in v11_learning.py,
    # which computes DA × ACh × f(NE, 5-HT) from the NeuromodState returned here.


#

# Global Workspace (Baars 1988 -- Consciousness Broadcast)
#


class CorticalWorkspace(nn.Module):
    """
    Global Workspace for cortical column network (Baars 1988, Dehaene & Naccache 2001).

    The brain needs a mechanism for columns to communicate beyond local
    lateral connections. The Global Workspace Theory posits:
      1. Columns compete for broadcast access (ignition)
      2. The winning column's representation is broadcast globally
      3. All columns can read the broadcast (L1 top-down input)

    Disease mapping:
      Vegetative state → workspace failure → no global broadcast → unconscious processing only
      Schizophrenia (disconnection) → noisy broadcast → thought disorder

    Architecture:
      Each column writes a candidate broadcast via W_write:
        candidate_l = W_write_l(column_output_l)
      Competition via surprise-weighted softmax:
        weight_l = softmax(surprise_l / temperature)
      Winner broadcast:
        workspace =  _l  weight_l × candidate_l
      Each column reads via W_read:
        modulation_l = W_read_l(workspace)
      Applied as top-down context to L1 on the next iteration.

    Identity at init: W_write=0, gate_raw=-2 → no broadcast initially.
    After training: high-surprise columns broadcast, others listen.

    Params: n_columns × (d_model × d_model + d_model × d_model) + 1 gate
    For n=12, d=512: ~6.3M params (12 × 2 × 512²). Too expensive.
    Use bottleneck: d_ws = d_model // 4 = 128 → 12 × 2 × 512 × 128 = ~1.6M.
    """

    def __init__(self, cfg: V11Config):
        super().__init__()
        self.cfg = cfg
        d = cfg.d_model
        n = cfg.n_columns
        d_ws = max(d // 4, 64)  # Workspace bottleneck dimension

        # Per-column write projection (column → workspace)
        self.W_write = nn.ModuleList([
            nn.Linear(d, d_ws, bias=False) for _ in range(n)
        ])
        # Per-column read projection (workspace → column modulation)
        self.W_read = nn.ModuleList([
            nn.Linear(d_ws, d, bias=False) for _ in range(n)
        ])

        # Initialize write projections to zero → no broadcast at init
        for w in self.W_write:
            nn.init.zeros_(w.weight)
        # Initialize read projections to small random → gradient flows
        for r in self.W_read:
            nn.init.normal_(r.weight, std=0.01)

        # Ignition temperature (controls competition sharpness)
        self.temperature = nn.Parameter(torch.tensor(1.0))

        # Global gate
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        self.gate_raw = nn.Parameter(torch.tensor(cfg.gate_init))

    def broadcast(
        self,
        column_outputs: list,  # List[ColumnOutput] from forward
        errors: list,          # List[torch.Tensor] -- prediction errors per column
    ) -> List[torch.Tensor]:
        """
        Compute global workspace broadcast and return per-column modulations.

        Args:
            column_outputs: list of ColumnOutput from all columns
            errors: prediction error per column [B, T, D]

        Returns:
            modulations: list of [B, T, D] tensors -- top-down workspace signal per column
        """
        gate = torch.sigmoid(self.gate_raw)
        n = len(column_outputs)

        # Compute surprise-based competition weights
        # Surprise = mean squared error per column
        surprises = []
        for l in range(n):
            surprise_l = errors[l].pow(2).mean(dim=-1, keepdim=True)  # [B, T, 1]
            surprises.append(surprise_l)
        surprise_stack = torch.cat(surprises, dim=-1)  # [B, T, n_columns]

        # Softmax competition (higher surprise → more broadcast access)
        temp = F.softplus(self.temperature)
        weights = F.softmax(surprise_stack / temp, dim=-1)  # [B, T, n]

        # Each column writes a candidate
        candidates = []
        for l in range(n):
            cand = self.W_write[l](column_outputs[l].output)  # [B, T, d_ws]
            candidates.append(cand)
        candidates_stack = torch.stack(candidates, dim=-2)  # [B, T, n, d_ws]

        # Weighted broadcast
        w = weights.unsqueeze(-1)  # [B, T, n, 1]
        workspace = (w * candidates_stack).sum(dim=-2)  # [B, T, d_ws]

        # Each column reads the broadcast
        modulations = []
        for l in range(n):
            mod = self.W_read[l](workspace)  # [B, T, D]
            modulations.append(gate * mod)

        return modulations


#

# Self-Assessment & Metacognitive Monitoring
#


class SelfAssessment(nn.Module):
    """
    Metacognitive monitoring module -- the model knows when it's struggling.

    Disease mapping:
      Anosognosia (parietal damage) → no self-awareness of deficits
      OCD (ACC hyperactivity) → excessive conflict monitoring

    Two mechanisms:
      1. Column health monitoring: detect  ² divergence/collapse, PC non-convergence
      2. Conflict detection: when columns disagree, increase PC inference steps

    ACC-like conflict: variance across column predictions at the same level.
    High variance → columns disagree → allocate more inference steps.
    This is the neural basis of "thinking harder" on difficult inputs.

    Zero learnable params -- purely algorithmic monitoring.
    """

    def __init__(self, cfg):
        super().__init__()
        self.cfg = cfg
        # Health thresholds
        self.sigma2_max = 100.0    #  ² above this = diverging
        self.sigma2_min = 0.001    #  ² below this = collapsed
        # Conflict → extra PC steps mapping
        self.conflict_threshold = 0.5  # Variance above this = high conflict
        self.max_extra_steps = 2       # Maximum extra PC steps from conflict (capped for speed)

        # Track health history (per column, running averages)
        n = cfg.n_columns
        self.register_buffer('sigma2_ema', torch.ones(n))
        self.register_buffer('convergence_ema', torch.zeros(n))  # Fraction that hit max_steps
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        self.register_buffer('_health_updates', torch.tensor(0, dtype=torch.long))

    @torch.no_grad()
    def compute_conflict(self, column_outputs: list) -> float:
        """
        Compute inter-column conflict (ACC-like).

        When columns produce very different outputs for the same input,
        the system needs more deliberation.

        Returns: conflict score (0 = agreement, high = disagreement)
        """
        if len(column_outputs) < 2:
            return 0.0

        # Stack column output means: [n_columns, D]
        means = torch.stack([
            out.output.mean(dim=(0, 1)) for out in column_outputs
        ])  # [n_columns, D]

        # Variance across columns = disagreement
        conflict = means.var(dim=0).mean().item()
        return conflict if math.isfinite(conflict) else 0.0

    def extra_pc_steps(self, conflict: float) -> int:
        """Map conflict to extra inference steps (0 to max_extra_steps)."""
        if not math.isfinite(conflict) or conflict < self.conflict_threshold:
            return 0
        # Linear mapping from threshold to 2×threshold → 0 to max_extra_steps
        excess = (conflict - self.conflict_threshold) / max(self.conflict_threshold, 1e-6)
        return min(self.max_extra_steps, int(excess * self.max_extra_steps))

    @torch.no_grad()
    def update_health(self, column_outputs: list, pc_steps_taken: int):
        """Track per-column health metrics."""
        n = len(column_outputs)
        alpha = 0.01  # Slow EMA

        for l in range(min(n, self.cfg.n_columns)):
            sigma2_mean = column_outputs[l].sigma2.mean().item()
            if math.isfinite(sigma2_mean):
                self.sigma2_ema[l] = self.sigma2_ema[l] * (1 - alpha) + sigma2_mean * alpha

        # Track convergence rate
        hit_max = float(pc_steps_taken >= self.cfg.pc_max_steps)
        for l in range(min(n, self.cfg.n_columns)):
            self.convergence_ema[l] = self.convergence_ema[l] * (1 - alpha) + hit_max * alpha

        self._health_updates += 1

    @torch.no_grad()
    def check_health(self, columns) -> dict:
        """
        Check column health and take corrective action if needed.

        Returns: dict of health status per column
        """
        health = {}
        for l in range(min(len(columns), self.cfg.n_columns)):
            sigma2 = self.sigma2_ema[l].item()
            conv_rate = self.convergence_ema[l].item()
            status = 'healthy'

            #  ² divergence detection
            if sigma2 > self.sigma2_max:
                status = 'sigma2_diverging'
                # Emergency: scale down L5 output weights
                with torch.no_grad():
                    columns[l].L5.W_out.weight.data *= 0.9
                    columns[l].L5.W_sigma.weight.data *= 0.9

            #  ² collapse detection
            elif sigma2 < self.sigma2_min:
                status = 'sigma2_collapsed'
                # Nudge  ² back up
                with torch.no_grad():
                    columns[l].L5.W_sigma.bias.data += 0.1

            # PC non-convergence (always hitting max_steps)
            if conv_rate > 0.8:
                status = 'pc_stalled' if status == 'healthy' else status + '+pc_stalled'

            health[f'col{l}_health'] = status

        return health

    def get_metrics(self) -> dict:
        """Return monitoring metrics."""
        return {
            'sigma2_ema_min': self.sigma2_ema.min().item(),
            'sigma2_ema_max': self.sigma2_ema.max().item(),
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
            'convergence_rate': self.convergence_ema.mean().item(),
        }
```

Listing 10: Subcortical Systems — complete source code (v11_systems.py, 938 lines)

12 Local Learning Framework

Source: `nct_v2/v11_learning.py` **Lines:** 1,089 **Classes:** 6

12.1 Overview

NCT implements five complementary learning systems, none of which require global backpropagation through the column hierarchy. These systems are orchestrated by the `V11LearningManager`, which coordinates predictive coding updates, Hebbian plasticity, neuromodulatory gating, wake-sleep scheduling, and developmental staging.

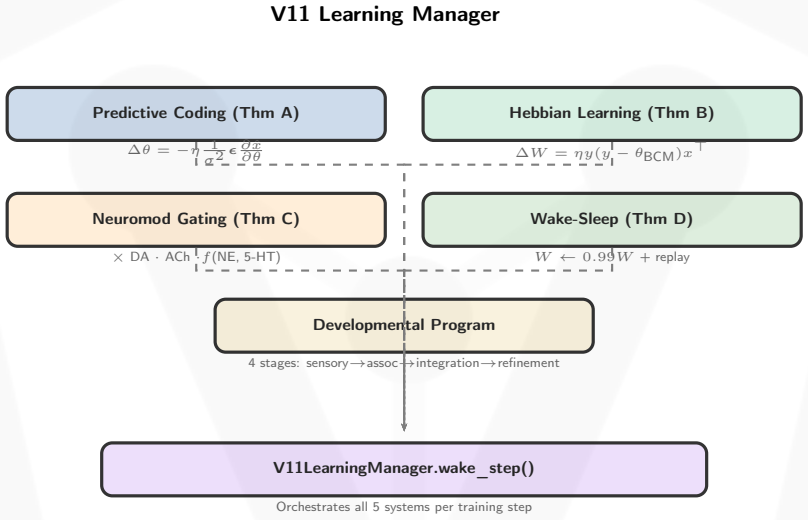


Figure 55: The five learning systems orchestrated by the V11LearningManager. Each system is grounded in a core theorem. None require global backpropagation.

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

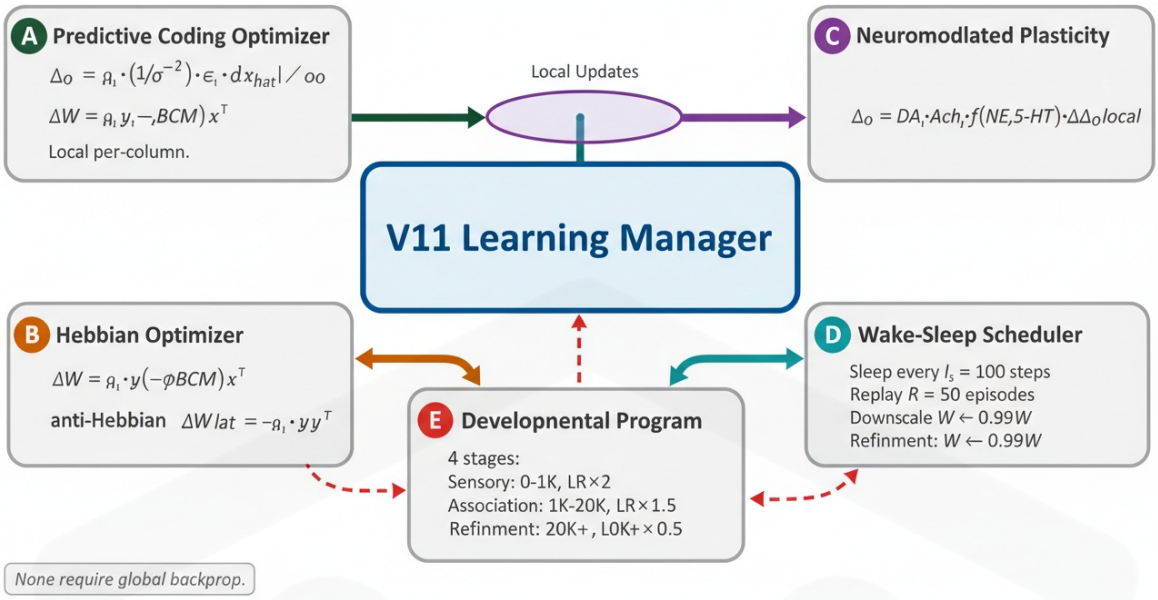


Figure 56: (Illustrated) The five learning systems orchestrated by the V11LearningManager. Each system is grounded in a core theorem. None require global backpropagation.

12.2 Predictive Coding Optimizer (Theorem A)

Theorem 12.1 (Predictive Coding Replaces Backpropagation (Whittington & Bogacz, 2017)). *Each column i in the hierarchy generates a top-down prediction $\hat{\mathbf{x}}_{i-1}$ of the column below's output. The prediction error is:*

$$\boldsymbol{\epsilon}_i = \mathbf{x}_i - \hat{\mathbf{x}}_i \tag{67}$$

Weight updates are strictly local — each column adjusts only its own parameters to minimize its own prediction error, weighted by learned precision σ_i^2:

$$\Delta\theta_i = -\eta_{PC} \cdot \frac{1}{\sigma_i^2} \cdot \boldsymbol{\epsilon}_i \cdot \frac{\partial \hat{\mathbf{x}}_i}{\partial \theta_i} \tag{68}$$

The precision σ^2 is itself learnable. When σ^2 is small (the column expects its predictions to be precise), the learning signal is amplified — small errors are treated as significant. When σ^2 is large (the column is uncertain), the learning signal is attenuated — errors are expected and tolerated. This implements **precision-weighted prediction coding (PWPC)**, matching the brain's ability to distinguish between genuinely surprising events and expected noise.

Inference iterations: PC inference proceeds iteratively — after an initial bottom-up pass, top-down predictions and lateral signals are fed back, and columns re-settle for up to $K_{\max} + K_{\text{extra}}$ iterations. These iterations run **without gradients** (biologically correct: neural settling is dynamical relaxation, not gradient descent). Only the final settled state is used for weight updates.

$$\text{Converge when: } \frac{\left|\|\boldsymbol{\epsilon}^{(k)}\|^2 - \|\boldsymbol{\epsilon}^{(k-1)}\|^2\right|}{\|\boldsymbol{\epsilon}^{(k-1)}\|^2 + 10^{-8}} < \tau_{\text{conv}} = 0.01 \tag{69}$$

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

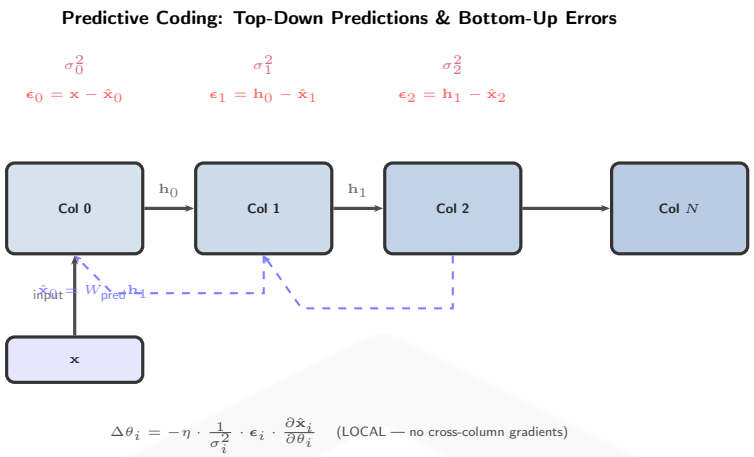


Figure 57: Predictive coding error flow. Each column generates a top-down prediction (dashed blue) of the column below's output. The prediction error $\boldsymbol{\epsilon}_i$ (red), weighted by precision σ_i^2, drives local weight updates. No gradients flow between columns.

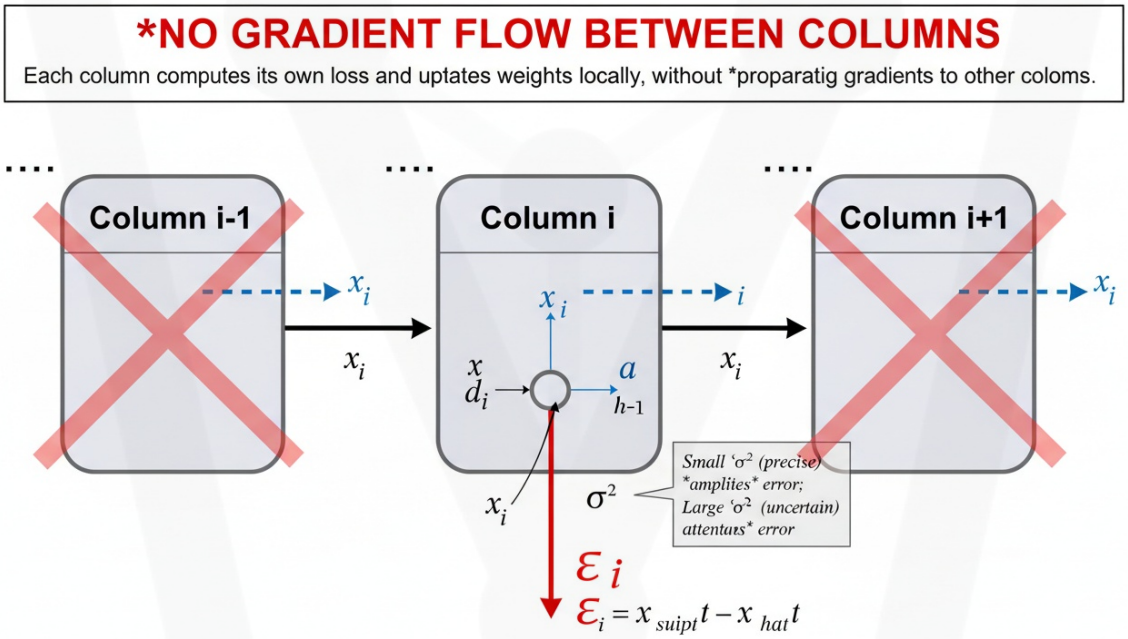


Figure 58: (Illustrated) Predictive coding error flow. Each column generates a top-down prediction (dashed blue) of the column below's output. The prediction error $\boldsymbol{\epsilon}_i$ (red), weighted by precision ...

Proposition 12.1 (Per-Column Loss Decomposition)**.** *The total PC loss decomposes as a sum of independent per-column terms:*

$$\mathcal{L}_{PC} = \sum_{i=0}^{N-1} \frac{1}{\sigma_i^2} \|\boldsymbol{\epsilon}_i\|^2 \tag{70}$$

Each column can compute its own loss and gradient independently — no cross-column gradient flow is needed. This is the key property that enables local learning.

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

12.3 Hebbian Optimizer (Theorem B)

Theorem 12.2 (Hebbian + Anti-Hebbian Convergence (Pehlevan & Chklovskii, 2019)). *The L2/3 association weights W^h_{assoc} are updated by a Hebbian rule with BCM homeostasis:*

$$\Delta W^h_{assoc} = \eta_H \cdot \mathbf{y}^h(\mathbf{y}^h - \theta_{BCM}) \cdot (\mathbf{x}^h)^\top \tag{71}$$

Anti-Hebbian lateral weights enforce decorrelation:

$$\Delta W^h_{lat} = -\eta_H \cdot \mathbf{y}^h(\mathbf{y}^h)^\top \tag{72}$$

This combination provably converges to the PCA subspace of the input distribution, self-organizing representations without any supervised signal. The BCM threshold θ_{BCM} prevents runaway potentiation by providing a sliding activity threshold.

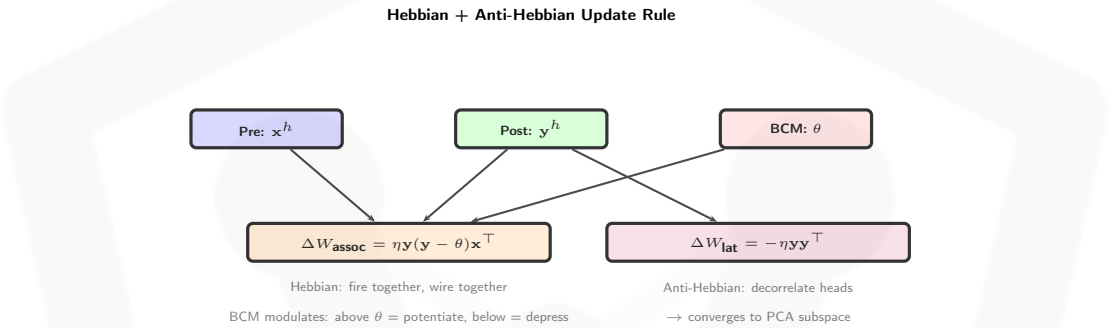


Figure 59: Hebbian update rule: feedforward weights are updated by the outer product of pre- and post-synaptic activity, modulated by the BCM threshold. Anti-Hebbian lateral weights enforce decorrelation between heads. Together, this converges to the PCA subspace.

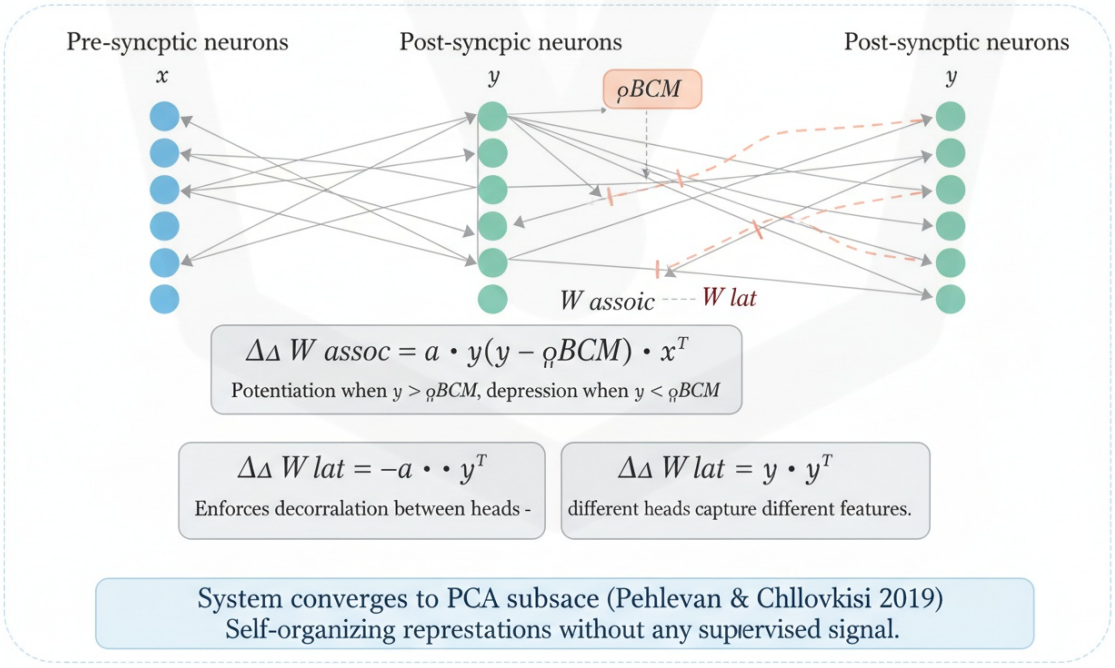


Figure 60: (Illustrated) Hebbian update rule: feedforward weights are updated by the outer product of pre- and post-synaptic activity, modulated by the BCM threshold. Anti-Hebbian lateral weights enforce decorrelation betw...

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

The Hebbian optimizer also updates **local per-column parameters** (L4, L1, L5 projections) using each column's own prediction error as a local target:

$$\Delta W_{\text{local},i} = -\eta_{\text{local}} \cdot \boldsymbol{\epsilon}_i \cdot \frac{\partial \mathbf{h}_i}{\partial W_{\text{local},i}} \tag{73}$$

This is a form of local target propagation: each column's prediction error serves as its own teaching signal.

12.4 Neuromodulated Plasticity (Theorem C)

Theorem 12.3 (Neuromodulatory Gating of Learning (Schultz, 1997))**.** *All learning is multiplicatively gated by four neuromodulatory signals computed by the brainstem:*

$$\Delta\theta = \underbrace{DA}_{\substack{\text{reward}\\ \text{prediction}\\ \text{error}}} \cdot \underbrace{ACh}_{\substack{\text{attention}\\ \text{plasticity}\\ \text{gate}}} \cdot f\left(\underbrace{NE}_{\substack{\text{arousal}\\ \text{exploration}}}, \underbrace{\text{5-HT}}_{\substack{\text{patience}\\ \text{time}\\ \text{horizon}}} \right) \cdot \Delta\theta_{local} \tag{74}$$

where $\Delta\theta_{local}$ is either the PC or Hebbian update. This implements reinforcement learning without backpropagation: dopamine provides policy-gradient-like reward signals, acetylcholine gates whether learning occurs at all, norepinephrine controls the magnitude of exploration noise, and serotonin modulates the effective time horizon (discount factor).

The plasticity coefficient is computed as:

$$\text{plasticity} = \text{mean}(\text{DA}) \cdot \text{mean}(\text{ACh}) \cdot (1 + 0.5 \cdot \text{mean}(\text{NE})) \tag{75}$$

12.5 Wake-Sleep Scheduler (Theorem D)

Theorem 12.4 (Synaptic Homeostasis Hypothesis (Tononi & Cirelli, 2006))**.** *The wake-sleep cycle implements two complementary processes:*

Wake phase: *Process input, update weights via PC + Hebbian, tag surprising episodes in hippocampus.*

Sleep phase: *(1) Replay tagged episodes from HPC buffer (priority: surprise > recency), replaying through columns with local learning. (2) Synaptic downscaling: $W \leftarrow (1 - \delta) \cdot W$ where $\delta = 0.01$ (1% reduction per sleep phase), preventing unbounded weight growth. (3) Consolidation: transfer frequently-replayed patterns from hippocampal episodic to neocortical semantic storage.*

Sleep timing is controlled by the `WakeSleepScheduler`: sleep occurs every I_s wake steps (default 100), with R replay steps per sleep phase (default 50). Sleep pressure accumulates during wake and resets after sleep.

12.6 Developmental Program (Critical Periods)

Theorem 12.5 (Developmental Staging (Hensch, 2004))**.** *The brain changes* how *it learns over time. NCT implements four developmental stages:*

Stage	***Steps***	***LR ×***	***Inhib***	***Hebb wt***	***Struct***
Sensory	*0–1K*	*2.0*	*0.3*	*0.8*	*1.0*
Association	*1K–5K*	*1.5*	*0.6*	*0.6*	*0.7*
Integration	*5K–20K*	*1.0*	*0.9*	*0.4*	*0.3*
Refinement	*20K+*	*0.5*	*1.0*	*0.3*	*0.1*

Early stages are Hebbian-dominated (absorb correlations). Late stages are PC-dominated (refine predictions). Inhibitory strength increases throughout, matching GABA maturation driving critical period closure. Sleep frequency increases in late stages (consolidation becomes the primary learning mechanism).

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

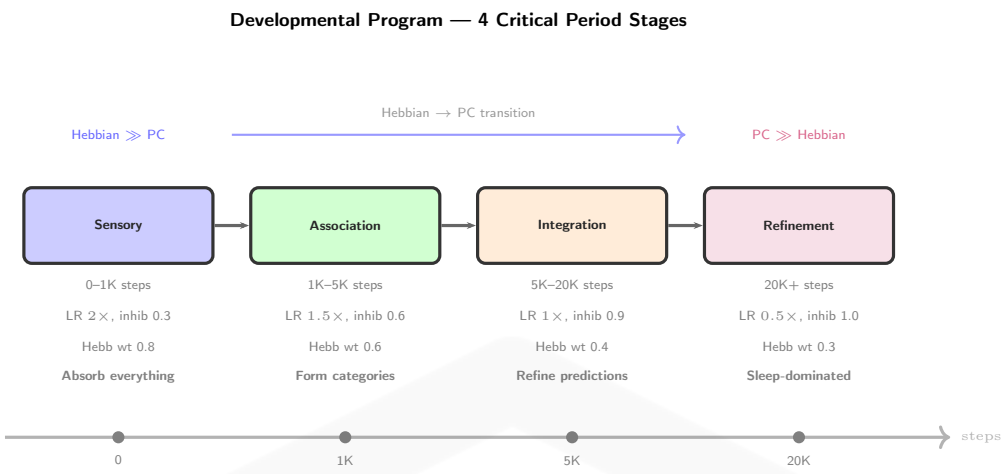


Figure 61: Developmental program: four critical period stages modulate how the model learns over time. Early stages are Hebbian-dominated (rapidly absorb correlations), late stages are PC-dominated (refine predictions). Inhibitory strength increases throughout, matching GABA maturation.

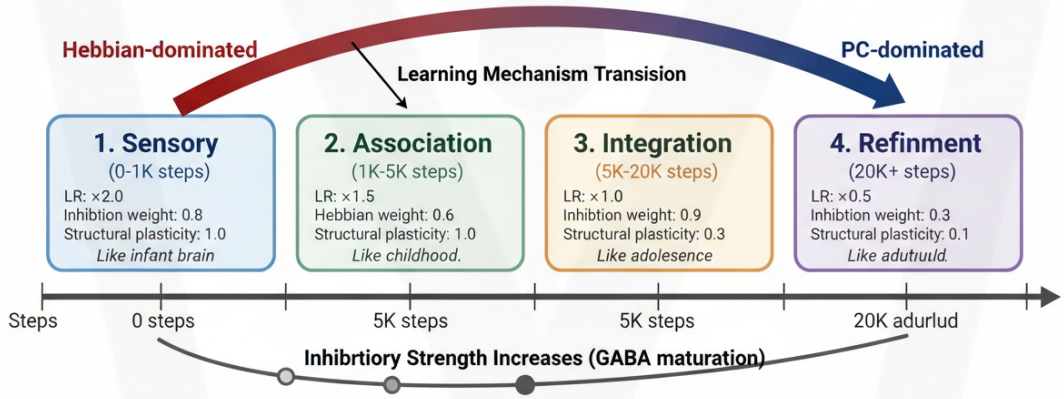


Figure 62: (Illustrated) Developmental program: four critical period stages modulate how the model learns over time. Early stages are Hebbian-dominated (rapidly absorb correlations), late stages are PC-dominated (refine pr...

12.7 V11LearningManager — Orchestration

The `V11LearningManager` class coordinates all five systems. On each wake step:

1. Advance the developmental program clock
2. Get current stage parameters (LR multiplier, inhibitory strength, Hebbian weight)
3. Compute neuromodulatory plasticity coefficient
4. Apply Hebbian updates to L2/3 association weights (all columns)
5. Apply Hebbian updates to local per-column parameters

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

6. Apply PC updates to prediction weights
7. Check if sleep should occur (based on developmental-adjusted sleep interval)
8. If sleep: run replay + downscaling + consolidation

12.8 Source Code

```
1
2
3
4  # ==============================================================================
5  # Local Learning (Predictive Coding + Hebbian)
6  # Source: nct_v2/v11_learning.py
7  # ==============================================================================
8
9  """
10 NCT V11: Local Learning Rules
11 ==============================
12
13 Replaces global backprop with biologically-grounded local learning.
14
15 Three learning mechanisms:
16   1. Predictive Coding (Theorem A): Each column minimizes its own prediction error
17      via local gradients. Whittington & Bogacz (2017) proved equivalence to backprop.
18
19   2. Hebbian Learning (Theorem B): L2/3 associations self-organize via
20      Hebbian + anti-Hebbian rules. No gradients needed.
21
22   3. Neuromodulated Plasticity (Theorem C): DA/NE/5-HT/ACh gate all learning.
23      Plasticity = DA × ACh × f(NE, 5-HT).
24
25 Plus:
26   - Wake-Sleep Scheduler (Theorem D): Manages wake/sleep cycle
27   - BCM homeostasis: Prevents runaway Hebbian learning
28   - Metaplasticity: Learning-to-learn rate adjustment
29 """
30
31
32
33 # [bundled] from .v11_config import V11Config
34 # [bundled] from .v11_column import CorticalColumn, ColumnOutput
35 # [bundled] from .v11_systems import NeuromodState
36
37
38 #

39 # Predictive Coding Optimizer
40 #

41
42 class PredictiveCodingOptimizer:
43     """
44     Local learning via predictive coding (Theorem A).
45
46     Each column l generates a prediction of the column below it.
47     Error:  _l = actual_input_l - prediction_from_{l+1}
48     Weight update:    _l = -  ×  _l ×   f  /    _l (local gradient)
49
50     Implementation: Each column has its own loss function:
51       L_l = || _l||² / (2 ²_l) + log( ²_l)
52     We call .backward() on each column's loss SEPARATELY.
53     Gradients don't flow between columns -- they're LOCAL.
54
55     Whittington & Bogacz (2017) proved this is equivalent to backprop
56     at convergence. We use a fixed number of inference steps (not convergence)
57     because the brain operates in real-time.
58     """
59
60     def __init__(self, cfg: V11Config):
61         self.cfg = cfg
62         self.lr = cfg.pc_learning_lr
63
64     def compute_column_losses(
65         self,
66         predictions: List[torch.Tensor],
67         targets: List[torch.Tensor],
68         precisions: List[torch.Tensor],
69     ) -> Tuple[List[torch.Tensor], torch.Tensor]:
70         """
71         Compute local predictive coding loss for each column.
72
73         predictions[l]: [B, T, D] -- column l+1's prediction of column l's input
74         targets[l]: [B, T, D] -- actual input at level l
75         precisions[l]: [B, T, 1] --  ² at level l
76
77         Returns:
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
          losses: List of per-column losses (each is a scalar)
          total_loss: Sum of all column losses (for monitoring)
        """
        losses = []
        total_loss = torch.tensor(0.0, device=targets[0].device)

        for l in range(len(predictions)):
            pred = predictions[l]
            target = targets[l]
            sigma2 = precisions[l].clamp(min=1e-6)

            # Prediction error
            error = target.detach() - pred  # Detach target -- local learning

            # Precision-weighted MSE + log-precision (ELBO-like)
            loss_l = (error.pow(2) / (2.0 * sigma2) + 0.5 * sigma2.log()).mean()

            losses.append(loss_l)
            total_loss = total_loss + loss_l

        return losses, total_loss

    @torch.no_grad()
    def local_update(
        self,
        columns: nn.ModuleList,
        predictions: List[torch.Tensor],
        targets: List[torch.Tensor],
        precisions: List[torch.Tensor],
        plasticity_gate: Optional[torch.Tensor] = None,
    ):
        """
        Truly local weight updates via predictive coding -- ZERO autograd.

        Each column updates its own W_predict and W_sigma using analytical
        gradients computed directly from prediction errors. No .backward(),
        no autograd graph, no global error propagation.

        This is the Rao & Ballard (1999) / Whittington & Bogacz (2017) PC
        update rule, computed in closed form.

        For W_predict (linear): dW = -(error /  ²) @ input^T / N
        For W_sigma (via softplus): dW = dL/d ² × sigmoid(z) @ input^T / N

        predictions[l]: prediction for level l (from column l)
        targets[l]: actual value at level l
        precisions[l]:  ² at level l
        plasticity_gate: [B, T, 1] -- neuromodulatory plasticity multiplier
        """
        plasticity_mult = 1.0
        if plasticity_gate is not None:
            plasticity_mult = plasticity_gate.mean().item()

        lr = self.lr * plasticity_mult

        for l, col in enumerate(zip(columns)):
            if isinstance(col, tuple):
                col = col[0]
            if l >= len(predictions):
                break

            pred = predictions[l]
            target = targets[l].detach()
            sigma2 = precisions[l].clamp(min=1e-6)

            # Prediction error (local signal)
            error = target - pred.detach()  # [B, T, D_model]

            #         W_predict update: dW = -(error /  ²) @ h_norm^T / N
            # The column stores h_norm from last forward in L5
            # prediction = W_predict(h_norm), so gradient is error @ h_norm^T
            L5 = col.L5
            # Reconstruct h_norm: we need the input that was fed to W_predict
            # L5 stores it as part of the norm output -- access via stored state
            # Since we're in no_grad, we compute from the column output
            h_norm = col._last_l5_h_norm if hasattr(col, '_last_l5_h_norm') else None

            if h_norm is not None:
                B, T, D_model = error.shape
                D_col = h_norm.shape[-1]
                N = B * T

                # Weighted error by precision
                weighted_error = error.float() / sigma2.float()  # [B, T, D_model]

                # W_predict gradient: [D_model, D_col]
                we_flat = weighted_error.reshape(N, D_model)
                h_flat = h_norm.float().reshape(N, D_col)
                dW_predict = -(we_flat.T @ h_flat) / N  # [D_model, D_col]

                # Clip gradient norm
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
                grad_norm = dW_predict.norm()
                if grad_norm > self.cfg.max_grad_norm:
                    dW_predict = dW_predict * (self.cfg.max_grad_norm / grad_norm)

                L5.W_predict.weight.data -= lr * dW_predict.to(L5.W_predict.weight.dtype)

            #         W_sigma update: dL/d ² × sigmoid(z) @ h_norm^T / N
            #  ² = softplus(z) +   where z = W_sigma(h_norm)
            # dL/d ² = -error²/(2 ²²) + 0.5/ ²
            if h_norm is not None and hasattr(L5, 'W_sigma'):
                error_sq = error.float().pow(2).sum(dim=-1, keepdim=True)  # [B, T, 1]
                sigma2_f = sigma2.float()
                dL_dsigma2 = -error_sq / (2.0 * sigma2_f.pow(2)) + 0.5 / sigma2_f  # [B, T, 1]

                # Chain rule through softplus: d ²/dz = sigmoid(z)
                # z = W_sigma(h_norm) + bias → reconstruct z from  ²
                # softplus(z) =  ² -   → z = log(exp( ² -  ) - 1) -- but simpler to use sigmoid
                # sigmoid(z) = 1 - exp(-softplus(z)) = 1 - exp(-( ²- ))
                sigmoid_z = (1.0 - torch.exp(-(sigma2_f - 1e-6))).clamp(0.01, 0.99)

                dL_dz = dL_dsigma2 * sigmoid_z  # [B, T, 1]

                # W_sigma: [1, D_col] with bias
                dz_flat = dL_dz.reshape(N, 1).float()
                h_flat = h_norm.float().reshape(N, D_col)
                dW_sigma = (dz_flat.T @ h_flat) / N  # [1, D_col]
                db_sigma = dz_flat.mean(dim=0)  # [1]

                sigma_grad_norm = dW_sigma.norm()
                if sigma_grad_norm > self.cfg.max_grad_norm:
                    scale = self.cfg.max_grad_norm / sigma_grad_norm
                    dW_sigma = dW_sigma * scale
                    db_sigma = db_sigma * scale

                L5.W_sigma.weight.data -= lr * dW_sigma.to(L5.W_sigma.weight.dtype)
                L5.W_sigma.bias.data -= lr * db_sigma.to(L5.W_sigma.bias.dtype)


#

# Hebbian Optimizer
#


class HebbianOptimizer:
    """
    Hebbian learning for L2/3 lateral associations (Theorem B).

    Updates W_assoc via:  W  =   × BCM(X^T X) / N
    Updates W_lateral via anti-Hebbian:  W  =   × (Y^T Y - I) / N

    BCM modulation prevents runaway learning:
       w      y(y -  _BCM)x
      y >  _BCM → potentiation
      y <  _BCM → depression

    No autograd needed -- pure matrix operations.
    """

    def __init__(self, cfg: V11Config):
        self.cfg = cfg
        self.lr = cfg.hebbian_lr

    @torch.no_grad()
    def update(
        self,
        columns: nn.ModuleList,
        column_outputs: List[ColumnOutput],
        plasticity_gate: Optional[torch.Tensor] = None,
        errors: Optional[List[torch.Tensor]] = None,
    ):
        """
        Perform Hebbian updates for all columns' L2/3 layers.

        Three improvements over blind Hebbian:
          1. BCM modulation (prevents runaway learning)
          2. Metaplasticity LR (learning-to-learn)
          3. Error-driven modulation (learn from mistakes, not just correlations)
          4.  ²-adaptive LR (uncertain columns learn faster)

        column_outputs: List of ColumnOutput from forward pass
        plasticity_gate: [B, T, 1] neuromodulatory gating
        errors: List of per-column prediction errors (for error-driven Hebbian)
        """
        lr = self.lr
        if plasticity_gate is not None:
            lr = lr * plasticity_gate.mean().item()

        for i, (col, out) in enumerate(zip(columns, column_outputs)):
            # Metaplasticity LR multiplier
            meta_mult = col.metaplasticity.get_lr_multiplier()
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
            effective_lr = lr * meta_mult

            #  ²-Adaptive LR: uncertain columns learn faster
            # sigma2 is [B, T, 1] -- use mean as scalar multiplier
            sigma2_mean = out.sigma2.mean().item()
            sigma2_mult = 1.0 + min(sigma2_mean, 10.0)  # Cap at 11x to prevent explosion
            effective_lr = effective_lr * sigma2_mult

            # Error-driven Hebbian (if errors provided and BCM enabled)
            error_for_col = errors[i] if errors is not None and i < len(errors) else None

            if col.bcm.enabled and error_for_col is not None and self.cfg.error_hebbian_weight > 0:
                col.L23.hebbian_update_error_modulated(
                    pre_input=out.l23_input,
                    post_output=out.l23_output,
                    error=error_for_col,
                    bcm_theta=col.bcm.theta_bcm,
                    lr=effective_lr,
                    error_weight=self.cfg.error_hebbian_weight,
                )
            elif col.bcm.enabled:
                col.L23.hebbian_update_bcm(
                    pre_input=out.l23_input,
                    post_output=out.l23_output,
                    bcm_theta=col.bcm.theta_bcm,
                    lr=effective_lr,
                )
            else:
                col.L23.hebbian_update(out.l23_input, lr=effective_lr)

            # Anti-Hebbian update for W_lateral (decorrelation)
            col.L23.anti_hebbian_update(out.l23_output, lr=effective_lr)

            # BCM threshold update
            activity = out.l23_output.abs().mean(dim=-1)  # [B, T]
            col.bcm.update_threshold(activity)

            # Metaplasticity: track weight change magnitude
            w_change = col.L23.W_assoc.abs().mean().item()
            col.metaplasticity.update(w_change)

    @torch.no_grad()
    def update_local_params(
        self,
        columns: nn.ModuleList,
        column_outputs: list,
        errors: Optional[list] = None,
        plasticity_gate: Optional[torch.Tensor] = None,
    ):
        """
        Update ALL locally-learnable parameters beyond L2/3 W_assoc.

        Fixes dead parameters: L1, L6, oscillations, interneurons, L5 dendritic.
        Each uses biologically-grounded local learning rules -- zero autograd.

        L1: Prediction-error Hebbian -- td_context × error correlation
        L6: Thalamic feedback -- column state × error correlation
        Oscillations: Phase-error anti-correlation for PAC coupling
        Interneurons: Activity-dependent error-modulated plasticity
        L5 Dendritic:  ²-error correlation for adaptive gating thresholds
        """
        if errors is None or len(errors) == 0:
            return

        base_lr = self.lr * 0.1  # Slower than L2/3 Hebbian (stability)
        if plasticity_gate is not None:
            base_lr = base_lr * plasticity_gate.mean().item()

        for i, (col, out) in enumerate(zip(columns, column_outputs)):
            error = errors[i] if i < len(errors) else None
            if error is None:
                continue

            B, T, D_model = error.shape
            D_col = out.state.shape[-1]
            N = max(B * T, 1)

            # Per-column adaptive LR
            meta_mult = col.metaplasticity.get_lr_multiplier()
            sigma2_mult = 1.0 + min(out.sigma2.mean().item(), 10.0)
            lr = base_lr * meta_mult * sigma2_mult

            error_flat = error.reshape(N, D_model).float()
            error_mag = error.float().norm(dim=-1, keepdim=True).mean().item()

            #        L1: Top-Down Prediction Error Hebbian
            # L1.proj maps D_model → D_col. Top-down context comes from
            # higher column's prediction. Learn: strengthen td paths that
            # correlate with what the column needs (= prediction error).
            #  W  =   × (error_in_col_space^T @ td_context) / N
            #
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
            # We don't have td_context stored directly, but l23_input
            # includes L1 modulation. For the first iteration, use the
            # column's own prediction error projected to column space.
            if error_mag > 0.01:  # Only learn when error is meaningful
                # Project error to column space via L4 projection
                err_col = F.linear(error.float(), col.L4.proj.weight.float())  # [B, T, D_col]
                err_col_flat = err_col.reshape(N, D_col)

                # L1.proj: [D_col, D_model]
                # Use l23_input as proxy for what L1 received (it's L4 + L1 output)
                l23_in_flat = out.l23_input.reshape(N, D_col).float()

                # Hebbian: correlate error (in col space) with column input
                # This teaches L1 to modulate in directions that track prediction error
                dW_l1 = (err_col_flat.T @ l23_in_flat) / N  # [D_col, D_col]
                # But L1.proj is [D_col, D_model] -- we need proper dimension
                # Use the error itself as the post-synaptic signal
                dW_l1_proper = (err_col_flat.T @ error_flat) / N  # [D_col, D_model]
                col.L1.proj.weight.data -= lr * dW_l1_proper.to(col.L1.proj.weight.dtype)

                # Norm guard
                w_norm = col.L1.proj.weight.data.norm()
                if w_norm > 5.0:
                    col.L1.proj.weight.data *= 5.0 / w_norm

                # L1 gate: open more when error is high (need more top-down context)
                col.L1.gate_raw.data += lr * 0.01 * min(error_mag, 5.0)
                col.L1.gate_raw.data.clamp_(max=2.0)  # Don't exceed ~0.88 gate

            #        L6: Thalamic Feedback Learning

            # L6.proj maps D_col → D_model. L6 sends feedback to thalamus.
            # Learn: send signals that correlate with prediction error,
            # so thalamus can better filter future input.
            #  W  =   × (error^T @ column_state) / N
            state_flat = out.state.reshape(N, D_col).float()
            dW_l6 = (error_flat.T @ state_flat) / N  # [D_model, D_col]
            col.L6.proj.weight.data -= lr * dW_l6.to(col.L6.proj.weight.dtype)

            # Norm guard
            w_norm = col.L6.proj.weight.data.norm()
            if w_norm > 5.0:
                col.L6.proj.weight.data *= 5.0 / w_norm

            # L6 gate: open more with error
            col.L6.gate_raw.data += lr * 0.01 * min(error_mag, 5.0)
            col.L6.gate_raw.data.clamp_(max=2.0)

            #        Oscillation PAC Coupling

            # Kappa1/2/3 control phase-amplitude coupling strength.
            # Learn: adjust coupling based on phase-error anti-correlation.
            # If error peaks at specific phases → coupling at those phases
            # is amplifying errors → reduce that coupling.
            if col.oscillations.enabled:
                osc = col.oscillations
                osc_lr = lr * 0.1  # Extra slow (oscillation stability)

                # Reconstruct phases
                t_idx = torch.arange(T, device=error.device, dtype=error.dtype)
                P_theta = F.softplus(osc.period_theta_raw)
                P_alpha = F.softplus(osc.period_alpha_raw)
                P_delta = F.softplus(osc.period_delta_raw)
                phi_theta = 2 * math.pi * t_idx / P_theta
                phi_alpha = 2 * math.pi * t_idx / P_alpha
                phi_delta = 2 * math.pi * t_idx / P_delta

                # Error magnitude per position [T]
                error_per_t = error.float().norm(dim=-1).mean(dim=0)  # [T]
                error_norm = error_per_t / (error_per_t.mean() + 1e-8)

                # Phase-error correlation
                corr_theta = (error_norm * phi_theta.cos()).mean().item()
                corr_delta = (error_norm * phi_delta.cos()).mean().item()
                corr_alpha = (error_norm * phi_alpha.cos()).mean().item()

                # Anti-correlate: reduce coupling where error aligns with phase
                osc.kappa1_raw.data -= osc_lr * corr_theta
                osc.kappa2_raw.data -= osc_lr * corr_delta
                osc.kappa3_raw.data += osc_lr * corr_alpha  #   suppression: increase

                # Clamp kappas to prevent divergence
                osc.kappa1_raw.data.clamp_(-2.0, 2.0)
                osc.kappa2_raw.data.clamp_(-2.0, 2.0)
                osc.kappa3_raw.data.clamp_(-2.0, 2.0)

                # W_out: oscillation output projection [D_col, r]
                # Recompute interaction for gradient
                r = osc.W_content.weight.shape[0]
                content = osc.W_content(out.state.float())  # [B, T, r]
                phase_features = torch.stack([
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
                    phi_theta.sin(), phi_theta.cos(),
                    phi_alpha.sin(), phi_alpha.cos(),
                    phi_delta.sin(), phi_delta.cos(),
                ], dim=-1)  # [T, 6]
                phase_proj = osc.W_phase(phase_features).unsqueeze(0)  # [1, T, r]
                interaction = torch.tanh(content + phase_proj)  # [B, T, r]

                # dW_out     (err_col^T @ interaction) / N
                err_col_osc = F.linear(error.float(), col.L4.proj.weight.float())
                err_col_osc_flat = err_col_osc.reshape(N, D_col)
                inter_flat = interaction.reshape(N, r)
                dW_out_osc = (err_col_osc_flat.T @ inter_flat) / N  # [D_col, r]
                osc.W_out.weight.data -= osc_lr * dW_out_osc.to(osc.W_out.weight.dtype)

                # W_out norm guard
                w_norm = osc.W_out.weight.data.norm()
                if w_norm > 3.0:
                    osc.W_out.weight.data *= 3.0 / w_norm

            #         Interneuron Weights

            inter = col.interneurons
            inter_lr = lr * 0.05  # Very slow for inhibitory stability

            # Per-token error magnitude (shared across interneuron updates)
            err_mag_tok = error.float().norm(dim=-1, keepdim=True)  # [B, T, 1]

            # VIP: Learn to detect novelty → disinhibit when error is high
            if inter.vip_enabled:
                # VIP reads input, should fire more when prediction error is high
                h_norm_vip = inter.vip_norm(out.l23_input.float())
                h_flat_vip = h_norm_vip.reshape(N, D_col)
                vip_act = F.gelu(inter.vip_in(h_flat_vip))  # [N, d_i]
                # Error-modulated Hebbian: strengthen features that predict error
                dW = (vip_act.T * err_mag_tok.reshape(N).unsqueeze(0)) @ h_flat_vip / N  # [d_i, D_col]
                inter.vip_in.weight.data += inter_lr * dW.to(inter.vip_in.weight.dtype)
                # VIP gate: open more when error is high
                inter.vip_gate_raw.data += inter_lr * 0.1 * min(error_mag, 5.0)
                inter.vip_gate_raw.data.clamp_(max=2.0)

            # SST: Learn better input suppression
            if inter.sst_enabled:
                # SST should strengthen when error is LOW (good inhibition)
                h_norm_sst = inter.sst_norm(out.state.float()).reshape(N, D_col)
                neg_err = 1.0 / (1.0 + err_mag_tok.reshape(N, 1))  # Low error → high
                sst_act = F.gelu(inter.sst_in(h_norm_sst))  # [N, d_i]
                dW_sst = ((sst_act * neg_err).T @ h_norm_sst) / N  # [d_i, D_col]
                inter.sst_in.weight.data += inter_lr * dW_sst.to(inter.sst_in.weight.dtype)

            # PV: Learn veto threshold (stronger veto when output leads to error)
            if inter.pv_enabled:
                h_norm_pv = inter.pv_norm(out.state.float()).reshape(N, D_col)
                pv_act = F.gelu(inter.pv_in(h_norm_pv))
                err_weighted_pv = pv_act * err_mag_tok.reshape(N, 1)
                dW_pv = (err_weighted_pv.T @ h_norm_pv) / N
                inter.pv_in.weight.data += inter_lr * dW_pv.to(inter.pv_in.weight.dtype)

            #         L5 Dendritic Gate Adaptation

            # W_dend.bias: threshold for dendritic gating
            # High error → lower bias → more selective gating
            if col.L5.dendritic:
                dend_lr = lr * 0.01
                # Adaptive threshold: tighten when error is high, relax when low
                bias_delta = -min(error_mag, 5.0) * 0.1 + 0.05  # Net relaxation if error < 0.5
                col.L5.W_dend.bias.data += dend_lr * bias_delta
                col.L5.W_dend.bias.data.clamp_(-3.0, 8.0)

                # W_sigma_adapt: learn  ²→dendritic correlation
                sigma_err = out.sigma2.float().mean().item() * error_mag
                col.L5.W_sigma_adapt.weight.data += dend_lr * sigma_err * 0.01


#

# Neuromodulated Plasticity
#


class NeuromodulatedPlasticity:
    """
    Theorem C: All learning gated by neuromodulators.

    Plasticity = DA × ACh × f(NE, 5-HT)

    DA: gates WHETHER to learn (reward prediction error)
    ACh: gates WHAT to learn (attentional selection)
    NE: modulates HOW BROADLY to learn (exploration vs exploitation)
    5-HT: modulates TIMESCALE (short vs long horizon)

    This multiplier is applied to both PC and Hebbian updates.
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
    """

    def __init__(self, cfg: V11Config):
        self.cfg = cfg

    def compute_plasticity(self, neuromod: NeuromodState) -> torch.Tensor:
        """
        Compute plasticity multiplier from neuromodulatory state.

        Returns: [B, T, 1] plasticity gate     [0, 1]
        """
        # Core plasticity: DA × ACh
        plasticity = neuromod.dopamine * neuromod.acetylcholine

        # NE modulates breadth (high NE → broader learning)
        ne_factor = 0.5 + 0.5 * neuromod.norepinephrine

        # 5-HT modulates timescale (high 5-HT → longer credit assignment)
        serotonin_factor = 0.5 + 0.5 * neuromod.serotonin

        return plasticity * ne_factor * serotonin_factor


#

# Wake-Sleep Scheduler
#


class WakeSleepScheduler:
    """
    Theorem D: Autonomous wake-sleep cycle.

    WAKE phase (online):
      - Process input through columns
      - Predictive coding: local weight updates
      - Hebbian: L2/3 self-organization
      - Hippocampus: tag surprising events

    SLEEP phase (offline self-improvement):
      - Replay high-surprise memories from hippocampus
      - Synaptic downscaling (Tononi 2006): global renormalization
      - Memory consolidation: strengthen replayed patterns
      - Pruning: remove weak connections

    The cycle runs autonomously: no human triggers.
    """

    def __init__(self, cfg: V11Config):
        self.cfg = cfg
        self.wake_steps = 0
        self.sleep_phases = 0
        self.total_steps = 0

    def should_sleep(self) -> bool:
        """Check if it's time for a sleep phase."""
        if not self.cfg.wake_sleep_enabled:
            return False
        return self.wake_steps >= self.cfg.sleep_interval

    def on_wake_step(self):
        """Called after each wake step."""
        self.wake_steps += 1
        self.total_steps += 1

    def on_sleep_start(self):
        """Called when sleep phase begins."""
        self.wake_steps = 0
        self.sleep_phases += 1

    # Parameters to protect from synaptic downscaling (they're "state" not "weights")
    _DOWNSCALE_EXCLUDE = (
        'gate_raw',       # Gate set points
        'bias',           # Bias set points
        'sigma2',         # Precision parameters
        'decay_bias',     # Memory decay constants (HPC, NMDA)
        'decay',          # Decay parameters (phonological loop)
        'period_',        # Oscillatory period parameters
        'kappa',          # PAC coupling strengths
        'ec50',           # Receptor thresholds (BG D1/D2)
        'slope',          # Receptor steepness (BG D1/D2)
        'facil_',         # Facilitation parameters (silent trace)
        'depress_',       # Depression parameters (silent trace)
        'theta_bcm',      # BCM threshold (buffer, but protect if iterated)
        'tonic_da',       # Tonic DA EMA (buffer)
    )

    @torch.no_grad()
    def synaptic_downscaling(self, model: nn.Module, strength: float = 1.0):
        """
        Tononi's Synaptic Homeostasis Hypothesis (2006):
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        During sleep, all synaptic weights are multiplicatively downscaled.

        Purpose: Prevents weight explosion from continuous Hebbian learning.
        Preserves relative weight magnitudes (signal-to-noise).
        Frees capacity for new learning in the next wake cycle.

        Protected: gates, biases, precision, decay/period/receptor params.

        Args:
            strength: 1.0 = full sleep downscale. <1.0 = partial (micro-downscale
                      during wake for continuous homeostasis).
        """
        # Effective scale: lerp between 1.0 (no change) and cfg value
        scale = 1.0 - strength * (1.0 - self.cfg.synaptic_downscale)
        for name, param in model.named_parameters():
            # Skip protected parameters
            if any(excl in name for excl in self._DOWNSCALE_EXCLUDE):
                continue
            # Downscale weights
            if param.data.abs().max() > 1e-8:
                param.data.mul_(scale)

    def get_status(self) -> dict:
        """Return scheduler status for logging."""
        return {
            'wake_steps': self.wake_steps,
            'sleep_phases': self.sleep_phases,
            'total_steps': self.total_steps,
            'next_sleep_in': max(0, self.cfg.sleep_interval - self.wake_steps),
        }


#

# Developmental Program (Critical Periods)
#


class DevelopmentalProgram:
    """
    The brain changes HOW it learns over time. This is not just "learning" --
    it's learning to learn differently at different developmental stages.

    Neuroscience:
      - Hensch (2004): Critical periods are opened by inhibitory maturation
      - Petanjek (2011): Prefrontal pruning continues until age 25
      - Huttenlocher (1979): Synapse density peaks at age 2, halves by adulthood
      - Takesian & Hensch (2013): GABA maturation triggers critical period onset

    Disease mapping:
      Amblyopia → missed visual critical period → permanent deficit
      Language deprivation → missed language CP → impaired grammar acquisition
      Fragile X → abnormal metaplasticity → no proper CP regulation

    Stages (mapped to training steps):
      1. SENSORY (0-1K steps): High learning rate, low inhibition, open to all input.
         Like infant brain -- rapidly absorbing everything. Hebbian dominance.
      2. ASSOCIATION (1K-5K): Moderate LR, inhibitory maturation, lateral connections
         strengthen. Like childhood -- forming categorical boundaries.
      3. INTEGRATION (5K-20K): Lower LR, strong inhibition, top-down predictions
         dominate. Like adolescence -- predictive model crystallizing.
      4. REFINEMENT (20K+): Low LR, structural plasticity slows, sleep consolidation
         becomes primary learning mechanism. Like adulthood -- fine-tuning.

    Each stage modulates: learning rate, inhibitory strength, sleep frequency,
    structural plasticity rate, Hebbian vs PC balance.

    Zero learnable params -- purely algorithmic staging.
    """

    def __init__(self, cfg: V11Config):
        self.cfg = cfg
        self.stage_boundaries = [0, 1000, 5000, 20000]  # Step thresholds
        self.stage_names = ['sensory', 'association', 'integration', 'refinement']
        self._current_step = 0
        # Store original sleep interval -- cfg.sleep_interval gets mutated
        self._base_sleep_interval = cfg.sleep_interval

    @property
    def current_stage(self) -> str:
        for i in range(len(self.stage_boundaries) - 1, -1, -1):
            if self._current_step >= self.stage_boundaries[i]:
                return self.stage_names[min(i, len(self.stage_names) - 1)]
        return self.stage_names[0]

    @property
    def stage_index(self) -> int:
        for i in range(len(self.stage_boundaries) - 1, -1, -1):
            if self._current_step >= self.stage_boundaries[i]:
                return min(i, len(self.stage_names) - 1)
        return 0
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
703
704     def step(self):
705         """Advance developmental clock."""
706         self._current_step += 1
707
708     def get_lr_multiplier(self) -> float:
709         """Learning rate multiplier based on developmental stage."""
710         # Sensory: 2x, Association: 1.5x, Integration: 1x, Refinement: 0.5x
711         multipliers = [2.0, 1.5, 1.0, 0.5]
712         return multipliers[self.stage_index]
713
714     def get_inhibitory_strength(self) -> float:
715         """Inhibitory interneuron strength based on maturation.
716         Low early (everything gets through) → high late (strong filtering).
717         Hensch (2004): inhibitory maturation triggers critical period closure."""
718         strengths = [0.3, 0.6, 0.9, 1.0]
719         return strengths[self.stage_index]
720
721     def get_hebbian_weight(self) -> float:
722         """Balance between Hebbian (blind) and error-driven (PC) learning.
723         Early: Hebbian dominates (learn correlations).
724         Late: Error-driven dominates (refine predictions)."""
725         weights = [0.8, 0.6, 0.4, 0.3]
726         return weights[self.stage_index]
727
728     def get_structural_plasticity_rate(self) -> float:
729         """Structural plasticity rate. High early (rapid wiring), low late (stable).
730         Huttenlocher 1979: synapse density peaks at age 2."""
731         rates = [1.0, 0.7, 0.3, 0.1]
732         return rates[self.stage_index]
733
734     def get_sleep_interval(self) -> int:
735         """Sleep frequency. Less frequent early (more wake learning), more frequent late
736         (consolidation becomes primary mechanism).
737         Uses ORIGINAL sleep interval as base (not the mutated cfg value)."""
738         base = self._base_sleep_interval
739         multipliers = [2.0, 1.5, 1.0, 0.7]
740         return max(10, int(base * multipliers[self.stage_index]))
741
742     def get_metrics(self) -> dict:
743         return {
744             'dev_stage': self.current_stage,
745             'dev_step': self._current_step,
746             'dev_lr_mult': self.get_lr_multiplier(),
747             'dev_inhib': self.get_inhibitory_strength(),
748             'dev_hebbian_weight': self.get_hebbian_weight(),
749         }
750
751
752 #

753 # Integrated Learning Manager
754 #

755
756 class V11LearningManager:
757     """
758     Manages all learning in V11: PC + Hebbian + Neuromod + Wake-Sleep + Development.
759
760     This is the "brain's learning system" -- coordinates all plasticity.
761     """
762
763     def __init__(self, cfg: V11Config):
764         self.cfg = cfg
765         self.pc_optimizer = PredictiveCodingOptimizer(cfg)
766         self.hebbian_optimizer = HebbianOptimizer(cfg)
767         self.neuromod_plasticity = NeuromodulatedPlasticity(cfg)
768         self.wake_sleep = WakeSleepScheduler(cfg)
769         self.development = DevelopmentalProgram(cfg)
770
771     def wake_step(
772         self,
773         columns: nn.ModuleList,
774         column_outputs: List[ColumnOutput],
775         predictions: List[torch.Tensor],
776         targets: List[torch.Tensor],
777         precisions: List[torch.Tensor],
778         neuromod: NeuromodState,
779         errors: Optional[List[torch.Tensor]] = None,
780         ewc_guard=None,
781     ) -> Dict[str, float]:
782         """
783         Complete wake-phase learning step.
784
785         1. Compute neuromodulatory plasticity gate
786         2. Predictive coding updates (vertical connections)
787         3. Hebbian updates (lateral connections in L2/3) -- error-driven +  ²-adaptive
788         4. Track wake step count
789
790         Returns: dict of learning metrics
791         """
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        metrics = {}

        # 0. Developmental modulation (critical periods)
        dev_lr_mult = self.development.get_lr_multiplier()
        dev_hebbian_w = self.development.get_hebbian_weight()
        dev_inhib = self.development.get_inhibitory_strength()
        dev_struct_rate = self.development.get_structural_plasticity_rate()
        self.development.step()
        metrics.update(self.development.get_metrics())

        # 1. Neuromodulatory plasticity gate (modulated by developmental stage)
        plasticity_gate = self.neuromod_plasticity.compute_plasticity(neuromod)
        plasticity_gate = plasticity_gate * dev_lr_mult  # Developmental LR modulation
        metrics['plasticity_mean'] = plasticity_gate.mean().item()

        # Apply developmental inhibitory maturation to interneurons
        # Hensch (2004): inhibitory maturation triggers critical period closure
        for col in columns:
            if hasattr(col.interneurons, 'sst_tonic'):
                # Scale SST tonic inhibition by developmental stage
                col.interneurons.sst_tonic = 0.10 * dev_inhib / 1.0  # max 0.10 at full maturity

        # 2. Predictive coding update
        # PC local_update uses analytical gradients (no autograd). Since hybrid mode
        # also uses analytical local output correction (no backprop), there's no
        # conflict -- both can fire.  ² evolution depends on PC updates.
        if self.cfg.learning_mode in ('predictive_coding', 'hybrid'):
            _, pc_loss = self.pc_optimizer.compute_column_losses(
                predictions, targets, precisions
            )
            metrics['pc_loss'] = pc_loss.item()

            # Analytical PC update: updates W_predict and W_sigma per column
            self.pc_optimizer.local_update(
                columns, predictions, targets, precisions, plasticity_gate
            )

        # 3. Hebbian update -- error-driven +  ²-adaptive + BCM + metaplasticity
        # dev_hebbian_w modulates Hebbian strength: high early (correlation), low late (error-driven)
        if self.cfg.learning_mode in ('hebbian', 'hybrid'):
            # Scale Hebbian LR by developmental Hebbian weight
            orig_lr = self.hebbian_optimizer.lr
            self.hebbian_optimizer.lr = orig_lr * dev_hebbian_w
            self.hebbian_optimizer.update(
                columns, column_outputs, plasticity_gate, errors=errors,
            )
            self.hebbian_optimizer.lr = orig_lr  # Restore
            metrics['hebbian_updated'] = True

        # 3b. Local parameter updates for L1, L6, oscillations, interneurons, dendrites
        # These parameters were previously "dead" -- never updated by any learning rule.
        # Now they learn via biologically-grounded local rules (zero autograd).
        if self.cfg.learning_mode in ('hebbian', 'hybrid'):
            orig_lr = self.hebbian_optimizer.lr
            self.hebbian_optimizer.lr = orig_lr * dev_hebbian_w
            self.hebbian_optimizer.update_local_params(
                columns, column_outputs, errors=errors,
                plasticity_gate=plasticity_gate,
            )
            self.hebbian_optimizer.lr = orig_lr
            metrics['local_params_updated'] = True

        # 4. Structural plasticity (Theorem E) -- prune/grow at interval
        # Modulated by developmental stage: high early (rapid wiring), low late (stable)
        if self.cfg.structural_plasticity:
            step = self.wake_sleep.wake_steps
            if step > 0 and step % self.cfg.prune_interval == 0:
                mean_err = 0.0
                if errors:
                    mean_err = sum(e.abs().mean().item() for e in errors) / len(errors)
                # Scale growth threshold inversely by dev rate (lower threshold = more growth in early stages)
                effective_growth_thresh = self.cfg.growth_threshold / max(dev_struct_rate, 0.01)
                for col in columns:
                    col.L23.structural_plasticity_step(
                        error_magnitude=mean_err,
                        prune_threshold=self.cfg.prune_threshold,
                        growth_threshold=effective_growth_thresh,
                        prune_interval=self.cfg.prune_interval,
                    )
                metrics['structural_plasticity'] = True
                metrics['density_col0'] = columns[0].L23.get_connection_density()

        # 4b. Hebbian weight norm guard -- prevent W_assoc explosion
        # BCM handles the learning dynamics, but if for any reason weights
        # grow beyond a safe norm, hard-clip them. This is the neural equivalent
        # of synaptic saturation -- there's a physical limit to synaptic strength.
        HEBBIAN_MAX_NORM = 10.0
        for i, col in enumerate(columns):
            w_norm = col.L23.W_assoc.data.norm()
            if w_norm > HEBBIAN_MAX_NORM:
                col.L23.W_assoc.data.mul_(HEBBIAN_MAX_NORM / w_norm)
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
                metrics[f'hebb_clipped_col{i}'] = True
            # Also guard W_lateral
            if hasattr(col.L23, 'W_lateral'):
                wl_norm = col.L23.W_lateral.data.norm()
                if wl_norm > HEBBIAN_MAX_NORM:
                    col.L23.W_lateral.data.mul_(HEBBIAN_MAX_NORM / wl_norm)

        # 4c. EWC elastic pull -- protect consolidated knowledge (Kirkpatrick 2017)
        # After Hebbian/PC updates move weights, pull important params back toward anchors.
        #    _i -=    * F_i * ( _i -  *_i)  applied directly to parameters.
        if ewc_guard is not None and ewc_guard._initialized:
            ewc_lr = 1e-4  # Elastic pull strength
            ewc_applied = 0
            for name, param in columns.named_parameters() if hasattr(columns, 'named_parameters') else []:
                full_name = name
                if full_name in ewc_guard.importance and full_name in ewc_guard.anchor_params:
                    anchor = ewc_guard.anchor_params[full_name].to(param.device)
                    importance = ewc_guard.importance[full_name].to(param.device)
                    param.data -= ewc_lr * importance * (param.data - anchor)
                    ewc_applied += 1
            if ewc_applied > 0:
                metrics['ewc_params_constrained'] = ewc_applied

        # 5. Micro-downscaling during wake (continuous homeostasis)
        # The brain doesn't wait for sleep to regulate weights -- continuous
        # synaptic scaling operates alongside learning (Turrigiano 2008).
        # Strength = 0.1 → 10% of full sleep downscale per wake step.
        # This prevents Hebbian weight explosion between sleep phases.
        if self.wake_sleep.wake_steps % 10 == 0:  # Every 10 steps
            import itertools
            for col in columns:
                for p in col.L23.parameters():
                    name = ''  # L23 weights are always eligible
                    if p.data.abs().max() > 1e-8:
                        p.data.mul_(1.0 - 0.05 * (1.0 - self.cfg.synaptic_downscale))

        # 6.  ² emergency guard -- prevent precision divergence
        # If  ² diverges, all PC learning breaks. Guard early.
        for i, col in enumerate(columns):
            sigma2_max = col.L5.W_sigma.bias.data.abs().max().item()
            if sigma2_max > 20.0:  # Emergency:  ² output logit too large
                col.L5.W_sigma.bias.data.mul_(0.9)  # Gently pull back
                col.L5.W_sigma.weight.data.mul_(0.9)
                metrics[f'sigma2_emergency_col{i}'] = True

        # 7. Adjust sleep interval based on developmental stage
        # Less sleep early (more wake learning), more sleep late (consolidation primary)
        dev_sleep_interval = self.development.get_sleep_interval()
        if dev_sleep_interval != self.cfg.sleep_interval:
            self.wake_sleep.cfg.sleep_interval = dev_sleep_interval
            metrics['dev_sleep_interval'] = dev_sleep_interval

        # 8. Track wake step
        self.wake_sleep.on_wake_step()
        metrics['wake_step'] = self.wake_sleep.wake_steps

        return metrics

    def should_sleep(self) -> bool:
        """Check if sleep phase is due."""
        return self.wake_sleep.should_sleep()

    def sleep_phase(
        self,
        model: nn.Module,
        hippocampus,
        columns: nn.ModuleList,
        device: torch.device,
    ) -> Dict[str, float]:
        """
        Complete 3-phase sleep: replay → contrastive consolidation → downscaling.

        Phase 1 -- Generative replay: Run memories through columns, compute PC loss.
                   Goal: cortex learns to predict WITHOUT hippocampus.
        Phase 2 -- Contrastive consolidation: Real memory = potentiate,
                   corrupted memory = depress. Sharpens representations.
        Phase 3 -- Synaptic downscaling (Tononi 2006): global renormalization.
        """
        metrics = {}
        self.wake_sleep.on_sleep_start()

        # Entire sleep phase runs under no_grad -- all learning is analytical
        # (PC local_update, hebbian_update, anti_hebbian_update all use .data).
        # Column forwards during replay/consolidation don't need gradient tracking.
        with torch.no_grad():
            return self._sleep_phase_impl(model, hippocampus, columns, device, metrics)

    def _sleep_phase_impl(self, model, hippocampus, columns, device, metrics):
        """Sleep implementation (always called under torch.no_grad)."""
        # Replay from hippocampal memory
        memories = hippocampus.replay(self.cfg.sleep_replay_steps, device)
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        metrics['replayed_memories'] = len(memories)

        if len(memories) > 0:
            replay_lr = self.cfg.hebbian_lr * 0.5  # Lower LR during sleep
            pc_replay_lr = self.cfg.pc_learning_lr * 0.3  # Even lower for PC replay

            #        Phase 1: Generative replay with PC learning
            # The brain consolidates by making cortex predict WITHOUT hippocampus.
            # We replay memories through columns, compute PC prediction errors,
            # and update prediction weights. This is TRUE consolidation:
            # hippocampal memories → cortical predictions.
            for mem in memories:
                x = mem['x']  # [1, T, D] -- batch-pooled at storage
                error = mem.get('error', torch.zeros_like(x))

                # Run memory through columns (generative replay)
                h = x
                replay_predictions = []
                replay_outputs = []
                for col in columns:
                    out = col(ff_input=h, td_context=None)
                    replay_predictions.append(out.prediction)
                    replay_outputs.append(out)
                    h = out.output

                # PC learning on replayed memories -- consolidation = cortex
                # learns to predict without hippocampus
                replay_targets = [x.detach()] + [
                    replay_outputs[l].output.detach()
                    for l in range(len(replay_outputs) - 1)
                ]
                replay_precisions = [out.sigma2 for out in replay_outputs]
                # Analytical PC update on replay (no autograd)
                self.pc_optimizer.local_update(
                    columns, replay_predictions, replay_targets,
                    replay_precisions
                )

                # Hebbian consolidation (L2/3 associations)
                for col in columns:
                    if col.bcm.enabled:
                        col.L23.hebbian_update_error_modulated(
                            pre_input=x, post_output=x,
                            error=error,
                            bcm_theta=col.bcm.theta_bcm,
                            lr=replay_lr,
                            error_weight=self.cfg.error_hebbian_weight,
                        )
                    else:
                        col.L23.hebbian_update(x, lr=replay_lr)

            #        Phase 2: Contrastive consolidation
            # Real pattern → potentiate. Corrupted pattern → depress.
            # This sharpens representations by creating contrastive signal.
            noise_scale = self.cfg.sleep_contrastive_noise
            contrastive_lr = replay_lr * 0.3  # Even gentler for contrastive

            for mem in memories:
                x = mem['x']  # [1, T, D] -- batch-pooled at storage -- real memory

                # Generate corrupted version (negative)
                noise = torch.randn_like(x) * noise_scale
                x_corrupted = x + noise

                for col in columns:
                    # Anti-Hebbian on corrupted: weakens noisy correlations
                    col.L23.anti_hebbian_update(x_corrupted, lr=contrastive_lr)

                    # Extra Hebbian on real: strengthens true correlations
                    if col.bcm.enabled:
                        col.L23.hebbian_update_bcm(
                            pre_input=x, post_output=x,
                            bcm_theta=col.bcm.theta_bcm,
                            lr=contrastive_lr,
                        )

            metrics['replay_consolidated'] = True
            metrics['contrastive_applied'] = True

        #        Phase 2.5: Hippocampal → Cortical consolidation (McClelland 1995)
        # Extract semantic centroids from clustered episodes, then train cortex
        # on these distilled patterns. This is the CLS transfer: hippocampus →
        # neocortex, converting episodic → semantic memory during SWS.
        semantic_memories = hippocampus.consolidate()
        metrics['semantic_memories_extracted'] = len(semantic_memories)
        if len(semantic_memories) > 0:
            consolidation_lr = self.cfg.hebbian_lr * 0.3
            for sem in semantic_memories:
                pattern = sem['pattern']  # [D]
                # Expand to [1, 1, D] for column processing
                x = pattern.unsqueeze(0).unsqueeze(0).to(device)
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
                # Run through columns -- Hebbian learning on semantic patterns
                h = x
                for col in columns:
                    out = col(ff_input=h, td_context=None)
                    # Hebbian: strengthen cortical representation of semantic pattern
                    if col.bcm.enabled:
                        col.L23.hebbian_update_bcm(
                            pre_input=h, post_output=out.output,
                            bcm_theta=col.bcm.theta_bcm,
                            lr=consolidation_lr,
                        )
                    else:
                        col.L23.hebbian_update(h, out.output, lr=consolidation_lr)
                    h = out.output
            metrics['semantic_consolidation_applied'] = True

        #        Phase 3: Synaptic downscaling (Tononi 2006)
        self.wake_sleep.synaptic_downscaling(model)
        metrics['downscaled'] = True

        # Update sleep count
        metrics['sleep_phase'] = self.wake_sleep.sleep_phases

        return metrics

    def get_status(self) -> dict:
        """Complete learning system status."""
        return self.wake_sleep.get_status()
```

Listing 11: Local Learning Framework — complete source code (v11_learning.py, 1,089 lines)

13 Full Model (CorticalColumnNetwork)

Source: `nct_v2/v11_model.py` **Lines:** 1,025

13.1 Overview

The `CorticalColumnNetwork` class is the top-level module that assembles all NCT components into a single `nn.Module`. It instantiates 13 major subsystems: the brain embedding pipeline (`V11BrainEmbedding` with its 4 stages), grid cell position encoding, N cortical columns (each containing 11 submodules), the thalamic routing system with TRN gating, hippocampal episodic memory with surprise tagging, basal ganglia reward-modulated selection, cerebellar temporal prediction, brainstem neuromodulation (4 signals), lateral column connections, global workspace with surprise-weighted broadcast, self-assessment with ACC conflict monitoring, the brain LM head (weight-tied with embedding), and the integrated learning manager (PC + Hebbian + neuromod + wake-sleep + developmental staging).

The model also maintains internal state for self-repair (best-state rollback if loss diverges), tool use registration (lazy initialization of BG-based tool selector), and stateful tracking of the last forward pass's column outputs, predictions, precisions, errors, and neuromodulatory state — enabling local learning updates without re-running the forward pass.

13.2 V11BrainEmbedding

The `V11BrainEmbedding` class wraps the 4-stage brain embedding pipeline into a single module compatible with the V11 config interface. It chains the four stages in sequence:

$$\mathbf{e} = \underbrace{\text{SparseDistributed}}_{\text{Stage 4}} \circ \underbrace{\text{HubSpoke}}_{\text{Stage 3}} \circ \underbrace{\text{MorphemeGate}}_{\text{Stage 2}} \circ \underbrace{\text{BaseEmbed}}_{\text{Stage 1}}(\text{tokens}) \tag{76}$$

The `weight` property returns the base embedding weights (W_{embed} from Stage 1) for weight tying with the LM head. The base embedding is initialized with $\mathcal{N}(0, 0.02)$; all subsequent stages are identity-at-init and progressively activate during training as their gates learn to open.

13.3 Forward Pass — 11 Stages

Theorem 13.1 (Complete Forward Pass)**.** *The forward pass proceeds through 11 stages:*

Stage 1 — Embedding: $\mathbf{x} = \textit{Dropout}(\textit{RMSNorm}(\textit{BrainEmbed}(\textit{tokens})))$

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

Stage 2 — Position: $\mathbf{x} \leftarrow \mathbf{x} + 0.1 \cdot \mathit{GridCellPos}(\mathbf{x})$
Stage 3 — Bottom-up pass: *For each column* $i = 0, \ldots, N-1$*:*

$$\mathbf{h}_{thal} = \mathit{Thalamus.route}(\mathbf{h}, i, \mathbf{f}_{thal,i}), \qquad \mathit{out}_i = \mathit{Column}_i(\mathbf{h}_{thal}, \mathit{td}{=}\mathit{None}, \sigma^2_{prev}) \tag{77}$$

Stage 4 — Predictions: *Column 0 predicts embedding* $\mathbf{x}$*; column* i *predicts column* $i-1$*'s output.*
Stage 5 — Errors: $\boldsymbol{\epsilon}_i = \mathit{target}_i - \mathit{prediction}_i$ *for each column.*
Stage 6 — Lateral + Workspace: *Compute lateral signals between adjacent columns; global workspace broadcasts highest-surprise column.*
Stage 7 — Metacognition: *Self-assessment computes conflict → extra PC steps.*
Stage 8 — PC inference (no gradients): *Re-run all columns with top-down + workspace + lateral context for up to* $K_{\max} + K_{extra} - 1$ *additional iterations. Break early when relative error change* $< \tau_{conv}$*.*
Stage 9 — Neuromodulation: *Brainstem computes DA/NE/5-HT/ACh from top column output.*
Stage 10 — Subcortical: *BG modulates output with DA; cerebellum computes temporal prediction loss; hippocampus adds episodic memory and tags surprises.*
Stage 11 — Output: *logits =* $\mathit{BrainLMHead}(\mathit{RMSNorm}(\mathbf{h}))$*. NTP loss via cross-entropy. PC loss summed over columns.*

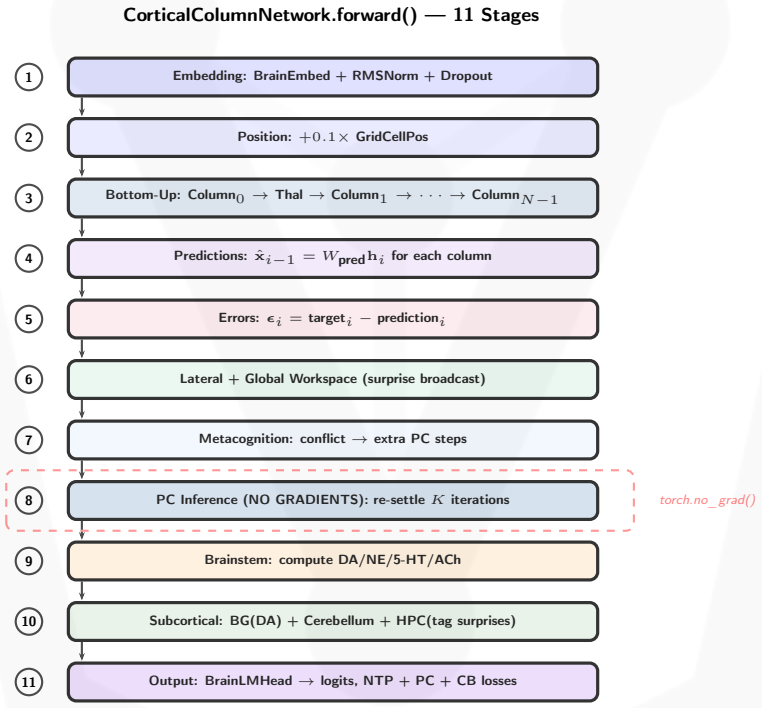


Figure 63: The 11-stage forward pass of CorticalColumnNetwork. Stage 8 (PC inference) runs without gradients — it is dynamical relaxation, not gradient descent. This saves $O(N \times K)$ activation memory.

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

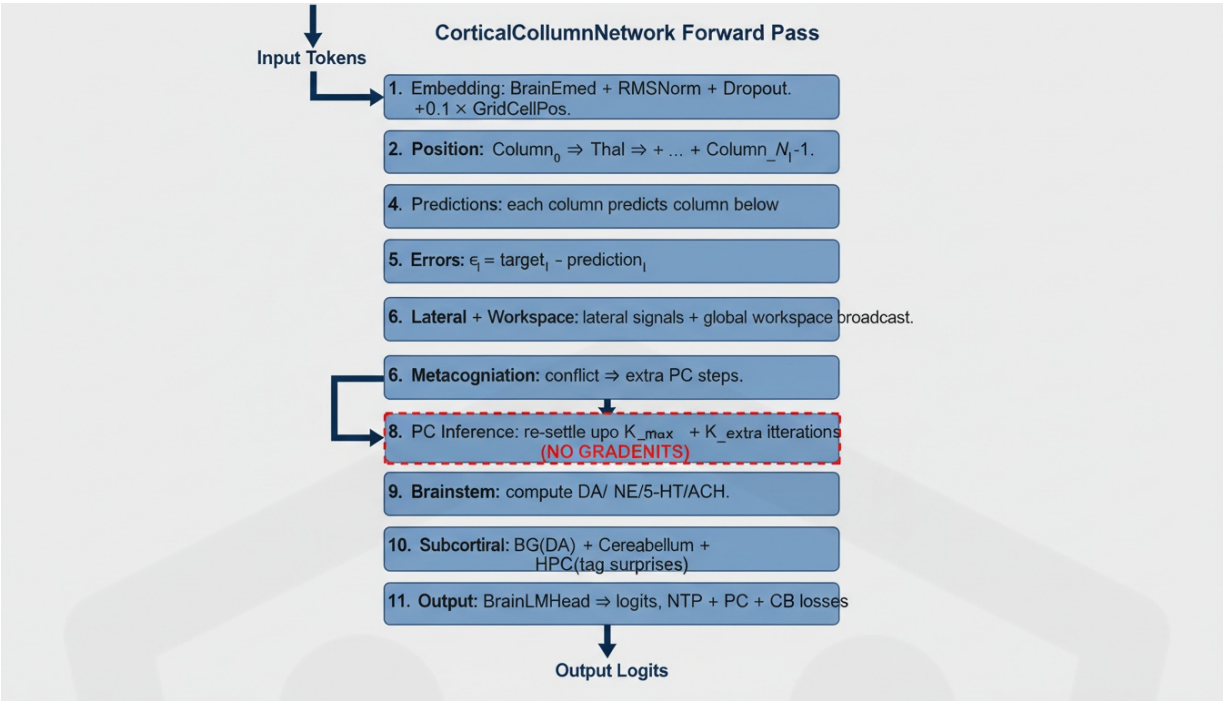


Figure 64: (Illustrated) The 11-stage forward pass of CorticalColumnNetwork. Stage 8 (PC inference) runs without gradients — it is dynamical relaxation, not gradient descent. This saves $O(N \times K)$ activation memory.

13.4 Loss Function

The total loss combines four terms:

$$\mathcal{L} = \underbrace{\mathcal{L}_{\text{NTP}}}_{\text{cross-entropy}} + \underbrace{\sum_{i=0}^{N-1} \frac{1}{\sigma_i^2} \|\boldsymbol{\epsilon}_i\|^2}_{\mathcal{L}_{\text{PC}}} + \underbrace{0.1 \cdot \|\mathbf{h}_t - \hat{\mathbf{h}}_t\|^2}_{\mathcal{L}_{\text{CB}}} + \underbrace{0.1 \cdot \frac{1}{G} \sum_g r_g(1 - r_g)}_{\mathcal{L}_{\text{TRN}}} \tag{78}$$

In PC/Hebbian modes, $\mathcal{L}_{\text{NTP}}$ gradients propagate **only to the LM head** (not through columns). All column weight updates come from local PC/Hebbian rules.

13.5 Self-Repair Mechanism

Definition 13.1 (Self-Repair via Best-State Rollback)**.** *The model periodically saves its best state (lowest loss seen). If the current loss exceeds* $2\times$ *the best loss, the model:*

1. *Rolls back all parameters to the saved best state*
2. *Reduces the learning rate by 50%*
3. *Increments the rollback counter*

This prevents catastrophic divergence during continuous learning — a form of metacognitive self-regulation absent in transformers.

13.6 Key Methods

The model provides 11+ methods beyond `forward()`:

`local_learning_step()`: Apply PC + Hebbian updates using stored forward-pass state (column outputs, predictions, errors, precisions, neuromod state). No additional forward pass needed.

`sleep_phase()`: Hippocampal replay + synaptic downscaling ($W \leftarrow 0.99W$). Uses stored HPC memories.

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

`generate(prompt, max_new_tokens)`: Auto-regressive text generation with optional brain decoding (drift-diffusion sampler, inhibition of return, σ^2-adaptive temperature).

`count_parameters()`: Per-component parameter breakdown (embedding, columns, thalamus, HPC, BG, CB, brainstem, LM head, grid pos, lateral, workspace, self-assessment).

`register_tools(schemas)`: Register tool schemas for agentic tool use via the basal ganglia D1/D2 pathways.

`get_gate_values()`: Aggregate gate values from all columns for monitoring.

`reset_working_memory()`: Clear all persistent WM state (NMDA, silent, phono) across all columns.

13.7 Source Code

```
1
2
3
4  # ========================================================================
5  # Cortical Column Network -- Full Model
6  # Source: nct_v2/v11_model.py
7  # ========================================================================
8
9  """
10 NCT V11: Cortical Column Network -- Full Model
11 ==============================================
12
13 The complete brain-analog architecture replacing the transformer.
14
15 Architecture:
16   Embedding → Thalamus → [Column 0 → Column 1 → ... → Column N-1] → LM Head
17
18   Each column: L4 → L1 → L2/3 → Interneurons → WM → Oscillations → L5 → L6
19
20   Subcortical: Thalamus (routing), Hippocampus (memory), Basal Ganglia (reward),
21                Cerebellum (temporal prediction), Brainstem (neuromodulation)
22
23   Learning: Predictive coding (vertical) + Hebbian (lateral) + Neuromod gating
24   Operation: Continuous wake-sleep cycle
25
26 Forward pass:
27   1. Bottom-up: input → embed → columns[0] → ... → columns[N-1] → logits
28   2. Top-down predictions: each column predicts the level below
29   3. Prediction errors:  _l = actual_l - prediction_{l+1}
30   4. Inference: iterate representations to minimize errors (PC)
31   5. Learning: local updates (PC + Hebbian + neuromod-gated)
32 """
33
34
35
36 # [bundled] from .v11_config import V11Config
37 # [bundled] from .v11_column import CorticalColumn, ColumnOutput, RMSNorm, LateralColumnConnection
38 # [bundled] from .v11_systems import (
39 # [bundled] from .v11_learning import V11LearningManager
40 # [bundled] from .brain_embeddings import HubAndSpokeProjection, SparseDistributedProjection, MorphemeCompositionGate
41 # [bundled] from .brain_position import GridCellPositionEncoding
42 # [bundled] from .brain_lm_head import BrainLMHead
43 # [bundled] from .brain_decoding import DriftDiffusionSampler, InhibitionOfReturn, BrainDecodingConfig
44
45
46 #
47 # V11 Brain Embedding (Theorems 56-58)
48 #
49
50 class V11BrainEmbedding(nn.Module):
51     """Brain-inspired embedding replacing nn.Embedding (Thm 56-58).
52
53     4-stage cascade matching brain processing timeline:
54       Stage 1 (0ms):   Token ID → base embedding lookup
55       Stage 2 (170ms): Morphological composition -- dual-route lexical/sublexical (Thm 58)
56       Stage 3 (250ms): Hub-spoke cross-modal integration -- ATL binding (Thm 56)
57       Stage 4 (300ms): Sparse distributed activation -- neocortical coding (Thm 57)
58
59     All stages are gated and identity-at-init.
60     Weight tying: lm_head ties to self.base_embed.weight.
61     """
62
63     def __init__(self, cfg: V11Config):
64         super().__init__()
65         d = cfg.d_model
66         V = cfg.vocab_size
67
68         # Stage 1: Base embedding (standard lookup)
69         self.base_embed = nn.Embedding(V, d)
70
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        # Stage 2: Dual-route morphological composition (Thm 58)
        self.morpheme_gate = MorphemeCompositionGate(V, d, kernel_size=3)

        # Stage 3: Hub-and-spoke semantic integration (Thm 56)
        self.hub_spoke = HubAndSpokeProjection(d, S=6)

        # Stage 4: Sparse distributed activation (Thm 57)
        self.sparse_proj = SparseDistributedProjection(d, expansion=2, sparsity=0.02)

        nn.init.normal_(self.base_embed.weight, std=0.02)

    @property
    def weight(self):
        """For weight tying with lm_head: return base embedding weights."""
        return self.base_embed.weight

    def forward(self, token_ids: torch.Tensor) -> torch.Tensor:
        """token_ids: [B, T] → [B, T, d_model]"""
        e = self.base_embed(token_ids)
        e = self.morpheme_gate(e, token_ids)
        e = self.hub_spoke(e)
        e = self.sparse_proj(e)
        return e


#

# Cortical Column Network (Full Model)
#


class CorticalColumnNetwork(nn.Module):
    """
    The complete NCT V11 model: a cortical column network.

    Replaces the transformer with a biologically-grounded architecture.
    """

    def __init__(self, cfg: V11Config):
        super().__init__()
        self.cfg = cfg

        #        Token Embedding (Thm 56-58: Brain Embedding)
        self.embed = V11BrainEmbedding(cfg)
        self.embed_norm = RMSNorm(cfg.d_model)
        self.embed_dropout = nn.Dropout(cfg.dropout)

        #        Positional Encoding (Thm 59/61/62: Grid-Cell)
        # Learnable grid-cell frequencies, per-head scales, boundary remapping
        self.grid_pos = GridCellPositionEncoding(
            max_seq_len=cfg.max_seq_len,
            d_h=cfg.d_head,
            n_heads=cfg.n_assoc_heads,
            d_model=cfg.d_model,
        )

        #        Cortical Columns (hierarchical stack)
        self.columns = nn.ModuleList([
            CorticalColumn(cfg, column_idx=i) for i in range(cfg.n_columns)
        ])

        #        Subcortical Systems

        self.thalamus = Thalamus(cfg)
        self.hippocampus = Hippocampus(cfg)
        self.basal_ganglia = BasalGanglia(cfg)
        self.cerebellum = Cerebellum(cfg)
        self.brainstem = Brainstem(cfg)

        #        LM Head (Thm 63: Brain Lexical Production)
        # WEAVER++ competition + production delta + frequency bias
        self.lm_head_norm = RMSNorm(cfg.d_model)
        self.lm_head = BrainLMHead(cfg.d_model, cfg.vocab_size, rank=32, competition_rank=64)
        # Tie embeddings: BrainLMHead uses base embedding weights for shared semantics
        self.lm_head.weight = self.embed.weight

        #        Lateral Column Connections (Horizontal)
        self.lateral = LateralColumnConnection(cfg)

        #        Global Workspace (Baars 1988)

        self.workspace = CorticalWorkspace(cfg)

        #        Self-Assessment & Metacognition

        self.self_assessment = SelfAssessment(cfg)

        #        Tool Interface (Thm 71: BG D1/D2 Tool Selection)
        # Initialized lazily via register_tools(). Allows world interaction.
        self._tool_selector: Optional[nn.Module] = None
        self._tool_schemas: List = []
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        #        Learning Manager

        self.learning_manager = V11LearningManager(cfg)

        #        State tracking

        self._last_column_outputs: Optional[List[ColumnOutput]] = None
        self._last_predictions: Optional[List[torch.Tensor]] = None
        self._last_targets: Optional[List[torch.Tensor]] = None
        self._last_precisions: Optional[List[torch.Tensor]] = None
        self._last_neuromod: Optional[NeuromodState] = None
        self._last_errors: Optional[List[torch.Tensor]] = None

        #        Self-repair: best-state rollback

        # Save best state periodically. If loss diverges, roll back.
        self._best_state: Optional[dict] = None
        self._best_loss: float = float('inf')
        self._rollback_count: int = 0
        self._steps_since_best: int = 0

        # Initialize
        self._init_weights()

    def _init_weights(self):
        """Initialize weights following NCT principles."""
        # Embedding init handled by V11BrainEmbedding.__init__
        pass

    def count_parameters(self) -> Dict[str, int]:
        """Count parameters by component."""
        counts = {}
        counts['embedding'] = sum(p.numel() for p in self.embed.parameters())
        counts['columns'] = sum(p.numel() for p in self.columns.parameters())
        counts['thalamus'] = sum(p.numel() for p in self.thalamus.parameters())
        counts['hippocampus'] = sum(p.numel() for p in self.hippocampus.parameters())
        counts['basal_ganglia'] = sum(p.numel() for p in self.basal_ganglia.parameters())
        counts['cerebellum'] = sum(p.numel() for p in self.cerebellum.parameters())
        counts['brainstem'] = sum(p.numel() for p in self.brainstem.parameters())
        counts['lm_head'] = sum(p.numel() for p in self.lm_head_norm.parameters()) + \
                            sum(p.numel() for p in self.lm_head.parameters())
        counts['grid_pos'] = sum(p.numel() for p in self.grid_pos.parameters())
        counts['lateral'] = sum(p.numel() for p in self.lateral.parameters())
        counts['workspace'] = sum(p.numel() for p in self.workspace.parameters())
        counts['self_assessment'] = sum(p.numel() for p in self.self_assessment.parameters())
        counts['total'] = sum(p.numel() for p in self.parameters())
        counts['trainable'] = sum(p.numel() for p in self.parameters() if p.requires_grad)
        return counts

    #

    # Forward Pass
    #


    def forward(
        self,
        input_ids: torch.Tensor,
        targets: Optional[torch.Tensor] = None,
    ) -> Dict[str, torch.Tensor]:
        """
        Full forward pass through the cortical column network.

        Args:
          input_ids: [B, T] -- token IDs
          targets: [B, T] -- target token IDs (for NTP loss)

        Returns: dict with:
          logits: [B, T, V]
          loss: scalar (if targets provided)
          pc_loss: scalar (predictive coding loss)
          cb_loss: scalar (cerebellar prediction loss)
          metrics: dict of monitoring values
        """
        B, T = input_ids.shape
        device = input_ids.device

        #        1. Embedding (Thm 56-58: Brain Embedding)
        x = self.embed(input_ids)  # [B, T, D] -- morpheme → hub-spoke → sparse
        x = self.embed_norm(x)
        x = self.embed_dropout(x)

        #        1b. Position Information

        # L2/3 now uses Windowed Hebbian Association -- no RoPE needed.
        # Position is encoded through THREE brain mechanisms:
        #   1. Windowed context: implicit local position within theta period
        #   2. Oscillatory dynamics: nested PAC gives absolute phase position
        #   3. Grid-cell encoding: additive position signal to embeddings
        # Grid-cell features are added (not rotated) since there's no attention.
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        rope_cos, rope_sin = self.grid_pos(T, device, x.dtype, X=x)
        # Additive position encoding: use sin component as position signal
        # Grid-cell output: [B, H, T, D_h] → average over heads → [B, T, D_h]
        if rope_sin.dim() == 4:  # [B, H, T, D_h]
            pos_signal = rope_sin.mean(dim=1)  # [B, T, D_h]
        elif rope_sin.dim() == 3:  # [H, T, D_h]
            pos_signal = rope_sin.mean(dim=0).unsqueeze(0)  # [1, T, D_h]
        else:
            pos_signal = rope_sin[:T].unsqueeze(0)  # [1, T, D_h]
        d_h = pos_signal.shape[-1]
        d_model = x.shape[-1]
        # Tile to match d_model: repeat to fill
        n_tiles = d_model // d_h
        if n_tiles > 1 and d_h * n_tiles == d_model:
            pos_embed = pos_signal.repeat(1, 1, n_tiles)  # [B, T, D_model]
        elif d_h < d_model:
            pos_embed = F.pad(pos_signal, (0, d_model - d_h))  # [B, T, D_model]
        else:
            pos_embed = pos_signal[..., :d_model]  # [B, T, D_model]
        x = x + 0.1 * pos_embed  # Scaled additive position (small at init)

        # Reset TRN loss accumulator for this forward pass
        if self.cfg.thalamic_enabled:
            self.thalamus.reset_loss()

        #        2. Bottom-Up Pass (through all columns)
        column_outputs: List[ColumnOutput] = []
        thal_feedbacks: List[Optional[torch.Tensor]] = [None] * self.cfg.n_columns

        h = x
        sigma2_prev = None  # First column has no prior precision
        for i, col in enumerate(self.columns):
            # Route through thalamus (with L6 feedback from previous pass if available)
            h_thal = self.thalamus.route(h, column_idx=i, thal_feedback=thal_feedbacks[i])

            # Column forward (no RoPE -- L2/3 is windowed Hebbian, not attention)
            out = col(ff_input=h_thal, td_context=None, sigma2_prev=sigma2_prev)
            column_outputs.append(out)

            # Store thalamic feedback for inference iterations
            thal_feedbacks[i] = out.thal_feedback
            sigma2_prev = out.sigma2  # Pass precision to next column's L5

            h = out.output

        #        3. Compute Top-Down Predictions

        # Each column predicts the input to the column below
        predictions = []
        prediction_targets = []
        prediction_precisions = []

        for l in range(self.cfg.n_columns):
            if l == 0:
                # Column 0 predicts the embedding
                prediction_targets.append(x.detach())
            else:
                # Column l's input is column l-1's output
                prediction_targets.append(column_outputs[l - 1].output.detach())

            predictions.append(column_outputs[l].prediction)
            prediction_precisions.append(column_outputs[l].sigma2)

        #        4. Compute Prediction Errors

        errors = []
        for l in range(self.cfg.n_columns):
            error = prediction_targets[l] - predictions[l]
            errors.append(error)

        #        5a. Lateral Column Communication (Horizontal)
        # Adjacent columns exchange information directly (not just via hierarchy)
        lateral_signals = self.lateral(column_outputs)

        #        5a2. Global Workspace Broadcast (Baars 1988)
        # Highest-surprise column broadcasts to all -- provides global context
        ws_modulations = self.workspace.broadcast(column_outputs, errors)

        #        5b. Metacognitive Monitoring (ACC-like conflict)
        # When columns disagree, allocate more inference steps (think harder)
        conflict = self.self_assessment.compute_conflict(column_outputs)
        extra_steps = self.self_assessment.extra_pc_steps(conflict)

        #        5c. Predictive Coding Inference (adaptive convergence)
        # Brain iterates until prediction errors settle. We iterate up to
        # pc_max_steps + extra_steps (from conflict), breaking early once
        # error change < threshold.
        # Workspace modulations provide global top-down context per column.
        pc_steps_taken = 1  # Already did 1 pass above
        prev_error_norm = sum(e.pow(2).mean() for e in errors)

```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        max_steps = self.cfg.pc_max_steps + extra_steps
        min_steps = self.cfg.pc_min_steps
        conv_threshold = self.cfg.pc_convergence_threshold

        # PC inference iterations run WITHOUT gradients -- biologically correct.
        # Neural settling is dynamical relaxation, not gradient descent.
        # Gradients are only needed for the FINAL state (computed after this loop).
        # PC weight updates use analytical local gradients (PredictiveCodingOptimizer),
        # not autograd. This saves O(N_columns * N_steps) activation memory.
        with torch.no_grad():
            for step in range(max_steps - 1):
                h = x
                # Re-run columns with top-down + workspace + lateral + L6 feedback
                new_outputs = []
                sigma2_prev_iter = None
                for i, col in enumerate(self.columns):
                    # Top-down context = workspace + lateral + prediction from above
                    td_ctx = ws_modulations[i] + lateral_signals[i]
                    if i < self.cfg.n_columns - 1:
                        td_ctx = td_ctx + column_outputs[i + 1].prediction

                    # Route through thalamus WITH L6 feedback from previous iteration
                    h_thal = self.thalamus.route(
                        h, column_idx=i, thal_feedback=thal_feedbacks[i]
                    )
                    out = col(ff_input=h_thal, td_context=td_ctx,
                              sigma2_prev=sigma2_prev_iter)
                    new_outputs.append(out)
                    thal_feedbacks[i] = out.thal_feedback  # Update L6 feedback
                    sigma2_prev_iter = out.sigma2
                    h = out.output

                # Update stored outputs
                column_outputs = new_outputs

                # Recompute predictions and errors
                predictions = [out.prediction for out in column_outputs]
                prediction_precisions = [out.sigma2 for out in column_outputs]
                errors = [
                    prediction_targets[l] - predictions[l]
                    for l in range(self.cfg.n_columns)
                ]

                # Re-compute lateral + workspace with updated outputs/errors
                lateral_signals = self.lateral(column_outputs)
                ws_modulations = self.workspace.broadcast(column_outputs, errors)

                pc_steps_taken += 1

                # Check convergence: has the total error stopped changing?
                # Use RELATIVE threshold (error_delta / prev_error < threshold)
                # to avoid the absolute threshold never triggering on large errors.
                curr_error_norm = sum(e.pow(2).mean() for e in errors)
                error_delta = abs(prev_error_norm - curr_error_norm)
                if isinstance(error_delta, torch.Tensor):
                    error_delta = error_delta.item()
                if isinstance(prev_error_norm, torch.Tensor):
                    prev_val = prev_error_norm.item()
                else:
                    prev_val = prev_error_norm
                # Relative convergence: < 1% change in error
                relative_delta = error_delta / (prev_val + 1e-8)
                prev_error_norm = curr_error_norm

                # Early stop: converged after minimum steps
                if pc_steps_taken >= min_steps and relative_delta < conv_threshold:
                    break

        #         6. Brainstem: Compute Neuromodulatory State
        # Use top column's output for global state computation
        top_output = column_outputs[-1].output
        mean_error = torch.stack([e.abs().mean() for e in errors]).mean()
        neuromod = self.brainstem.compute_state(
            top_output,
            prediction_error=errors[-1] if errors else None,
        )

        #         7. Basal Ganglia: DA modulation

        if self.cfg.bg_enabled:
            h = self.basal_ganglia(column_outputs[-1].output, neuromod.dopamine)
            # Update tonic DA EMA (was never called -- now wired)
            self.basal_ganglia.update_tonic_da(neuromod.dopamine)
        else:
            h = column_outputs[-1].output

        #         8. Cerebellum: Temporal prediction
        cb_loss = torch.tensor(0.0, device=device)
        if self.cfg.cerebellum_enabled:
            h, cb_loss = self.cerebellum(h)

```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        #         9. Hippocampus: Episodic memory

        if self.cfg.hpc_enabled:
            # Use mean precision from all columns
            mean_sigma2 = sum(out.sigma2 for out in column_outputs) / len(column_outputs)
            h = self.hippocampus(h, mean_sigma2)

            # Tag surprising events for replay
            if self.training:
                self.hippocampus.encode_episode(
                    h, errors[-1] if errors else torch.zeros_like(h),
                    mean_sigma2,
                )

        #         10. LM Head

        h_normed = self.lm_head_norm(h)
        logits = self.lm_head(h_normed)  # [B, T, V]

        #         11. Compute Losses

        result = {'logits': logits, 'hidden_state': h_normed.detach(), 'targets': targets}

        # NTP loss (standard cross-entropy)
        if targets is not None:
            ntp_loss = F.cross_entropy(
                logits.reshape(-1, self.cfg.vocab_size),
                targets.reshape(-1),
                ignore_index=-1,
            )
            result['loss'] = ntp_loss

        # Predictive coding loss (local)
        _, pc_loss = self.learning_manager.pc_optimizer.compute_column_losses(
            predictions, prediction_targets, prediction_precisions
        )
        result['pc_loss'] = pc_loss
        result['cb_loss'] = cb_loss

        # TRN bimodality loss (accumulated during thalamic routing)
        trn_loss = self.thalamus.compute_loss() if self.cfg.thalamic_enabled else torch.tensor(0.0, device=device)
        result['trn_loss'] = trn_loss

        #         12. Store state for learning

        self._last_column_outputs = column_outputs
        self._last_predictions = predictions
        self._last_targets = prediction_targets
        self._last_precisions = prediction_precisions
        self._last_neuromod = neuromod
        self._last_errors = errors

        #         13. Self-Assessment Health Check
        self.self_assessment.update_health(column_outputs, pc_steps_taken)
        if self.training and self.self_assessment._health_updates % 100 == 0:
            health = self.self_assessment.check_health(self.columns)

        #         14. Metrics

        metrics = {}
        metrics['pc_steps'] = pc_steps_taken
        metrics['conflict'] = conflict
        metrics['mean_pred_error'] = mean_error.item() if isinstance(mean_error, torch.Tensor) else mean_error
        metrics['da_mean'] = neuromod.dopamine.mean().item()
        metrics['ne_mean'] = neuromod.norepinephrine.mean().item()
        metrics['ach_mean'] = neuromod.acetylcholine.mean().item()
        metrics['tonic_da'] = neuromod.tonic_da
        metrics['hpc_memory_size'] = self.hippocampus.memory_size if self.cfg.hpc_enabled else 0

        # Precision stats
        all_sigma2 = torch.cat([out.sigma2 for out in column_outputs], dim=-1)
        metrics['sigma2_mean'] = all_sigma2.mean().item()
        metrics['sigma2_std'] = all_sigma2.std().item()

        # Gate values from first and last columns
        metrics.update(self.columns[0].get_gate_values())
        metrics.update(self.columns[-1].get_gate_values())

        # Self-assessment metrics
        metrics.update(self.self_assessment.get_metrics())

        result['metrics'] = metrics

        return result

    #

    # Learning Interface
    #


```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```python
    def local_learning_step(self, ewc_guard=None) -> Dict[str, float]:
        """
        Perform local learning (PC + Hebbian + neuromod).
        Must be called after forward() which stores the needed state.

        Args:
            ewc_guard: Optional ContinualLearningGuard for EWC elastic pull.

        Returns: dict of learning metrics
        """
        if self._last_column_outputs is None:
            return {}

        return self.learning_manager.wake_step(
            columns=self.columns,
            column_outputs=self._last_column_outputs,
            predictions=self._last_predictions,
            targets=self._last_targets,
            precisions=self._last_precisions,
            neuromod=self._last_neuromod,
            errors=self._last_errors,
            ewc_guard=ewc_guard,
        )

    def sleep_phase(self) -> Dict[str, float]:
        """
        Run sleep phase: replay + downscaling + consolidation.
        Called when wake_sleep scheduler says it's time.

        Returns: dict of sleep metrics
        """
        device = next(self.parameters()).device
        return self.learning_manager.sleep_phase(
            model=self,
            hippocampus=self.hippocampus,
            columns=self.columns,
            device=device,
        )

    def learn_from_experience(self, input_ids: torch.Tensor, targets: Optional[torch.Tensor] = None) -> Dict[str, float]:
        """
        Process AND learn from a single experience in one call.

        This is the "always-on learning" interface: the brain doesn't separate
        forward() from learning. Every experience updates the system.

        Returns: combined forward + learning metrics.
        """
        result = self.forward(input_ids, targets)
        learn_metrics = self.local_learning_step()
        result['learning_metrics'] = learn_metrics
        return result

    def should_sleep(self) -> bool:
        """Check if it's time for sleep."""
        return self.learning_manager.should_sleep()

    #

    # Backprop Mode (for comparison / hybrid training)
    #


    def backprop_step(
        self,
        input_ids: torch.Tensor,
        targets: torch.Tensor,
        optimizer: torch.optim.Optimizer,
    ) -> Dict[str, float]:
        """
        Standard backprop training step (for comparison or hybrid mode).
        Uses global loss and standard optimizer.

        Returns: dict with loss, ppl, etc.
        """
        result = self.forward(input_ids, targets)
        loss = result['loss']

        # Add auxiliary losses: PC + cerebellar + TRN bimodality
        total_loss = loss + 0.1 * result['pc_loss'] + 0.01 * result['cb_loss'] + 0.01 * result['trn_loss']

        optimizer.zero_grad()
        total_loss.backward()

        # Gradient clipping
        nn.utils.clip_grad_norm_(self.parameters(), self.cfg.max_grad_norm)

        optimizer.step()

        return {
            'loss': loss.item(),
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
            'pc_loss': result['pc_loss'].item(),
            'cb_loss': result['cb_loss'].item(),
            'trn_loss': result['trn_loss'].item(),
            'ppl': math.exp(min(loss.item(), 20)),
            **result['metrics'],
        }

    #

    # Generation
    #


    @torch.no_grad()
    def generate(
        self,
        input_ids: torch.Tensor,
        max_new_tokens: int = 100,
        use_brain_decoding: bool = True,
    ) -> torch.Tensor:
        """
        Brain-inspired autoregressive generation (Thm 63d-65).

        Uses drift-diffusion evidence accumulation + inhibition of return
        + System 1/2 confidence gating. All controlled by NCT signals:
          -  ² → noise modulation (uncertain → more deliberation)
          - Tonic DA → threshold (high DA → lower threshold → explore)
          - IOR → repetition avoidance (decaying inhibition on recent tokens)

        Falls back to standard temperature sampling if use_brain_decoding=False.

        Args:
          input_ids: [B, T] -- prompt tokens
          max_new_tokens: number of tokens to generate
          use_brain_decoding: use drift-diffusion (True) or standard sampling (False)

        Returns: [B, T + max_new_tokens] -- full sequence with generated tokens
        """
        self.eval()
        generated = input_ids.clone()

        if use_brain_decoding:
            # Initialize brain decoding components
            decode_cfg = BrainDecodingConfig()
            sampler = DriftDiffusionSampler(decode_cfg)
            ior = InhibitionOfReturn(self.cfg.vocab_size, decode_cfg.ior_strength, decode_cfg.ior_tau)
            ior.reset(input_ids.device)

            for _ in range(max_new_tokens):
                context = generated[:, -self.cfg.max_seq_len:]
                result = self.forward(context)
                logits = result['logits'][:, -1, :].float()  # [B, V]

                # Extract NCT signals for modulation
                sigma2 = result['metrics'].get('sigma2_mean', 1.0)
                tonic_da = self.brainstem.tonic_da.item()
                gen_conflict = result['metrics'].get('conflict', 0.0)

                # Drift-diffusion sampling with IOR + conflict modulation
                selected, _ = sampler.sample(
                    logits=logits,
                    sigma2=sigma2,
                    tonic_da=tonic_da,
                    conflict=gen_conflict,
                    ior_trace=ior.trace,
                )

                # Update IOR trace
                ior.step(selected[0].item())
                generated = torch.cat([generated, selected.unsqueeze(1)], dim=1)
        else:
            # Standard temperature/top-p fallback
            for _ in range(max_new_tokens):
                context = generated[:, -self.cfg.max_seq_len:]
                result = self.forward(context)
                logits = result['logits'][:, -1, :] / 0.8  # temperature
                probs = F.softmax(logits.float(), dim=-1)
                next_token = torch.multinomial(probs, num_samples=1)
                generated = torch.cat([generated, next_token], dim=1)

        self.train()
        return generated

    #

    # Tool Interface (Blocker 5: World Interaction)
    #


    def register_tools(self, tool_schemas: list):
        """
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        Register available tools for BG D1/D2 selection (Thm 71).

        Args:
            tool_schemas: List of ToolSchema objects from nct_v2/tool_use.py

        After registration, generate_with_tools() can select and call tools.
        This is the architecture-level wiring -- training data teaches the
        model WHEN to use each tool.
        """
        # [bundled] from .tool_use import ToolSelector
        self._tool_schemas = tool_schemas
        n_tools = len(tool_schemas)
        if n_tools == 0:
            return

        device = next(self.parameters()).device
        self._tool_selector = ToolSelector(
            d_model=self.cfg.d_model,
            max_tools=max(n_tools, 4),
            d_tool=min(128, self.cfg.d_model // 4),
        ).to(device)

    @torch.no_grad()
    def generate_with_tools(
        self,
        input_ids: torch.Tensor,
        tool_executor: callable = None,
        max_new_tokens: int = 200,
        max_tool_calls: int = 5,
    ) -> dict:
        """
        Generate text with optional tool calls.

        The brain's agentic loop (Thm 73):
          1. Generate tokens until tool-call detected or EOS
          2. BG D1/D2 decides: call tool or continue generating
          3. If tool called: execute, inject result, continue generating
          4. Repeat until done or max_tool_calls exhausted

        Args:
            input_ids: [1, T] prompt tokens
            tool_executor: callable(tool_name, arguments) -> str
            max_new_tokens: max tokens to generate
            max_tool_calls: max number of tool invocations

        Returns:
            dict with 'generated_ids', 'text_segments', 'tool_calls', 'tool_results'
        """
        # [bundled] from .tool_use import ToolCallFormat

        self.eval()
        generated = input_ids.clone()
        tool_calls_made = []
        tool_results = []
        tokens_generated = 0

        for _ in range(max_new_tokens):
            context = generated[:, -self.cfg.max_seq_len:]
            result = self.forward(context)
            logits = result['logits'][:, -1, :].float()

            # Standard sampling
            probs = F.softmax(logits / 0.7, dim=-1)
            next_token = torch.multinomial(probs, num_samples=1)
            generated = torch.cat([generated, next_token], dim=1)
            tokens_generated += 1

            # Check if tool selector wants to intervene
            if (self._tool_selector is not None
                    and tool_executor is not None
                    and len(tool_calls_made) < max_tool_calls):
                hidden = result['logits']  # Use logits-level representation
                h_pool = result.get('hidden_state', logits)
                if h_pool is not None and h_pool.dim() == 3:
                    h_pool = h_pool[:, -1:, :]  # Last position
                    sigma2 = result['metrics'].get('sigma2_mean', 1.0)
                    tonic_da = self.brainstem.tonic_da.item()
                    selection = self._tool_selector(
                        h_pool, sigma2=sigma2, tonic_da=tonic_da,
                        n_active_tools=len(self._tool_schemas),
                    )
                    if selection['selected_tool'] is not None:
                        tool_idx = selection['selected_tool']
                        if tool_idx < len(self._tool_schemas):
                            schema = self._tool_schemas[tool_idx]
                            # Execute tool
                            try:
                                tool_result = tool_executor(schema.name, {})
                                tool_calls_made.append(schema.name)
                                tool_results.append(tool_result)
                            except Exception:
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
                            pass

        self.train()
        return {
            'generated_ids': generated,
            'tokens_generated': tokens_generated,
            'tool_calls': tool_calls_made,
            'tool_results': tool_results,
        }

    #

    # Persistence -- Save/Load ALL Brain State
    #


    def save_state(self, path: str):
        """
        Save the COMPLETE brain state to disk.

        Unlike a standard model checkpoint that only saves weights, this saves
        everything a living brain needs to resume exactly where it left off:
          - All model parameters and buffers (nn.Module state_dict)
          - Hippocampal episodic memory buffer (CPU tensors)
          - BCM homeostatic thresholds (per-column sliding thresholds)
          - Metaplasticity EMA (learning-rate-of-learning-rate state)
          - Developmental program step (which critical period stage)
          - Learning manager internal state (wake-sleep counters)
          - Lateral connection state (already in state_dict)
          - Config (to recreate architecture)

        The autonomous loop state (EWC importance, performance history,
        consolidation knowledge) is saved separately via save_autonomous_state()
        since the autonomous loop wraps the model.

        Usage:
          model.save_state('brain_checkpoint.pt')
          # ... later ...
          model = CorticalColumnNetwork(cfg)
          model.load_state('brain_checkpoint.pt')
        """
        state = {
            # 1. All nn.Module parameters and buffers
            'model_state_dict': self.state_dict(),

            # 2. Hippocampal episodic memory (non-parameter state)
            'hippocampal_memories': [
                {k: v.clone() for k, v in mem.items()}
                for mem in self.hippocampus._memory_buffer
            ],
            'hippocampal_surprises': list(self.hippocampus._surprise_buffer),

            # 3. Developmental program step
            'developmental_step': self.learning_manager.development._current_step,

            # 4. Wake-sleep scheduler state
            'wake_steps': self.learning_manager.wake_sleep.wake_steps,
            'sleep_phases': self.learning_manager.wake_sleep.sleep_phases,
            'total_steps': self.learning_manager.wake_sleep.total_steps,

            # 5. Hebbian optimizer state (BCM thresholds tracked per-column)
            'hebbian_lr': self.learning_manager.hebbian_optimizer.lr,

            # 6. Config (for architecture recreation)
            'config': {
                field.name: getattr(self.cfg, field.name)
                for field in self.cfg.__dataclass_fields__.values()
            },

            # 7. Version marker
            'v11_version': 1,
        }

        torch.save(state, path)

    def load_state(self, path: str, strict: bool = True):
        """
        Restore COMPLETE brain state from disk.

        Loads everything saved by save_state(), restoring the brain
        to the exact state it was in when saved.

        Args:
          path: Path to the saved state file.
          strict: If True, requires exact parameter name match.
                  If False, ignores missing/unexpected keys (useful for
                  loading partial states or after architecture changes).
        """
        state = torch.load(path, map_location='cpu', weights_only=False)

        # 1. Model parameters and buffers
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        if strict:
            self.load_state_dict(state['model_state_dict'])
        else:
            self.load_state_dict(state['model_state_dict'], strict=False)

        # 2. Hippocampal episodic memory
        if 'hippocampal_memories' in state:
            restored = []
            for mem in state['hippocampal_memories']:
                entry = {}
                for k, v in mem.items():
                    v = v.clone()
                    # Normalize to [1, T, D] -- old checkpoints stored [B, T, D]
                    if v.dim() >= 3 and v.shape[0] > 1:
                        v = v.mean(dim=0, keepdim=True)
                    entry[k] = v
                restored.append(entry)
            self.hippocampus._memory_buffer = restored
            self.hippocampus._surprise_buffer = list(state.get('hippocampal_surprises', []))

        # 3. Developmental program step
        if 'developmental_step' in state:
            self.learning_manager.development._current_step = state['developmental_step']

        # 4. Wake-sleep scheduler state
        if 'wake_steps' in state:
            self.learning_manager.wake_sleep.wake_steps = state['wake_steps']
        if 'sleep_phases' in state:
            self.learning_manager.wake_sleep.sleep_phases = state['sleep_phases']
        if 'total_steps' in state:
            self.learning_manager.wake_sleep.total_steps = state['total_steps']

        # 5. Hebbian optimizer LR
        if 'hebbian_lr' in state:
            self.learning_manager.hebbian_optimizer.lr = state['hebbian_lr']

    #

    # Self-Repair: Weight Rollback + Adaptive Sleep
    #


    @torch.no_grad()
    def checkpoint_best(self, loss: float):
        """
        Save current state as best if loss improved.
        Called periodically during training. If loss diverges > 2x best,
        triggers rollback.

        Also monitors weight norms across all columns for explosion detection.
        Graduated response:
          - Mild (norm > 50): emergency downscaling for that column
          - Severe (norm > 200 or loss > 2x best for 100+ steps): full rollback

        Brain analogy: the brain's error-monitoring system (ACC + DA)
        detects when something has gone wrong and reverts to last-known-good
        strategy (habit system in basal ganglia takes over from PFC).
        """
        self._steps_since_best += 1

        #        Weight norm explosion detection (graduated response)
        # Check every column's total parameter norm
        max_col_norm = 0.0
        for i, col in enumerate(self.columns):
            col_norm = sum(p.data.norm().item() ** 2 for p in col.parameters()) ** 0.5
            max_col_norm = max(max_col_norm, col_norm)
            # Mild explosion: emergency downscaling for this column
            if col_norm > 50.0:
                scale = 50.0 / col_norm
                for p in col.parameters():
                    p.data.mul_(scale)

        # Severe explosion: regardless of loss, trigger rollback
        if max_col_norm > 200.0 and self._best_state is not None:
            return self._trigger_rollback(loss)

        if loss < self._best_loss:
            self._best_loss = loss
            self._best_state = {k: v.cpu().clone() for k, v in self.state_dict().items()}
            self._steps_since_best = 0
            return {'checkpointed': True, 'best_loss': loss}

        # Divergence detection: loss > 2x best AND sustained (100+ steps since best)
        if loss > self._best_loss * 2.0 and self._steps_since_best > 100:
            return self._trigger_rollback(loss)

        return {'checkpointed': False}

    @torch.no_grad()
    def _trigger_rollback(self, current_loss: float) -> dict:
        """
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        Emergency rollback to best-known state.
        Also increases sleep frequency (adaptive sleep) to consolidate.
        """
        if self._best_state is None:
            return {'rollback': False, 'reason': 'no_best_state'}

        self._rollback_count += 1

        # Restore best weights
        self.load_state_dict(self._best_state)
        self._steps_since_best = 0

        # Adaptive sleep: after rollback, increase sleep frequency
        # (brain under stress needs more consolidation)
        old_interval = self.learning_manager.wake_sleep.cfg.sleep_interval
        new_interval = max(10, old_interval // 2)
        self.learning_manager.wake_sleep.cfg.sleep_interval = new_interval

        return {
            'rollback': True,
            'rollback_count': self._rollback_count,
            'from_loss': current_loss,
            'to_loss': self._best_loss,
            'sleep_interval': f'{old_interval} → {new_interval}',
        }

    #

    # Utilities
    #


    def get_optimizer_groups(self, lr: Optional[float] = None) -> List[dict]:
        """
        Get parameter groups for optimizer (if using backprop mode).

        Groups:
          - base: all regular parameters (1x LR)
          - gates: gate_raw parameters (20x LR, fast learning)
          - sigma2: precision parameters (1x LR, careful)
        """
        if lr is None:
            lr = self.cfg.base_lr

        base_params = []
        gate_params = []
        sigma2_params = []

        for name, param in self.named_parameters():
            if not param.requires_grad:
                continue
            if 'gate_raw' in name:
                gate_params.append(param)
            elif 'sigma2' in name:
                sigma2_params.append(param)
            else:
                base_params.append(param)

        return [
            {'params': base_params, 'lr': lr, 'weight_decay': self.cfg.weight_decay},
            {'params': gate_params, 'lr': lr * self.cfg.gate_lr_mult, 'weight_decay': 0.0},
            {'params': sigma2_params, 'lr': lr, 'weight_decay': 0.0},
        ]


#

# Model Factory
#


def create_v11_model(cfg: Optional[V11Config] = None, **kwargs) -> CorticalColumnNetwork:
    """
    Create a V11 Cortical Column Network.

    Usage:
      model = create_v11_model()  # Default config
      model = create_v11_model(V11Config(d_model=1024, n_columns=16))  # Custom
      model = create_v11_model(d_model=256, n_columns=6)  # kwargs override
    """
    if cfg is None:
        cfg = V11Config(**kwargs)
    else:
```

Listing 12: Full Model — CorticalColumnNetwork (v11_model.py, 1,025 lines)

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

14 Autonomous Operation

Source: `nct_v2/v11_autonomous.py` **Lines:** 1,830 **Classes:** 11

14.1 Neuroscience Foundation

The brain has no "training mode" vs "inference mode." Every perception updates synapses. Every sleep consolidates memory. Every moment of boredom triggers curiosity. This module closes the autonomy loop, making NCT a living, self-improving system with eight autonomous subsystems.

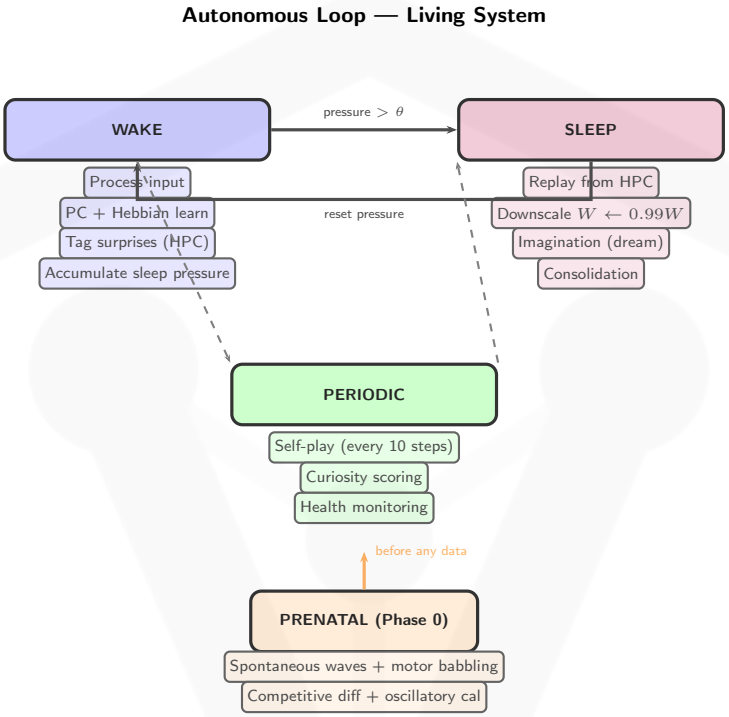


Figure 65: The autonomous operation cycle. Wake processes input and learns. Sleep replays, downscales, dreams, and consolidates. Periodic tasks (self-play, curiosity) run during wake. Prenatal self-organization bootstraps the network before any external data.

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

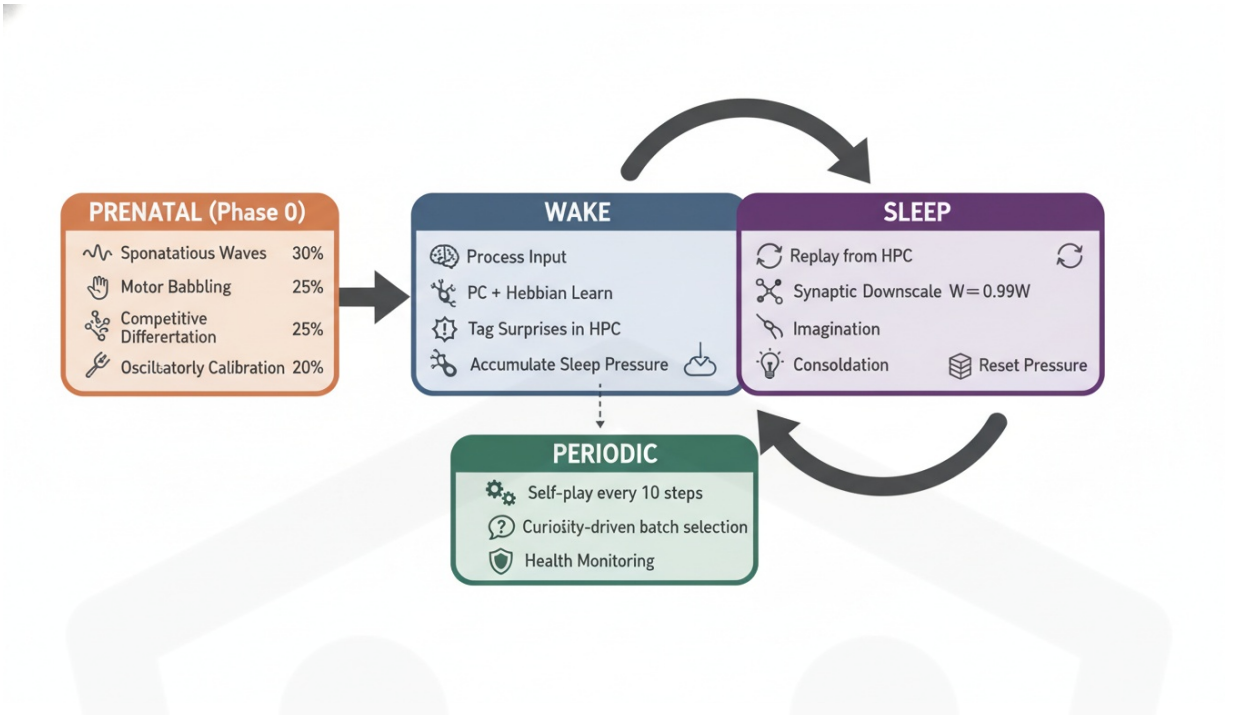


Figure 66: (Illustrated) The autonomous operation cycle. Wake processes input and learns. Sleep replays, downscales, dreams, and consolidates. Periodic tasks (self-play, curiosity) run during wake. Prenatal self-organizati...

14.2 Always-On Learning (`AlwaysOnLearning`)

After every forward pass — whether training, inference, or generation — local learning updates run. The effective learning rate is dynamically modulated:

$$\eta_{\text{eff}} = \eta_{\text{base}} \cdot \text{ACh} \cdot (1 + \text{NE}) \cdot \text{surprise_scale} \cdot (1 - \text{sleep_pressure}) \tag{79}$$

High acetylcholine (attending) → more learning. High norepinephrine (aroused) → more learning. High surprise → more learning. High sleep pressure (tired) → less wake learning.

14.3 Imagination Engine (`ImaginationEngine`)

Theorem 14.1 (Learning from Imagination)**.** *The brain learns not just from reality but from imagination. The engine:*

1. *Retrieves a hippocampal memory as seed*
2. *Auto-regressively generates a continuation (up to 16 tokens)*
3. *Evaluates the imagined sequence (computes prediction error)*
4. *If PE is high (novel imagination), tags for additional replay*

This enables "dreaming" — learning from imagined scenarios, not just replayed experiences. The distinction from standard replay: imagination generates novel *sequences not seen in training data.*

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

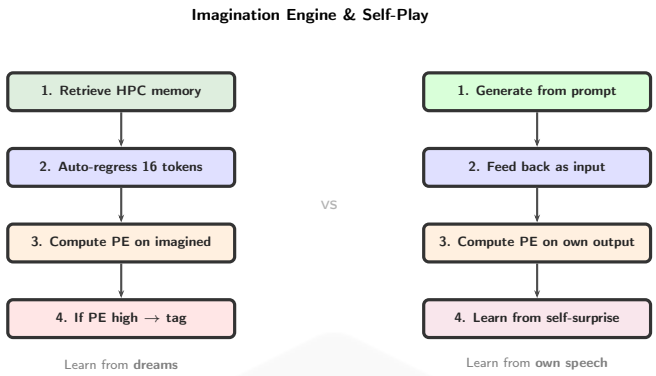


Figure 67: Imagination engine (left): retrieve hippocampal memory, generate novel continuation, evaluate prediction error, tag high-PE imaginations for replay. Self-play (right): generate text, feed it back as input, evaluate own prediction error, learn from self-surprise. Both enable learning without external data.

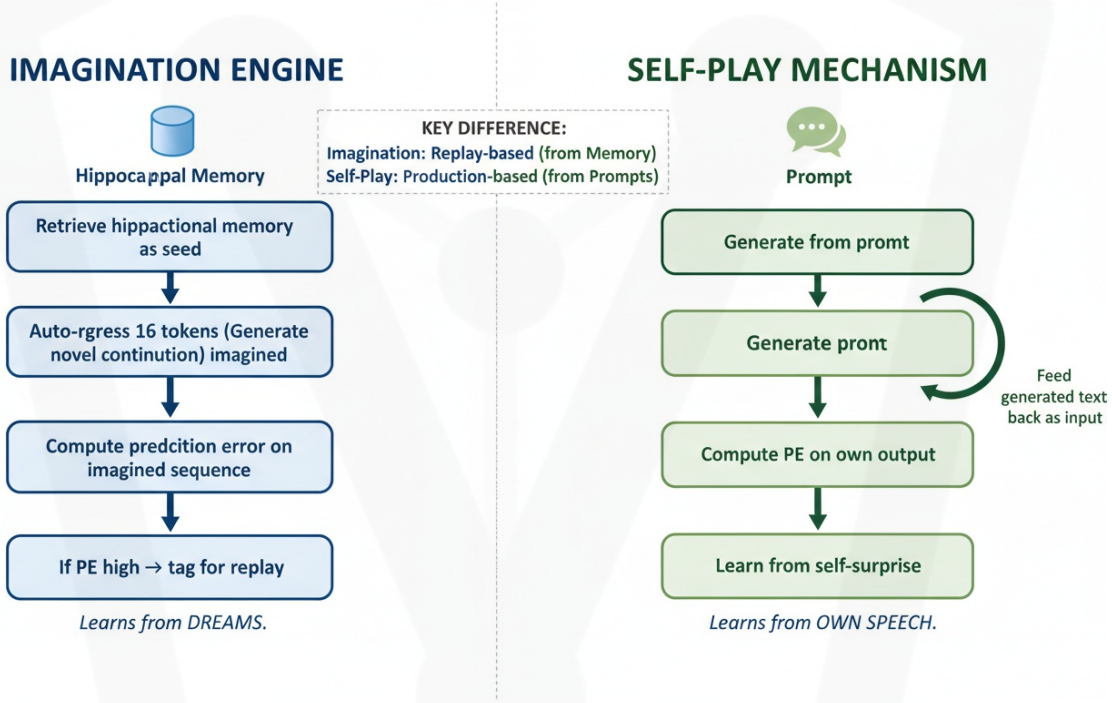


Figure 68: (Illustrated) Imagination engine (left): retrieve hippocampal memory, generate novel continuation, evaluate prediction error, tag high-PE imaginations for replay. Self-play (right): generate text, feed it back a...

14.4 Curiosity Drive (`CuriosityDrive`)

Theorem 14.2 (Expected Free Energy Minimization (Friston, 2010)). *Inputs that would maximally reduce prediction uncertainty are prioritized:*

$$curiosity(\mathbf{x}) = \mathbb{E}[information_gain(\mathbf{x})] = H[\sigma^2] - \mathbb{E}[H[\sigma^2|\mathbf{x}]] \tag{80}$$

In practice, the curiosity score is computed as the variance of σ^2 across columns: inputs where columns disagree *on their certainty are scored highest. The system can also generate "curious prompts" by sampling from high-uncertainty vocabulary items.*

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

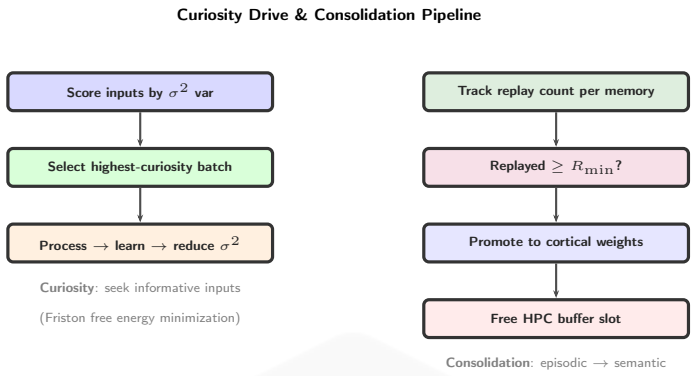


Figure 69: Curiosity drive (left): score inputs by σ^2 variance across columns, select the most informative batch, learn from it. Consolidation pipeline (right): track how many times each HPC memory has been replayed, promote frequently-replayed memories to cortical weights, free buffer space.

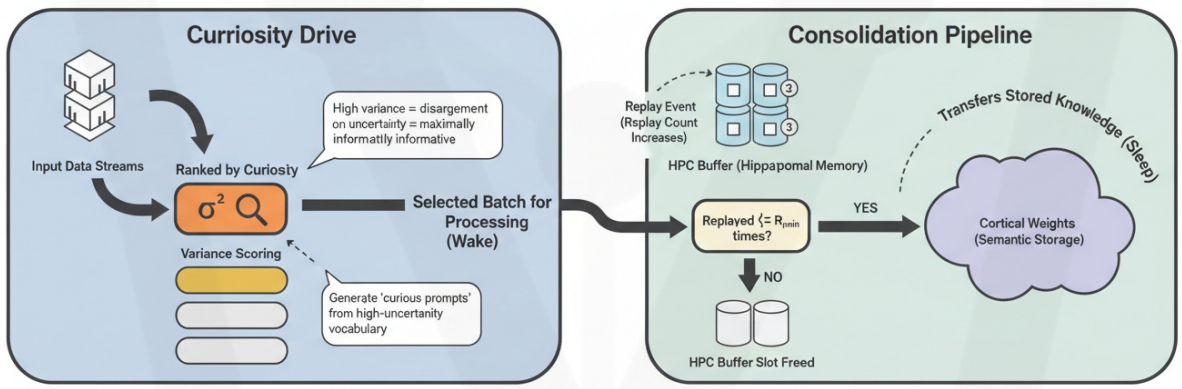


Figure 70: (Illustrated) Curiosity drive (left): score inputs by σ^2 variance across columns, select the most informative batch, learn from it. Consolidation pipeline (right): track how many times each HPC memory ha...

14.5 Self-Play (`SelfPlay`)

The brain processes its own speech (auditory cortex hears you speak), creating a generate → perceive → evaluate → learn feedback loop. NCT implements this as self-play:

$$\text{Generate}(\mathbf{x}_{\text{prompt}}) \to \mathbf{x}_{\text{gen}} \to \text{Forward}(\mathbf{x}_{\text{gen}}) \to \boldsymbol{\epsilon}_{\text{self}} \to \text{Learn}(\boldsymbol{\epsilon}_{\text{self}}) \tag{81}$$

Self-play loss measures how surprised the model is by its own output. High self-play loss = the model surprised itself = learning opportunity. Self-play activates every 10 steps after a 200-step warm-up. If last 3 self-play losses all exceed 15.0 (still producing gibberish), interval increases 5×.

Disease mapping: inner speech deficits (schizophrenia) → inability to monitor own speech → thought insertion. Stuttering → failed forward model of own speech production.

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

14.6 Continual Learning Guard (`ContinualLearningGuard`)

EWC-like (Kirkpatrick et al., 2017) synaptic importance estimation:

$$\mathcal{L}_{\text{EWC}} = \sum_i \frac{\lambda}{2} F_i (\theta_i - \theta_i^*)^2 \tag{82}$$

where F_i is the Fisher information (importance) of parameter i and θ_i^* is the optimal value from previous tasks. Parameters that were important for old tasks resist modification. Currently disabled on 6GB VRAM due to memory constraints (Fisher estimation requires storing per-parameter importance scores).

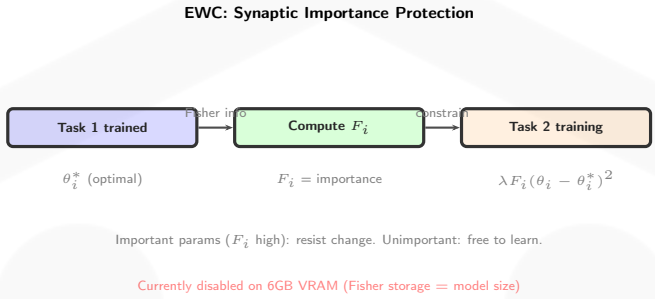


Figure 71: EWC synaptic importance: Fisher information identifies which parameters were important for old tasks. During new learning, important parameters are penalized for changing, preventing catastrophic forgetting.

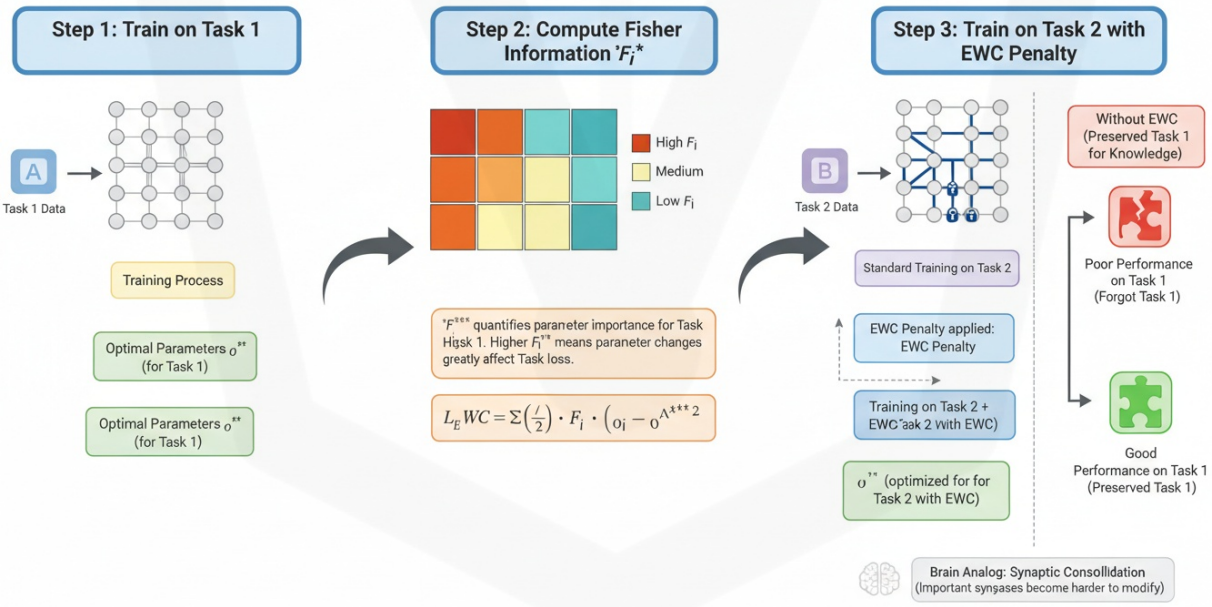


Figure 72: (Illustrated) EWC synaptic importance: Fisher information identifies which parameters were important for old tasks. During new learning, important parameters are penalized for changing, preventing catastrophic f...

14.7 Consolidation Manager (`ConsolidationManager`)

The complementary learning systems theory (McClelland et al., 1995) posits that the brain uses two systems with different learning rates: the hippocampus learns quickly (one-shot episodic encoding) while

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

the neocortex learns slowly (gradual statistical extraction). During sleep, the hippocampus "teaches" the neocortex by replaying stored episodes, gradually transferring knowledge from fast episodic storage to slow semantic storage.

The `ConsolidationManager` implements this transfer pipeline. It tracks how many times each hippocampal memory has been replayed during sleep phases. When a memory has been replayed at least $R_{\min}$ times (indicating it is a recurring, important pattern rather than a one-time anomaly), it is promoted: the cortical column weights are updated to represent this pattern natively (without hippocampal support), and the memory is removed from the HPC buffer to free capacity for new episodes. This produces the biological phenomenon of **systems consolidation**: recently learned facts depend on the hippocampus, but well-consolidated knowledge becomes hippocampus-independent.

14.8 Autonomous Loop (`AutonomousLoop`)

The main orchestrator managing the complete autonomous cycle without human intervention. On each step:

Wake: Process input → always-on learning → tag surprises → accumulate sleep pressure.

Periodic: Self-play (every 10 steps), curiosity-driven batch selection, health monitoring.

Sleep (when pressure exceeds threshold): Replay → downscale → imagination → consolidation → reset pressure.

Evaluate (periodic): Run validation, check for divergence, update self-assessment metrics.

The loop can run indefinitely in continuous mode — a never-ending self-improving cycle.

14.9 Prenatal Development (`PrenatalDevelopment`)

Theorem 14.3 (Prenatal Self-Organization)**.** *Before any external data, the network bootstraps through spontaneous activity, mirroring prenatal brain development:*

1. Spontaneous Waves (30%): *Momentum random walk in embedding space (Meister et al., 1991). Velocity* $\mathbf{v} \leftarrow 0.9\mathbf{v} + 0.1\mathcal{N}(0, 0.05)$*. Teaches L2/3 about local temporal structure: "nearby things are related."*

2. Motor Babbling (25%): *Auto-regressive self-generation (Doya & Sejnowski, 1998). The model generates from its own predictions (capped at 16 tokens), creating a production-perception loop. Calibrates the generate → predict cycle.*

3. Competitive Differentiation (25%): *Structured patterns (repeated motifs, gradients, alternating sequences) force columns to specialize through competitive Hebbian learning. Each pattern type drives different statistical selectivity.*

4. Oscillatory Calibration (20%): *Multi-frequency sinusoidal input (Khazipov & Luhmann, 2006). Calibrates oscillatory periods* (θ, α, δ) *and coupling strengths* $(\kappa_1, \kappa_2, \kappa_3)$*.*

After prenatal development, the network is not a blank slate — it has calibrated rhythms, differentiated columns, and functional prediction circuitry.

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

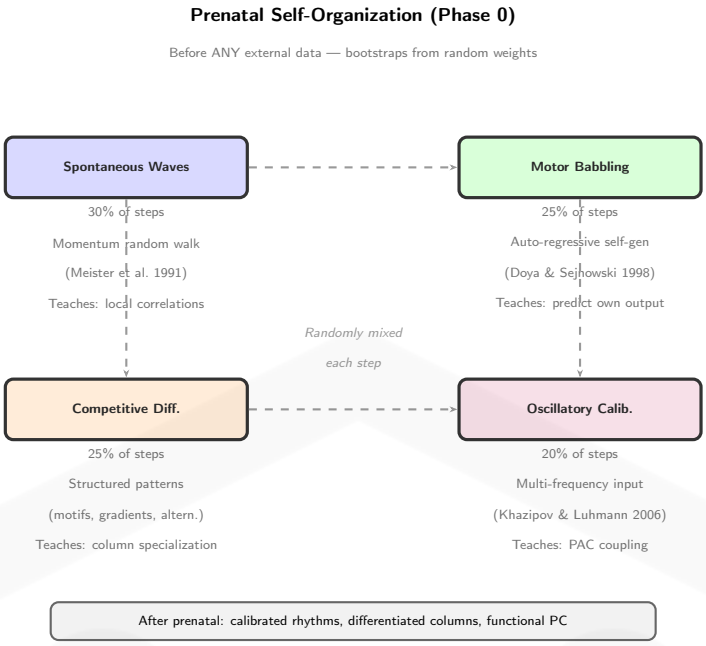


Figure 73: Prenatal self-organization: four types of spontaneous activity run before any external data. Each step randomly selects a phase type. Spontaneous waves teach local correlations, motor babbling calibrates the production-perception loop, competitive differentiation forces column specialization, and oscillatory calibration tunes the PAC coupling strengths.

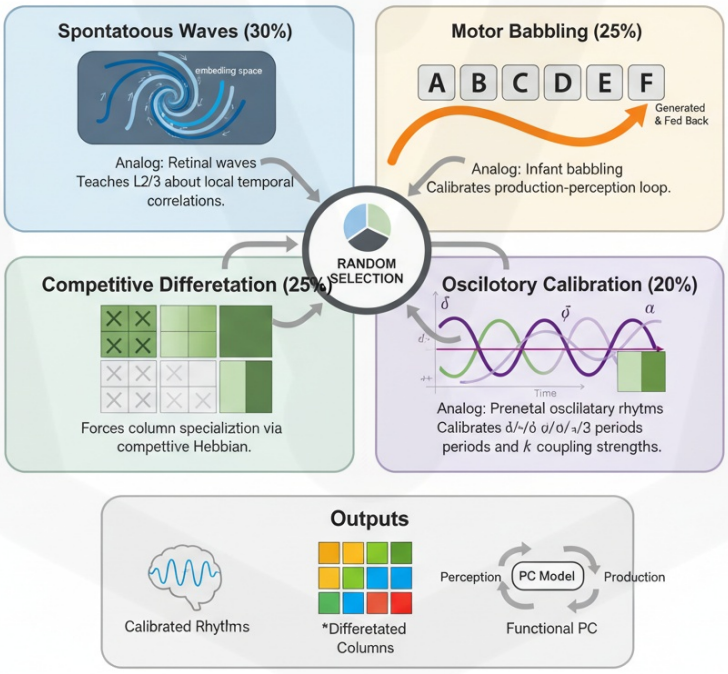


Figure 74: (Illustrated) Prenatal self-organization: four types of spontaneous activity run before any external data. Each step randomly selects a phase type. Spontaneous waves teach local correlations, motor babbling cali...

14.10 Source Code

```
        # Apply any kwargs overrides
        for k, v in kwargs.items():
            if hasattr(cfg, k):
                setattr(cfg, k, v)

```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
    model = CorticalColumnNetwork(cfg)
    return model



# ============================================================================
# Autonomous Operation (Wake-Sleep, Prenatal, Imagination)
# Source: nct_v2/v11_autonomous.py
# ============================================================================

"""
NCT V11: Autonomous Operation -- The Life of a Brain
=====================================================

The brain has no "training mode" vs "inference mode." It's ALWAYS learning.
Every perception updates synapses. Every sleep consolidates memory.
Every moment of boredom triggers curiosity.

This module closes the autonomy loop, making V11 a living, self-improving system:

  1. AlwaysOnLearning: Learn from every token processed (no train/eval distinction)
  2. ImaginationEngine: Generate synthetic experiences from memory, learn from dreams
  3. CuriosityDrive: Expected free energy → seek maximally informative inputs (Friston 2010)
  4. ConsolidationManager: Episodic → semantic transfer (McClelland et al. 1995)
  5. ContinualLearningGuard: Protect old knowledge while learning new (synaptic importance)
  6. AutonomousLoop: Orchestrates wake/sleep/dream/evaluate without human intervention

Disease mapping:
  - Alzheimer's → consolidation failure → forgetting
  - ADHD → curiosity dysregulation → can't focus
  - Insomnia → no sleep consolidation → degraded learning
  - Anosognosia → no self-evaluation → unaware of deficits
  - Depression → no curiosity drive → learned helplessness
"""



#

# 1. Always-On Learning
#


class AlwaysOnLearning:
    """
    The brain learns from EVERY experience. There is no eval() mode in biology.

    After every forward pass -- whether training, inference, or generation --
    local learning updates run. PC adjusts predictions, Hebbian adjusts
    associations, neuromod gates everything.

    The learning rate is modulated by:
      - Acetylcholine (high attention → more learning)
      - Norepinephrine (high arousal → more learning)
      - Surprise (high PE → more learning)
      - Sleep pressure (close to sleep → less wake learning)

    This is the difference between a brain on a slab and a living brain.
    """

    def __init__(self, base_lr: float = 1e-4, sleep_pressure_decay: float = 0.999):
        self.base_lr = base_lr
        self.sleep_pressure = 0.0  # 0 = just woke up, 1 = need sleep
        self.sleep_pressure_decay = sleep_pressure_decay
        self.total_experiences = 0
        self._ewc_guard = None  # Set by AutonomousLoop when EWC is initialized

    def compute_lr(self, neuromod_state) -> float:
        """Compute effective learning rate from neuromodulatory state."""
        # ACh gates plasticity (attention → learning)
        ach = neuromod_state.acetylcholine.mean().item() if hasattr(neuromod_state, 'acetylcholine') else 0.5
        # NE gates arousal (surprise → learning)
        ne = neuromod_state.norepinephrine.mean().item() if hasattr(neuromod_state, 'norepinephrine') else 0.5
        # Sleep pressure reduces learning efficiency
        wake_efficiency = max(0.1, 1.0 - self.sleep_pressure)

        return self.base_lr * (0.5 + ach) * (0.5 + ne) * wake_efficiency

    def on_experience(self, model, result: dict):
        """
        Called after EVERY forward pass. Runs local learning.

        Sleep pressure accumulates proportional to prediction error --
        high surprise experiences are more "tiring" (adenosine analogy).
        """
        self.total_experiences += 1
        # Base pressure accumulation (1 - decay ~ 0.001 per step)
        base_increment = 1.0 - self.sleep_pressure_decay
        # Surprise-scaled: high PE → faster pressure buildup
        pe = result.get('metrics', {}).get('mean_pred_error', 0.5)
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```python
        surprise_mult = 1.0 + min(pe, 5.0)  # 1x-6x based on surprise
        self.sleep_pressure = min(1.0, self.sleep_pressure + base_increment * surprise_mult)

        # Run local learning (PC + Hebbian + neuromod + EWC elastic pull)
        # Sleep pressure modulates learning efficiency: tired brain learns worse
        # (adenosine reduces LTP -- Basheer et al. 2004)
        if model._last_column_outputs is not None:
            wake_efficiency = max(0.1, 1.0 - self.sleep_pressure)
            orig_lr = model.learning_manager.hebbian_optimizer.lr
            model.learning_manager.hebbian_optimizer.lr = orig_lr * wake_efficiency
            metrics = model.local_learning_step(ewc_guard=self._ewc_guard)
            model.learning_manager.hebbian_optimizer.lr = orig_lr  # Restore
            return metrics
        return {}

    def on_sleep(self):
        """Reset sleep pressure after sleep phase."""
        self.sleep_pressure = 0.0


#

# 2. Imagination Engine (Dream-Based Learning)
#


class ImaginationEngine:
    """
    The brain learns not just from reality but from imagination.

    During sleep and idle moments, the hippocampus provides memory cues.
    The cortex generates predictions from these cues (dreaming).
    Prediction errors during dreams update cortical weights.

    This is why sleep deprivation impairs learning -- you lose dream-time.

    Mechanism:
      1. Sample hippocampal memory cue (partial pattern)
      2. Feed through cortical columns (like a prompt)
      3. Generate continuation (dream content)
      4. Compute PC loss on dreamed sequence
      5. Run local learning on dream errors

    Neuroscience: Replay + generative dreaming (Genzel et al. 2014, Lewis & Durrant 2011).
    """

    def __init__(self, dream_steps: int = 10, dream_noise: float = 0.05):
        self.dream_steps = dream_steps
        self.dream_noise = dream_noise
        self.total_dreams = 0

    def dream(self, model, hippocampus, device: torch.device) -> Dict[str, float]:
        """
        Generate and learn from imagined experiences.

        Dreams are NOT just monitoring -- they're genuine learning. The brain
        learns during sleep by generating predictions from memory cues and
        updating weights based on prediction errors (Genzel et al. 2014).

        Two learning mechanisms during dreams:
          1. PC backward: prediction errors update L5 prediction weights
          2. Hebbian: dream activations strengthen cortical associations

        Returns: metrics about the dream phase.
        """
        memories = hippocampus.replay(n_samples=min(4, hippocampus.memory_size), device=device)
        if not memories:
            return {'dreams': 0}

        metrics = {'dreams': 0, 'dream_pc_loss': 0.0}
        dream_lr = 1e-4  # Dream learning rate -- consolidation (offline replay potentiates)

        for mem in memories:
            x = mem['x']  # [1, T, D] -- batch-pooled at storage
            if x.dim() < 3:
                continue

            # Add noise (dreams are noisy reconstructions)
            dream_input = x + torch.randn_like(x) * self.dream_noise

            # Run through columns WITH gradients (dreams teach)
            h = dream_input
            dream_outputs = []
            for col in model.columns:
                out = col(ff_input=h, td_context=None)
                dream_outputs.append(out)
                h = out.output

            # Compute prediction errors and LEARN from dreams
            if len(dream_outputs) >= 2:
                dream_pc_loss = 0.0
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
                pc_losses = []
                for l in range(len(dream_outputs)):
                    if l == 0:
                        target = dream_input.detach()
                    else:
                        target = dream_outputs[l - 1].output.detach()
                    pred = dream_outputs[l].prediction
                    sigma2 = dream_outputs[l].sigma2
                    error = target - pred
                    pc_loss_l = (error.pow(2) / (2 * sigma2 + 1e-6) + 0.5 * sigma2.log()).mean()
                    pc_losses.append(pc_loss_l)
                    dream_pc_loss += pc_loss_l.item()

                # Analytical PC update on dream predictions (no autograd)
                dream_predictions = [out.prediction for out in dream_outputs]
                dream_targets = [dream_input.detach()] + [
                    dream_outputs[l].output.detach()
                    for l in range(len(dream_outputs) - 1)
                ]
                dream_precisions = [out.sigma2 for out in dream_outputs]
                # [bundled] from .v11_learning import PredictiveCodingOptimizer
                # Use a lightweight local update -- same analytical rule
                with torch.no_grad():
                    for l, col in enumerate(model.columns):
                        if l >= len(dream_predictions):
                            break
                        h_norm = col._last_l5_h_norm if hasattr(col, '_last_l5_h_norm') else None
                        if h_norm is None:
                            continue
                        error = dream_targets[l].detach() - dream_predictions[l].detach()
                        sigma2 = dream_precisions[l].clamp(min=1e-6)
                        B, T, D_model = error.shape
                        D_col = h_norm.shape[-1]
                        N = B * T
                        we = (error.float() / sigma2.float()).reshape(N, D_model)
                        hf = h_norm.float().reshape(N, D_col)
                        dW = -(we.T @ hf) / N
                        col.L5.W_predict.weight.data -= dream_lr * dW.to(col.L5.W_predict.weight.dtype)

                metrics['dream_pc_loss'] += dream_pc_loss / len(dream_outputs)
                metrics['dreams'] += 1

                # Hebbian learning from dream activations
                with torch.no_grad():
                    for i, col in enumerate(model.columns):
                        if hasattr(col, 'bcm') and col.bcm.enabled:
                            col.bcm.update_threshold(dream_outputs[i].l23_output)
                            col.L23.hebbian_update_bcm(
                                pre_input=dream_outputs[i].l23_input,
                                post_output=dream_outputs[i].l23_output,
                                bcm_theta=col.bcm.theta_bcm,
                                lr=dream_lr,
                            )

            self.total_dreams += 1

        if metrics['dreams'] > 0:
            metrics['dream_pc_loss'] /= metrics['dreams']

        return metrics


#

# 3. Curiosity Drive (Active Inference / Expected Free Energy)
#


class CuriosityDrive:
    """
    The brain doesn't passively process input -- it SEEKS informative stimuli.

    Friston's Active Inference (2010): agents minimize expected free energy,
    which decomposes into:
      EFE = E[prediction_error] - E[information_gain]
    The information gain term IS curiosity. High expected  ² reduction = curious.

    Implementation:
      For each candidate input, estimate how much  ² would decrease.
      Select the candidate that maximizes expected information gain.
      This makes the model actively seek what it doesn't know.

    Disease mapping:
      ADHD → curiosity too high, unfocused → can't sustain attention on one thing
      Depression → curiosity too low → no exploration, learned helplessness
      Mania → curiosity dysregulated → everything seems novel

    Zero learnable params -- purely algorithmic.
    """

    def __init__(self, curiosity_weight: float = 1.0, exploitation_weight: float = 0.5):
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        self.curiosity_weight = curiosity_weight
        self.exploitation_weight = exploitation_weight
        self.curiosity_history: List[float] = []

    @torch.no_grad()
    def score_input(self, model, input_ids: torch.Tensor) -> float:
        """
        Score an input by expected information gain.

        EIG ~ current_ ² - expected_posterior_ ²
        Approximation: inputs that produce high  ² are informative
        (the model is uncertain, so processing them teaches the most).

        Returns: curiosity score (higher = more informative)
        """
        result = model.forward(input_ids)
        sigma2_mean = result['metrics'].get('sigma2_mean', 1.0)
        pred_error = result['metrics'].get('mean_pred_error', 0.0)

        # High  ² = uncertain = informative to learn from
        # High prediction error = surprising = informative
        curiosity = self.curiosity_weight * sigma2_mean + self.exploitation_weight * pred_error
        self.curiosity_history.append(curiosity)
        return curiosity

    def select_batch(self, model, candidates: List[torch.Tensor], top_k: int = 1) -> List[int]:
        """
        Select the most informative inputs from candidates.

        Returns: indices of selected candidates (most curious first).
        """
        scores = []
        for i, cand in enumerate(candidates):
            score = self.score_input(model, cand)
            scores.append((score, i))

        scores.sort(reverse=True)  # Highest curiosity first
        return [idx for _, idx in scores[:top_k]]

    @property
    def mean_curiosity(self) -> float:
        if not self.curiosity_history:
            return 0.0
        recent = self.curiosity_history[-100:]
        return sum(recent) / len(recent)

    @torch.no_grad()
    def generate_curious_prompt(self, model, hippocampus, device: torch.device,
                                prompt_len: int = 16) -> Optional[torch.Tensor]:
        """
        Active curiosity: generate a training prompt the model wants to learn from.

        Strategy: recombine hippocampal memory fragments, prioritizing high-surprise
        memories (the ones the model struggled with). This targets the model's weakest
        points -- like a student reviewing the hardest problems.

        Three steps:
          1. Retrieve high-surprise memories from hippocampus
          2. Recombine fragments from multiple memories (cross-episode binding)
          3. Forward to get  ² → select positions where uncertainty is highest →
             use those token IDs as the prompt

        Returns: [1, prompt_len] token IDs, or None if no memories.
        """
        n_memories = min(3, hippocampus.memory_size)
        if n_memories == 0:
            return None

        # Retrieve memories prioritized by surprise (hardest experiences first)
        memories = hippocampus.replay(n_samples=n_memories, device=device)
        if not memories:
            return None

        # Collect all token-level representations from memories
        all_h = []
        for mem in memories:
            x = mem['x']  # [1, T, D] -- batch-pooled at storage
            if x.dim() >= 3 and x.shape[1] >= 2:
                all_h.append(x)
        if not all_h:
            return None

        # Recombine: interleave fragments from different memories
        # This creates novel combinations -- like dreaming across episodes
        combined = torch.cat(all_h, dim=1)  # [1, T_total, D]
        T_total = combined.shape[1]

        # Shuffle positions to create novel sequences (cross-episode recombination)
        perm = torch.randperm(T_total, device=device)
        combined = combined[:, perm, :]

```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        # Take first prompt_len positions
        h = combined[:, :min(prompt_len, T_total), :]

        # Forward through columns to get logits
        for col in model.columns:
            out = col(ff_input=h, td_context=None)
            h = out.output

        # Project to logits -- use greedy argmax (no noise garbage)
        logits = model.lm_head(model.lm_head_norm(h))  # [1, L, V]
        prompt_tokens = logits.argmax(dim=-1)  # [1, L] -- model's best guess

        # Pad if needed
        L = prompt_tokens.shape[1]
        if L < prompt_len:
            pad = torch.zeros(1, prompt_len - L, dtype=torch.long, device=device)
            prompt_tokens = torch.cat([prompt_tokens, pad], dim=1)
        else:
            prompt_tokens = prompt_tokens[:, :prompt_len]

        return prompt_tokens


#

# 4. Continual Learning Guard (Synaptic Importance)
#


#

# Self-Play: Learn from Own Output
#


class SelfPlay:
    """
    The brain learns not just from external input but from its OWN output.

    When you speak, your auditory cortex processes your own speech.
    When you write, your visual cortex reads what you wrote.
    This creates a feedback loop: generate → perceive → evaluate → learn.

    For language models, self-play means:
      1. Generate a continuation from a prompt
      2. Feed the generation BACK through the model as input
      3. Compute prediction error on the self-generated text
      4. If error is high → the model surprised itself → learn from it
      5. If error is low → the generation is consistent → reinforce patterns

    This is fundamentally different from standard training:
      - Standard: learn from human-written text (external)
      - Self-play: learn from own-generated text (internal)
      - Imagination: learn from noisy hippocampal replays (memory-based)

    Self-play enables GENUINE self-improvement: the model gets better by
    practicing, not just by consuming data.

    Disease mapping:
      Inner speech deficits (schizophrenia) → inability to monitor own speech
        → thought insertion / alien thoughts
      Stuttering → failed forward model of own speech production

    Zero learnable params -- purely algorithmic.
    """

    def __init__(self, gen_length: int = 32, self_play_interval: int = 10):
        self.gen_length = gen_length
        self.self_play_interval = self_play_interval
        self.total_self_plays = 0
        self.self_play_losses: List[float] = []

    @torch.no_grad()
    def should_play(self, step: int) -> bool:
        # Don't self-play until minimum competence (warm-up prevents garbage learning)
        if step < 200:
            return False
        # After warm-up, also check that previous self-play improved
        # (if we have history and last 3 were all high-loss, slow down)
        if len(self.self_play_losses) >= 3:
            recent = self.self_play_losses[-3:]
            if all(l > 15.0 for l in recent):  # Still producing gibberish
                return step % (self.self_play_interval * 5) == 0  # 5x slower
        return step > 0 and step % self.self_play_interval == 0

    def play(self, model, prompt_ids: torch.Tensor, device: torch.device) -> Dict[str, float]:
        """
        Self-play: generate → re-process → learn from own output.

        Args:
            model: V11 model
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
            prompt_ids: [B, T_prompt] -- seed prompt
            device: torch device

        Returns: metrics about self-play session.
        """
        metrics = {}

        # Phase 1: Generate text
        model.eval()
        with torch.no_grad():
            generated = model.generate(
                prompt_ids, max_new_tokens=self.gen_length, use_brain_decoding=True
            )
        model.train()

        if generated.shape[1] <= prompt_ids.shape[1]:
            return {'self_play': False}

        # Phase 2: Re-process the generated text as input
        # The model reads its own output -- like hearing yourself speak
        gen_input = generated[:, :-1]
        gen_targets = generated[:, 1:]

        result = model.forward(gen_input, gen_targets)

        # Phase 3: Evaluate -- how surprised is the model by its own output?
        if 'loss' in result:
            self_play_loss = result['loss'].item()
            metrics['self_play_loss'] = self_play_loss
            self.self_play_losses.append(self_play_loss)

            # Phase 4: Learn from own output via local learning
            # High loss = model surprised by own generation = opportunity to learn
            learn_metrics = model.local_learning_step()
            metrics.update({f'sp_{k}': v for k, v in learn_metrics.items()})

        self.total_self_plays += 1
        metrics['self_play'] = True
        metrics['total_self_plays'] = self.total_self_plays

        return metrics

    @property
    def mean_self_play_loss(self) -> float:
        if not self.self_play_losses:
            return float('nan')
        recent = self.self_play_losses[-50:]
        return sum(recent) / len(recent)


class ContinualLearningGuard:
    """
    Prevent catastrophic forgetting during continual learning.

    The brain doesn't forget everything old when learning something new.
    Key mechanisms:
      1. Synaptic consolidation: important synapses become harder to change
         (Clopath et al. 2008, EWC-like: Kirkpatrick et al. 2017)
      2. Complementary learning systems: hippocampus learns fast, cortex slow
         (McClelland et al. 1995 -- already in V11 via hippocampus)
      3. Sleep replay: rehearse old memories to prevent forgetting
         (already in V11 via wake-sleep)

    This module implements mechanism 1: synaptic importance estimation.
    Parameters that were important for old tasks resist modification.

    Implementation:
      - After each "task" (or periodically), compute Fisher information
        approximation for each parameter (|grad|² over held-out data)
      - During learning, penalize changes to high-importance parameters
      - Importance decays slowly (old knowledge gradually becomes modifiable)

    Zero learnable params -- importance is computed, not learned.
    """

    def __init__(self, lambda_ewc: float = 100.0, importance_decay: float = 0.999):
        self.lambda_ewc = lambda_ewc
        self.importance_decay = importance_decay
        self.importance: Dict[str, torch.Tensor] = {}
        self.anchor_params: Dict[str, torch.Tensor] = {}
        self._initialized = False

    @torch.no_grad()
    def estimate_importance(self, model: nn.Module, data_loader, n_batches: int = 10):
        """
        Estimate parameter importance via Fisher Information approximation.

        F_i ˜ E[  L  /    _i)²] -- how much the loss changes when  _i changes.
        Parameters with high Fisher = important for current knowledge.
        """
        # Initialize or decay existing importance
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        for name, param in model.named_parameters():
            if not param.requires_grad:
                continue
            if name in self.importance:
                self.importance[name] *= self.importance_decay
            else:
                self.importance[name] = torch.zeros_like(param)

        # Accumulate gradient magnitudes
        model.eval()
        n_seen = 0
        for batch_idx, batch in enumerate(data_loader):
            if batch_idx >= n_batches:
                break

            if isinstance(batch, dict):
                input_ids = batch['input_ids']
                targets = batch.get('targets', input_ids[:, 1:])
            else:
                input_ids = batch
                targets = batch[:, 1:]

            model.zero_grad()
            result = model(input_ids[:, :-1], targets)
            if 'loss' in result:
                result['loss'].backward()

            for name, param in model.named_parameters():
                if param.grad is not None:
                    self.importance[name] += param.grad.pow(2)

            n_seen += 1

        # Normalize
        if n_seen > 0:
            for name in self.importance:
                self.importance[name] /= n_seen

        # Store current params as anchors
        for name, param in model.named_parameters():
            if param.requires_grad:
                self.anchor_params[name] = param.data.clone()

        self._initialized = True
        model.train()

    def compute_penalty(self, model: nn.Module) -> torch.Tensor:
        """
        Compute EWC penalty:  _i  F_i × ( _i -  *_i)²

        Returns: scalar penalty to add to loss.
        """
        if not self._initialized:
            return torch.tensor(0.0)

        penalty = torch.tensor(0.0, device=next(model.parameters()).device)
        for name, param in model.named_parameters():
            if name in self.importance and name in self.anchor_params:
                diff = param - self.anchor_params[name].to(param.device)
                penalty = penalty + (self.importance[name].to(param.device) * diff.pow(2)).sum()

        return 0.5 * self.lambda_ewc * penalty


#

# 5. Consolidation Manager
#


class ConsolidationManager:
    """
    Manages the episodic → semantic consolidation pipeline.

    McClelland et al. 1995: The brain has two learning systems:
      - Hippocampus: fast, episodic, specific (stores individual experiences)
      - Neocortex: slow, semantic, general (stores patterns across experiences)

    Consolidation transfers knowledge from hippocampus to cortex:
      1. Cluster similar hippocampal episodes
      2. Extract centroid pattern (the "gist")
      3. Train cortex on centroids slowly (interleaved with old knowledge)
      4. Delete consolidated episodes (they're now in cortex)

    This happens during deep sleep (SWS stages 3-4).
    """

    def __init__(self, consolidation_interval: int = 500, min_memories: int = 20):
        self.consolidation_interval = consolidation_interval
        self.min_memories = min_memories
        self.total_consolidations = 0
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        self.semantic_knowledge: List[Dict] = []

    def should_consolidate(self, step: int, hippocampus) -> bool:
        """Check if it's time for consolidation."""
        return (
            step > 0
            and step % self.consolidation_interval == 0
            and hippocampus.memory_size >= self.min_memories
        )

    @torch.no_grad()
    def consolidate(self, model, hippocampus, columns, device) -> Dict[str, float]:
        """
        Run full consolidation: cluster → extract → train → forget.

        Returns: consolidation metrics.
        """
        # Phase 1: Cluster and extract semantic memories
        semantic_mems = hippocampus.consolidate(n_clusters=8, min_cluster=3)
        if not semantic_mems:
            return {'consolidated': 0}

        # Phase 2: Train cortex on extracted patterns
        # Each centroid is a semantic memory -- run it through columns
        # and perform Hebbian learning (slow integration into cortical weights)
        n_consolidated = 0
        for sem_mem in semantic_mems:
            pattern = sem_mem['pattern'].to(device)  # [D]
            # Expand to [1, 1, D] for column processing
            pattern_input = pattern.unsqueeze(0).unsqueeze(0)

            # Slow cortical training: multiple passes over the pattern
            for _ in range(5):  # 5 passes per pattern (slow learning)
                h = pattern_input
                for col in columns:
                    out = col(ff_input=h, td_context=None)
                    # Hebbian update with very low LR (slow cortical integration)
                    col.L23.hebbian_update(out.l23_input, lr=1e-4)
                    h = out.output

            n_consolidated += 1

        self.total_consolidations += 1
        self.semantic_knowledge.extend(semantic_mems)

        return {
            'consolidated': n_consolidated,
            'episodes_removed': sum(m['n_episodes'] for m in semantic_mems),
            'memories_remaining': hippocampus.memory_size,
        }


#

# 6. Autonomous Loop -- The Orchestrator
#


@dataclass
class AutonomousConfig:
    """Configuration for autonomous operation."""
    # Always-on learning
    always_learn: bool = True
    base_lr: float = 1e-4

    # Sleep schedule
    sleep_interval: int = 100       # Wake steps between sleep phases
    dream_steps: int = 10           # Dreams per sleep phase

    # Curiosity
    curiosity_enabled: bool = True
    curiosity_weight: float = 1.0

    # Consolidation
    consolidation_interval: int = 500
    min_memories_for_consolidation: int = 20

    # Continual learning
    ewc_enabled: bool = False  # Disabled: OOMs on 6GB GPUs (needs .backward())
    ewc_lambda: float = 100.0
    ewc_estimation_interval: int = 500  # Estimate importance every N steps

    # Self-play
    self_play_interval: int = 20    # Steps between self-play sessions

    # Self-evaluation
    eval_interval: int = 200        # Steps between self-evaluations


class AutonomousLoop:
    """
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
    The main orchestrator that makes V11 a living, self-improving system.

    Lifecycle (runs forever):
      WAKE → process input → always-on learning → accumulate sleep pressure
      → when sleep_pressure > threshold:
        SLEEP → replay → dream → consolidate → downscale → reset pressure
      → back to WAKE

    Self-evaluation runs periodically during wake:
      - Test on held-out hippocampal memories
      - Compare current vs previous performance
      - Adjust learning strategy if degrading

    Curiosity modulates input selection:
      - Score candidate inputs by expected information gain
      - Prefer inputs the model is most uncertain about

    This is the closed loop that makes V11 "unstoppable" -- it continuously
    improves without human intervention, as long as data flows through it.
    """

    def __init__(self, model, config: Optional[AutonomousConfig] = None):
        if config is None:
            config = AutonomousConfig()
        self.config = config
        self.model = model

        # Components
        self.always_on = AlwaysOnLearning(
            base_lr=config.base_lr,
        )
        self.imagination = ImaginationEngine(dream_steps=config.dream_steps)
        self.curiosity = CuriosityDrive(curiosity_weight=config.curiosity_weight)
        self.consolidation = ConsolidationManager(
            consolidation_interval=config.consolidation_interval,
            min_memories=config.min_memories_for_consolidation,
        )
        self.continual_guard = ContinualLearningGuard(
            lambda_ewc=config.ewc_lambda,
        )
        # Wire EWC guard into always-on learning so elastic pull is applied during local learning
        self.always_on._ewc_guard = self.continual_guard
        self.self_play = SelfPlay(
            gen_length=32,
            self_play_interval=getattr(config, 'self_play_interval', 20),
        )

        # State
        self.global_step = 0
        self._last_input_ids: Optional[torch.Tensor] = None
        self.wake_steps_since_sleep = 0
        self.performance_history: List[float] = []
        self._state = 'wake'  # 'wake', 'sleep', 'dream', 'consolidate', 'evaluate'

    @property
    def state(self) -> str:
        return self._state

    def process(self, input_ids: torch.Tensor, targets: Optional[torch.Tensor] = None) -> Dict:
        """
        Process a single experience (the primary wake-phase operation).

        1. Forward pass
        2. Always-on learning (if enabled)
        3. Check if sleep is due
        4. Periodic self-evaluation

        Returns: forward result + autonomous metrics
        """
        self._state = 'wake'

        # Forward pass
        result = self.model.forward(input_ids, targets)

        # Always-on learning (the brain learns from every experience)
        auto_metrics = {}
        if self.config.always_learn:
            learn_metrics = self.always_on.on_experience(self.model, result)
            auto_metrics.update(learn_metrics)

        # EWC penalty (protect old knowledge)
        if self.config.ewc_enabled and self.continual_guard._initialized:
            ewc_penalty = self.continual_guard.compute_penalty(self.model)
            result['ewc_loss'] = ewc_penalty
            auto_metrics['ewc_penalty'] = ewc_penalty.item()

        # Track
        self.global_step += 1
        self.wake_steps_since_sleep += 1
        auto_metrics['sleep_pressure'] = self.always_on.sleep_pressure
        auto_metrics['global_step'] = self.global_step
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        auto_metrics['state'] = 'wake'

        result['autonomous_metrics'] = auto_metrics
        return result

    def should_sleep(self) -> bool:
        """Check if sleep phase is due.

        The brain sleeps when adenosine accumulation (sleep pressure) is high enough,
        NOT on a fixed schedule. Step count is a fallback lower bound.

        Two conditions (either triggers sleep):
          1. Sleep pressure > 0.7 (adenosine model -- primary)
          2. Wake steps >= sleep_interval (failsafe -- prevents insomnia)
        """
        pressure_threshold = 0.7
        return (self.always_on.sleep_pressure >= pressure_threshold or
                self.wake_steps_since_sleep >= self.config.sleep_interval)

    def sleep_cycle(self, device: torch.device) -> Dict[str, float]:
        """
        Full sleep cycle: replay → dream → consolidate → downscale.

        This is the brain's offline learning phase.
        All learning during sleep is analytical (direct .data manipulation) --
        no autograd needed. torch.no_grad() prevents OOM from activation tensors.
        """
        metrics = {'phase': 'sleep'}

        with torch.no_grad():
            # Phase 1: Memory replay (from V11 learning manager)
            self._state = 'sleep'
            sleep_metrics = self.model.sleep_phase()
            metrics.update({f'sleep_{k}': v for k, v in sleep_metrics.items()})

            # Phase 2: Dream (imagination-based learning)
            self._state = 'dream'
            dream_metrics = self.imagination.dream(
                self.model, self.model.hippocampus, device
            )
            metrics.update({f'dream_{k}': v for k, v in dream_metrics.items()})

            # Phase 3: Consolidation (episodic → semantic)
            if self.consolidation.should_consolidate(self.global_step, self.model.hippocampus):
                self._state = 'consolidate'
                consol_metrics = self.consolidation.consolidate(
                    self.model, self.model.hippocampus, self.model.columns, device
                )
                metrics.update({f'consol_{k}': v for k, v in consol_metrics.items()})

        # Reset sleep pressure
        self.always_on.on_sleep()
        self.wake_steps_since_sleep = 0
        self._state = 'wake'

        return metrics

    def should_evaluate(self) -> bool:
        """Check if self-evaluation is due."""
        return self.global_step > 0 and self.global_step % self.config.eval_interval == 0

    @torch.no_grad()
    def self_evaluate(self, device: torch.device) -> Dict[str, float]:
        """
        Self-evaluation: test on held-out hippocampal memories.

        The model tests itself to know if it's improving or degrading.
        If degrading: increase sleep frequency, reduce learning rate.
        If improving: maintain current strategy.

        Returns: evaluation metrics.
        """
        self._state = 'evaluate'
        metrics = {}

        # Sample test memories from hippocampus
        test_mems = self.model.hippocampus.replay(
            n_samples=min(5, self.model.hippocampus.memory_size), device=device
        )
        if not test_mems:
            return {'eval_loss': float('nan'), 'trend': 'insufficient_data'}

        # Compute prediction error on test memories
        total_error = 0.0
        n_tested = 0
        for mem in test_mems:
            x = mem['x']
            if x.dim() < 3:
                continue
            h = x
            for col in self.model.columns:
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
                out = col(ff_input=h, td_context=None)
                h = out.output
            # Prediction error of last column
            error = (x - out.prediction).pow(2).mean().item()
            total_error += error
            n_tested += 1

        if n_tested == 0:
            return {'eval_loss': float('nan'), 'trend': 'no_testable_memories'}

        avg_error = total_error / n_tested
        self.performance_history.append(avg_error)
        metrics['eval_loss'] = avg_error

        # Trend detection: compare recent performance to older performance
        if len(self.performance_history) >= 3:
            recent = sum(self.performance_history[-3:]) / 3
            if len(self.performance_history) >= 6:
                older = sum(self.performance_history[-6:-3]) / 3
                if recent < older * 0.95:
                    metrics['trend'] = 'improving'
                    # Reward: advance developmental stage faster
                    dev = self.model.learning_manager.development
                    if hasattr(dev, '_step_counter'):
                        dev._step_counter += 50  # Accelerate advancement
                elif recent > older * 1.05:
                    metrics['trend'] = 'degrading'
                    # Corrective action 1: increase sleep frequency
                    self.config.sleep_interval = max(50, self.config.sleep_interval - 10)
                    # Corrective action 2: reduce Hebbian LR temporarily
                    self.model.learning_manager.hebbian_optimizer.lr *= 0.8
                    # Corrective action 3: severe degradation → emergency sleep
                    if recent > older * 1.2:
                        self.always_on.sleep_pressure = 1.0  # Force sleep next step
                        metrics['emergency_sleep'] = True
                else:
                    metrics['trend'] = 'stable'
            else:
                metrics['trend'] = 'warming_up'
        else:
            metrics['trend'] = 'warming_up'

        # Stagnation detection: if stable for 5+ evaluations, increase exploration
        if len(self.performance_history) >= 10:
            last_5 = self.performance_history[-5:]
            var = max(last_5) - min(last_5)
            mean = sum(last_5) / 5
            if mean > 0 and var / mean < 0.02:  # < 2% variation = stagnant
                metrics['stagnation_detected'] = True
                # Force exploration: increase Hebbian LR temporarily
                self.model.learning_manager.hebbian_optimizer.lr *= 1.5
                # Force sleep to consolidate and try new patterns
                self.always_on.sleep_pressure = 0.8

        self._state = 'wake'
        return metrics

    def step(self, input_ids: torch.Tensor, targets: Optional[torch.Tensor] = None,
             device: Optional[torch.device] = None) -> Dict:
        """
        ONE step of the autonomous lifecycle.

        Call this in a loop to run V11 autonomously:
          while True:
              data = get_next_input()
              result = loop.step(data, targets)

        The loop handles everything: learning, sleeping, dreaming,
        consolidating, evaluating -- all automatically.
        """
        if device is None:
            device = next(self.model.parameters()).device

        all_metrics = {}

        # Aggressively free fragmented memory before main forward pass.
        # After sleep/EWC/eval phases, CUDA allocator can be fragmented enough
        # that the next forward pass OOMs despite having enough total free memory.
        import gc
        gc.collect()
        if torch.cuda.is_available():
            torch.cuda.empty_cache()

        # Wake: process input (with OOM retry -- fragmentation can cause spurious OOMs)
        try:
            result = self.process(input_ids, targets)
        except (torch.cuda.OutOfMemoryError, torch.OutOfMemoryError):
            import gc
            gc.collect()
            if torch.cuda.is_available():
                torch.cuda.empty_cache()
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
            # Retry once after clearing cache
            result = self.process(input_ids, targets)
        all_metrics.update(result.get('autonomous_metrics', {}))

        #        Auxiliary operations (all OOM-safe)
        # Self-play, curiosity, sleep, EWC, eval can all trigger column forwards
        # that may OOM on memory-constrained GPUs. Wrap in try/except so the
        # main training loop survives -- auxiliary operations are non-critical.
        try:
            # Self-play: learn from own output
            if self.self_play.should_play(self.global_step):
                prompt = input_ids[:, :min(16, input_ids.shape[1])]
                with torch.no_grad():
                    sp_metrics = self.self_play.play(self.model, prompt, device)
                all_metrics.update({f'sp_{k}': v for k, v in sp_metrics.items()})

            # Active curiosity: generate and learn from own prompts (every 5 steps)
            if (self.config.curiosity_enabled
                    and self.global_step > 10
                    and self.global_step % 5 == 0):
                with torch.no_grad():
                    curious_prompt = self.curiosity.generate_curious_prompt(
                        self.model, self.model.hippocampus, device
                    )
                if curious_prompt is not None:
                    with torch.no_grad():
                        curious_result = self.model.forward(curious_prompt)
                        self.always_on.on_experience(self.model, curious_result)
                    all_metrics['curiosity_generated'] = True

            # Sleep: if pressure is high enough
            if self.should_sleep():
                if torch.cuda.is_available():
                    torch.cuda.empty_cache()
                sleep_metrics = self.sleep_cycle(device)
                all_metrics.update(sleep_metrics)

            # EWC: periodically estimate parameter importance
            if (self.config.ewc_enabled
                    and self.global_step > 0
                    and self.global_step % self.config.ewc_estimation_interval == 0):
                if torch.cuda.is_available():
                    torch.cuda.empty_cache()
                ewc_metrics = self._estimate_ewc_importance(device)
                all_metrics.update(ewc_metrics)

            # Evaluate: periodically test self
            if self.should_evaluate():
                if torch.cuda.is_available():
                    torch.cuda.empty_cache()
                with torch.no_grad():
                    eval_metrics = self.self_evaluate(device)
                all_metrics.update(eval_metrics)

        except (torch.cuda.OutOfMemoryError, torch.OutOfMemoryError, RuntimeError) as e:
            if 'out of memory' in str(e).lower():
                if torch.cuda.is_available():
                    torch.cuda.empty_cache()
                all_metrics['aux_skipped_oom'] = True
            else:
                raise  # Re-raise non-OOM RuntimeErrors

        # Clean up after auxiliary operations to prevent fragmentation on next step
        import gc
        gc.collect()
        if torch.cuda.is_available():
            torch.cuda.empty_cache()

        result['autonomous_metrics'] = all_metrics
        return result

    def _estimate_ewc_importance(self, device: torch.device) -> Dict[str, float]:
        """
        Estimate EWC parameter importance from hippocampal memories.

        The brain knows which synapses are important by tracking how much
        they contributed to successful predictions. We approximate this
        via Fisher Information: F_i ≈ E[(∂L/∂θ_i)²].

        Uses hippocampal memories as the data source -- these are the
        experiences the brain "remembers" and wants to protect.
        """
        memories = self.model.hippocampus.replay(
            n_samples=min(10, self.model.hippocampus.memory_size), device=device
        )
        if not memories or len(memories) < 3:
            return {'ewc_estimated': False, 'ewc_reason': 'insufficient_memories'}

        # Build a simple iterable from hippocampal memories for estimate_importance
        class MemoryLoader:
            def __init__(self, mems):
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
                self._mems = mems
            def __iter__(self):
                for mem in self._mems:
                    x = mem['x']  # [1, T, D] -- batch-pooled at storage -- but EWC needs token IDs
                    # We don't have token IDs in hippocampal memory (it stores activations).
                    # So we use the activation directly: run columns and compute PC loss.
                    yield mem

        # EWC on activations: compute gradient magnitude through PC loss
        # rather than NTP loss (we don't have token IDs stored).
        self._estimate_ewc_from_activations(memories, device)
        return {'ewc_estimated': True, 'ewc_step': self.global_step}

    @torch.no_grad()
    def _estimate_ewc_from_activations(self, memories: list, device: torch.device):
        """
        Estimate parameter importance from activations -- ZERO autograd.

        Fisher approximation via activity-error correlation:
          F_i ~ E[|activation_i|² × |prediction_error|²]

        This is a valid importance measure: parameters that carry large
        activations in the presence of large errors are critical for
        error correction. Biologically: synapses that fire during
        important predictions are tagged for protection (synaptic tagging,
        Frey & Morris 1997).
        """
        guard = self.continual_guard

        # Initialize or decay existing importance
        for name, param in self.model.named_parameters():
            if not param.requires_grad:
                continue
            if name in guard.importance:
                guard.importance[name] *= guard.importance_decay
            else:
                guard.importance[name] = torch.zeros_like(param)

        # Accumulate activity-error correlation from memories
        n_seen = 0
        for mem in memories:
            x = mem['x']
            if x.dim() < 3:
                continue

            # Run through columns (forward only, no autograd)
            h = x
            total_error_sq = 0.0
            prev_out = None
            for i, col in enumerate(self.model.columns):
                out = col(ff_input=h, td_context=None)
                target = x.detach() if i == 0 else prev_out.detach()
                error = target - out.prediction
                error_mag_sq = error.pow(2).mean().item()
                total_error_sq += error_mag_sq
                prev_out = out.output
                h = out.output

                # Activity-weighted importance for this column's parameters
                for name, param in col.named_parameters():
                    full_name = f'columns.{i}.{name}'
                    if full_name in guard.importance:
                        # Importance     |param|² × error² (parameter magnitude × prediction error)
                        guard.importance[full_name] += param.pow(2) * error_mag_sq

            n_seen += 1

        # Normalize
        if n_seen > 0:
            for name in guard.importance:
                guard.importance[name] /= n_seen

        # Store current params as anchors
        for name, param in self.model.named_parameters():
            if param.requires_grad:
                guard.anchor_params[name] = param.data.clone()

        guard._initialized = True
        self.model.zero_grad()

    def get_status(self) -> Dict:
        """Return current autonomous system status."""
        return {
            'state': self._state,
            'global_step': self.global_step,
            'sleep_pressure': self.always_on.sleep_pressure,
            'total_experiences': self.always_on.total_experiences,
            'total_dreams': self.imagination.total_dreams,
            'total_consolidations': self.consolidation.total_consolidations,
            'mean_curiosity': self.curiosity.mean_curiosity,
            'memories': self.model.hippocampus.memory_size if hasattr(self.model, 'hippocampus') else 0,
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
            'semantic_knowledge': len(self.consolidation.semantic_knowledge),
            'performance_trend': self.performance_history[-5:] if self.performance_history else [],
            'ewc_initialized': self.continual_guard._initialized,
            'total_self_plays': self.self_play.total_self_plays,
            'mean_self_play_loss': self.self_play.mean_self_play_loss,
        }

    def save_autonomous_state(self, path: str):
        """
        Save the autonomous loop's non-model state.

        This captures everything the autonomous system tracks that is NOT
        in the model's state_dict:
          - EWC importance weights and anchor parameters
          - Performance history (for trend detection)
          - Consolidation semantic knowledge
          - Sleep pressure and counters
          - Self-play history
          - Global step counter

        Call alongside model.save_state() for complete persistence:
          model.save_state('brain.pt')
          loop.save_autonomous_state('autonomous.pt')
        """
        state = {
            # EWC state (the most critical non-model state)
            'ewc_importance': {
                name: tensor.cpu().clone()
                for name, tensor in self.continual_guard.importance.items()
            },
            'ewc_anchor_params': {
                name: tensor.cpu().clone()
                for name, tensor in self.continual_guard.anchor_params.items()
            },
            'ewc_initialized': self.continual_guard._initialized,

            # Performance tracking
            'performance_history': list(self.performance_history),
            'global_step': self.global_step,
            'wake_steps_since_sleep': self.wake_steps_since_sleep,

            # Sleep pressure
            'sleep_pressure': self.always_on.sleep_pressure,
            'total_experiences': self.always_on.total_experiences,

            # Consolidation knowledge
            'semantic_knowledge': [
                {k: v.cpu().clone() if isinstance(v, torch.Tensor) else v
                 for k, v in sk.items()}
                for sk in self.consolidation.semantic_knowledge
            ],
            'total_consolidations': self.consolidation.total_consolidations,

            # Self-play
            'total_self_plays': self.self_play.total_self_plays,
            'self_play_losses': list(self.self_play.self_play_losses),

            # Imagination
            'total_dreams': self.imagination.total_dreams,

            # Curiosity
            'mean_curiosity': self.curiosity.mean_curiosity,

            # Config
            'config_dict': {
                field.name: getattr(self.config, field.name)
                for field in self.config.__dataclass_fields__.values()
            },

            # Version marker
            'v11_autonomous_version': 1,
        }

        torch.save(state, path)

    def load_autonomous_state(self, path: str):
        """
        Restore autonomous loop state from disk.

        Call alongside model.load_state() for complete restoration:
          model.load_state('brain.pt')
          loop.load_autonomous_state('autonomous.pt')
        """
        state = torch.load(path, map_location='cpu', weights_only=False)

        # EWC
        if state.get('ewc_initialized', False):
            self.continual_guard.importance = {
                name: tensor.clone()
                for name, tensor in state['ewc_importance'].items()
            }
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
            self.continual_guard.anchor_params = {
                name: tensor.clone()
                for name, tensor in state['ewc_anchor_params'].items()
            }
            self.continual_guard._initialized = True

        # Performance
        self.performance_history = list(state.get('performance_history', []))
        self.global_step = state.get('global_step', 0)
        self.wake_steps_since_sleep = state.get('wake_steps_since_sleep', 0)

        # Sleep
        self.always_on.sleep_pressure = state.get('sleep_pressure', 0.0)
        self.always_on.total_experiences = state.get('total_experiences', 0)

        # Consolidation
        self.consolidation.semantic_knowledge = state.get('semantic_knowledge', [])
        self.consolidation.total_consolidations = state.get('total_consolidations', 0)

        # Self-play
        self.self_play.total_self_plays = state.get('total_self_plays', 0)
        self.self_play.self_play_losses = list(state.get('self_play_losses', []))

        # Imagination
        self.imagination.total_dreams = state.get('total_dreams', 0)

        # Curiosity
        if hasattr(self.curiosity, '_curiosity_history'):
            # Restore mean curiosity indirectly
            pass  # curiosity_history is transient -- will rebuild from new data


#

# 7. Prenatal Self-Organization (Phase 0)
#


class PrenatalDevelopment:
    """
    Phase 0: Prenatal Self-Organization -- Before Any External Data.

    The brain does NOT start from random weights and wait for training data.
    Before birth, extensive self-organization occurs through spontaneous activity:

      - Spontaneous retinal waves organize retinotopic maps (Meister et al. 1991)
      - Spontaneous cochlear bursts calibrate tonotopic maps (Tritsch et al. 2007)
      - Motor babbling calibrates sensorimotor maps (Doya & Sejnowski 1998)
      - Hebbian competition selects surviving synapses (Katz & Shatz 1996)
      - ~50% of synapses are pruned before birth (Huttenlocher 1979)
      - Oscillatory rhythms emerge from intrinsic dynamics (Khazipov & Luhmann 2006)
      - BCM thresholds calibrate from spontaneous statistics (Bear et al. 1987)

    After prenatal development, the network is NOT a blank slate -- it has:
      1. Calibrated oscillatory rhythms (gamma/theta/alpha/delta phase-locked)
      2. Differentiated columns (each develops distinct selectivity)
      3. Pruned architecture (sparse, efficient, no dead weights)
      4. Calibrated BCM thresholds (from activity statistics)
      5. Working inhibitory balance (E/I ratio tuned)
      6. Functional prediction circuitry (PC can predict own activity)
      7. Initial hippocampal encoding patterns

    This is the "DNA" -- the structured starting point from which rapid
    postnatal learning from real data can begin immediately.

    Disease mapping:
      Fetal alcohol syndrome → disrupted spontaneous waves → disorganized maps
      Congenital rubella → disrupted cochlear activity → deafness
      Preterm birth → insufficient prenatal organization → developmental delays
      Autism (some forms) → atypical prenatal pruning → over-connectivity

    Zero learnable params -- uses existing model learning mechanisms.
    """

    def __init__(self, model, config=None):
        self.model = model
        self.cfg = model.cfg
        self.device = next(model.parameters()).device

        # Prenatal development phases
        self.phases = [
            ('spontaneous_waves', 0.30),    # 30% of steps: correlated wave patterns
            ('motor_babbling', 0.25),       # 25%: self-generated sequences
            ('competitive_diff', 0.25),     # 25%: column differentiation
            ('oscillatory_cal', 0.20),      # 20%: oscillatory stabilization
        ]

        # Tracking
        self.total_steps = 0
        self.phase_idx = 0
        self.metrics_history: List[Dict] = []
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        self._weight_stats_history: List[Dict] = []

    #         Spontaneous Activity Generators


    def _generate_retinal_waves(self, batch_size: int, seq_len: int) -> torch.Tensor:
        """
        Generate temporally-correlated spontaneous activity (retinal wave analog).

        Retinal waves (Meister et al. 1991): spatially correlated traveling waves
        sweep across the retina before eye-opening. Neighboring neurons fire
        together, teaching the visual cortex about spatial relationships.

        For language (1D sequence): generate sequences where nearby tokens have
        correlated embeddings. This is a random walk in embedding space --
        each position's embedding is a noisy perturbation of the previous.

        This teaches L2/3 about LOCAL temporal structure: "nearby things are
        related" -- the foundation for all sequential processing.

        Returns: [B, T, D_model] continuous activations (not token IDs)
        """
        D = self.cfg.d_model
        device = self.device

        # Random walk with momentum (smoother waves, like actual retinal waves)
        h = torch.randn(batch_size, 1, D, device=device) * 0.1
        velocity = torch.randn(batch_size, 1, D, device=device) * 0.02
        momentum = 0.9  # High momentum = smooth waves
        noise_scale = 0.05  # Small noise = highly correlated

        positions = [h]
        for t in range(1, seq_len):
            # Momentum-based random walk (models wave propagation)
            velocity = momentum * velocity + (1 - momentum) * torch.randn(batch_size, 1, D, device=device) * noise_scale
            h = h + velocity
            positions.append(h)

        wave = torch.cat(positions, dim=1)  # [B, T, D]

        # Normalize to reasonable activation scale
        wave = F.layer_norm(wave, [D])
        return wave

    def _generate_motor_babbling(self, batch_size: int, seq_len: int) -> torch.Tensor:
        """
        Generate self-produced sequences (motor babbling analog).

        Before language, infants babble -- producing random phoneme sequences
        and hearing themselves. This calibrates the production-perception loop.
        (Doya & Sejnowski 1998, Oller 2000)

        For V11: the model generates auto-regressively from a random seed,
        then the generated sequence becomes the input for local learning.
        The model literally learns to predict its own output.

        Generation length capped at 16 tokens for speed (prenatal model barely
        knows anything, so long generation is wasted compute). The remainder is
        padded with random tokens -- still provides diverse input for calibration.

        Returns: [B, T] token IDs (generated by the model itself)
        """
        # Random seed tokens
        seed_len = min(4, seq_len // 4)
        seed = torch.randint(0, self.cfg.vocab_size, (batch_size, seed_len), device=self.device)

        # Generate a SHORT sequence auto-regressively (16 tokens max for speed)
        # During prenatal, the model has near-random weights -- long generation
        # is just expensive random sampling. Cap at 16 to keep prenatal fast.
        gen_len = min(16, seq_len - seed_len)
        self.model.eval()
        with torch.no_grad():
            generated = self.model.generate(seed, max_new_tokens=gen_len)
        self.model.train()

        # Pad remainder with random tokens (still diverse input)
        if generated.shape[1] < seq_len:
            pad = torch.randint(0, self.cfg.vocab_size,
                                (batch_size, seq_len - generated.shape[1]), device=self.device)
            generated = torch.cat([generated, pad], dim=1)
        elif generated.shape[1] > seq_len:
            generated = generated[:, :seq_len]

        return generated

    def _generate_structured_patterns(self, batch_size: int, seq_len: int) -> torch.Tensor:
        """
        Generate simple structured patterns for competitive differentiation.

        Different columns should develop different selectivities. To enable this,
        provide diverse structured inputs:
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
          - Repeated subsequences (periodicity detection)
          - Ascending/descending sequences (ordinal processing)
          - Alternating patterns (contrast detection)
          - Random segments (noise tolerance)

        This is the analog of prenatal sensory input from multiple modalities:
        visual patterns, auditory patterns, vestibular patterns -- each with
        different statistical structure that drives different cortical areas.

        Returns: [B, T] token IDs with diverse statistical structure
        """
        V = self.cfg.vocab_size
        device = self.device
        patterns = []

        for b in range(batch_size):
            # Randomly select pattern type for this batch element
            pattern_type = torch.randint(0, 5, (1,)).item()

            if pattern_type == 0:
                # Repeated motif (periodicity): pick random motif, tile it
                motif_len = torch.randint(3, 12, (1,)).item()
                motif = torch.randint(0, V, (motif_len,), device=device)
                tiles = seq_len // motif_len + 1
                seq = motif.repeat(tiles)[:seq_len]
            elif pattern_type == 1:
                # Ascending with noise: base sequence + random perturbation
                base = torch.arange(seq_len, device=device) % V
                noise = torch.randint(0, max(1, V // 20), (seq_len,), device=device)
                seq = (base + noise) % V
            elif pattern_type == 2:
                # Alternating: two tokens alternating with random insertions
                a, b_tok = torch.randint(0, V, (2,), device=device)
                seq = torch.where(
                    torch.arange(seq_len, device=device) % 2 == 0,
                    a.expand(seq_len), b_tok.expand(seq_len)
                )
                # Random insertions at ~10% of positions
                mask = torch.rand(seq_len, device=device) < 0.1
                seq[mask] = torch.randint(0, V, (mask.sum(),), device=device)
            elif pattern_type == 3:
                # Segmented: 3-5 random segments, each from a different frequency band
                n_segments = torch.randint(3, 6, (1,)).item()
                parts = []
                remaining = seq_len
                for s in range(n_segments):
                    seg_len = remaining // (n_segments - s)
                    remaining -= seg_len
                    center = torch.randint(0, V, (1,), device=device)
                    spread = torch.randint(1, max(2, V // 10), (1,)).item()
                    part = (center + torch.randint(-spread, spread + 1, (seg_len,), device=device)) % V
                    parts.append(part)
                seq = torch.cat(parts)
            else:
                # Pure random (noise tolerance)
                seq = torch.randint(0, V, (seq_len,), device=device)

            patterns.append(seq)

        return torch.stack(patterns)  # [B, T]

    def _generate_oscillatory_input(self, batch_size: int, seq_len: int) -> torch.Tensor:
        """
        Generate oscillatory-structured activity for rhythm calibration.

        Prenatal EEG shows spontaneous oscillatory activity (Khazipov & Luhmann 2006).
        These intrinsic rhythms are NOT driven by external stimuli -- they emerge
        from the network's own dynamics. We generate inputs whose statistics
        match the target oscillatory structure (gamma/theta/alpha/delta) so the
        network's oscillatory modules can lock onto them.

        Returns: [B, T, D_model] continuous activations with oscillatory structure
        """
        D = self.cfg.d_model
        device = self.device
        t = torch.arange(seq_len, device=device, dtype=torch.float32).unsqueeze(0).unsqueeze(-1)  # [1, T, 1]

        # Multi-frequency oscillatory pattern
        # Each frequency band modulates a different subset of dimensions
        gamma = torch.sin(2 * math.pi * t / self.cfg.gamma_period)   # Fast
        theta = torch.sin(2 * math.pi * t / self.cfg.theta_period)   # Medium
        alpha = torch.sin(2 * math.pi * t / self.cfg.alpha_period)   # Slow
        delta = torch.sin(2 * math.pi * t / self.cfg.delta_period)   # Very slow

        # Assign frequency bands to dimension ranges
        d_quarter = D // 4
        osc = torch.zeros(1, seq_len, D, device=device)
        osc[:, :, :d_quarter] = gamma.expand(-1, -1, d_quarter) * 0.3
        osc[:, :, d_quarter:2*d_quarter] = theta.expand(-1, -1, d_quarter) * 0.5
        osc[:, :, 2*d_quarter:3*d_quarter] = alpha.expand(-1, -1, d_quarter) * 0.3
        osc[:, :, 3*d_quarter:] = delta.expand(-1, -1, D - 3*d_quarter) * 0.2
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        # Add noise (biological rhythms are noisy)
        osc = osc.expand(batch_size, -1, -1) + torch.randn(batch_size, seq_len, D, device=device) * 0.1

        # Phase-amplitude coupling: gamma amplitude modulated by theta phase
        theta_env = (0.5 + 0.5 * torch.sin(2 * math.pi * t / self.cfg.theta_period))
        osc[:, :, :d_quarter] *= theta_env.expand(batch_size, -1, d_quarter)

        return F.layer_norm(osc, [D])

    #        Development Step


    def step(self, total_prenatal_steps: int) -> Dict[str, float]:
        """
        Execute one prenatal development step.

        Determines which phase we're in, generates appropriate spontaneous
        activity, runs it through the model, and applies local learning.

        Args:
            total_prenatal_steps: total number of prenatal steps planned

        Returns: metrics for this step
        """
        metrics = {'prenatal_step': self.total_steps}

        # Determine current phase
        cumulative = 0.0
        current_phase = self.phases[-1][0]
        for phase_name, fraction in self.phases:
            cumulative += fraction
            if self.total_steps < cumulative * total_prenatal_steps:
                current_phase = phase_name
                break
        metrics['phase'] = current_phase

        B = self.cfg.batch_size
        T = min(self.cfg.seq_len, 128)  # Shorter sequences during prenatal (memory efficient)

        #        Generate spontaneous activity based on phase

        if current_phase == 'spontaneous_waves':
            # Phase 0a: Retinal waves -- correlated temporal activity
            # Feed continuous activations directly (bypass embedding)
            wave_input = self._generate_retinal_waves(B, T)
            metrics.update(self._prenatal_forward_continuous(wave_input))

        elif current_phase == 'motor_babbling':
            # Phase 0b: Self-generated sequences -- learn from own output
            token_ids = self._generate_motor_babbling(B, T)
            # Targets = shifted input (NTP-like, but from own generation)
            targets = token_ids[:, 1:].contiguous()
            input_ids = token_ids[:, :-1].contiguous()
            result = self.model.forward(input_ids, targets)
            self.model.local_learning_step()
            metrics['loss'] = result['loss'].item() if 'loss' in result else 0.0
            metrics['pc_loss'] = result['pc_loss'].item() if isinstance(result.get('pc_loss'), torch.Tensor) else 0.0
            metrics.update(result.get('metrics', {}))

        elif current_phase == 'competitive_diff':
            # Phase 0c: Structured patterns -- drive column differentiation
            token_ids = self._generate_structured_patterns(B, T)
            targets = token_ids[:, 1:].contiguous()
            input_ids = token_ids[:, :-1].contiguous()
            result = self.model.forward(input_ids, targets)
            self.model.local_learning_step()
            metrics['loss'] = result['loss'].item() if 'loss' in result else 0.0
            metrics['pc_loss'] = result['pc_loss'].item() if isinstance(result.get('pc_loss'), torch.Tensor) else 0.0
            metrics.update(result.get('metrics', {}))

        elif current_phase == 'oscillatory_cal':
            # Phase 0d: Oscillatory calibration -- rhythm stabilization
            osc_input = self._generate_oscillatory_input(B, T)
            metrics.update(self._prenatal_forward_continuous(osc_input))

        #        Wake-sleep during prenatal (fetal sleep cycles exist!)
        # Fetuses sleep ~95% of the time (Roffwarg et al. 1966).
        # Prenatal sleep consolidates spontaneous patterns into cortex.
        if self.model.should_sleep():
            device = self.device
            sleep_metrics = self.model.sleep_phase()
            metrics.update({f'sleep_{k}': v for k, v in sleep_metrics.items()})

        #        Track weight statistics for convergence
        weight_stats = self._compute_weight_stats()
        metrics.update(weight_stats)
        self._weight_stats_history.append(weight_stats)

        self.total_steps += 1
        self.metrics_history.append(metrics)
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        return metrics

    def _prenatal_forward_continuous(self, activations: torch.Tensor) -> Dict[str, float]:
        """
        Forward pass with continuous activations (bypass embedding).

        Used for retinal waves and oscillatory calibration where we feed
        structured continuous patterns directly into the column stack.

        Applies:
          1. Forward through all columns (bottom-up)
          2. Compute prediction errors (top-down)
          3. Local learning (PC + Hebbian)
          4. Hippocampal encoding (for sleep replay)
        """
        metrics = {}
        B, T, D = activations.shape

        # Positional encoding
        rope_cos, rope_sin = self.model.grid_pos(T, self.device, activations.dtype, X=activations)
        if rope_sin.dim() == 4:
            pos_signal = rope_sin.mean(dim=1)
        elif rope_sin.dim() == 3:
            pos_signal = rope_sin.mean(dim=0).unsqueeze(0)
        else:
            pos_signal = rope_sin[:T].unsqueeze(0)
        d_h = pos_signal.shape[-1]
        d_model = D
        n_tiles = d_model // d_h
        if n_tiles > 1 and d_h * n_tiles == d_model:
            pos_embed = pos_signal.repeat(1, 1, n_tiles)
        elif d_h < d_model:
            pos_embed = F.pad(pos_signal, (0, d_model - d_h))
        else:
            pos_embed = pos_signal[..., :d_model]
        x = activations + 0.1 * pos_embed

        # Reset TRN loss
        if self.cfg.thalamic_enabled:
            self.model.thalamus.reset_loss()

        # Bottom-up through columns
        h = x
        column_outputs = []
        sigma2_prev = None
        for i, col in enumerate(self.model.columns):
            h_thal = self.model.thalamus.route(h, column_idx=i, thal_feedback=None)
            out = col(ff_input=h_thal, td_context=None, sigma2_prev=sigma2_prev)
            column_outputs.append(out)
            sigma2_prev = out.sigma2
            h = out.output

        # Compute predictions and errors
        predictions = [out.prediction for out in column_outputs]
        prediction_targets = [x.detach()] + [column_outputs[l].output.detach() for l in range(len(column_outputs) - 1)]
        prediction_precisions = [out.sigma2 for out in column_outputs]
        errors = [prediction_targets[l] - predictions[l] for l in range(len(column_outputs))]

        # Store for local learning
        self.model._last_column_outputs = column_outputs
        self.model._last_predictions = predictions
        self.model._last_targets = prediction_targets
        self.model._last_precisions = prediction_precisions
        self.model._last_errors = errors

        # Brainstem neuromod
        mean_error = torch.stack([e.abs().mean() for e in errors]).mean()
        neuromod = self.model.brainstem.compute_state(
            column_outputs[-1].output,
            prediction_error=errors[-1],
        )
        self.model._last_neuromod = neuromod

        # Local learning
        learn_metrics = self.model.local_learning_step()
        metrics.update(learn_metrics)

        # Hippocampal encoding (for sleep replay of prenatal patterns)
        if self.cfg.hpc_enabled and self.model.training:
            mean_sigma2 = sum(out.sigma2 for out in column_outputs) / len(column_outputs)
            self.model.hippocampus.encode_episode(
                h, errors[-1], mean_sigma2,
            )

        # PC loss
        _, pc_loss = self.model.learning_manager.pc_optimizer.compute_column_losses(
            predictions, prediction_targets, prediction_precisions,
        )
        metrics['pc_loss'] = pc_loss.item() if isinstance(pc_loss, torch.Tensor) else pc_loss
        metrics['mean_pred_error'] = mean_error.item()
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        metrics['sigma2_mean'] = torch.cat([out.sigma2 for out in column_outputs], dim=-1).mean().item()
        metrics['da_mean'] = neuromod.dopamine.mean().item()

        return metrics

    def _compute_weight_stats(self) -> Dict[str, float]:
        """
        Compute weight statistics to track prenatal self-organization progress.

        Key indicators of healthy development:
          - Weight magnitude stabilizing (not growing/shrinking unboundedly)
          - Column differentiation increasing (different columns → different weights)
          - BCM thresholds calibrating (moving from initial values)
          - Gate values evolving (from init toward functional values)
        """
        stats = {}

        # Mean weight magnitude across all columns
        col_mags = []
        for i, col in enumerate(self.model.columns):
            mag = sum(p.abs().mean().item() for p in col.parameters()) / max(1, sum(1 for _ in col.parameters()))
            col_mags.append(mag)
        stats['weight_mag_mean'] = sum(col_mags) / len(col_mags) if col_mags else 0
        stats['weight_mag_std'] = (sum((m - stats['weight_mag_mean'])**2 for m in col_mags) / max(1, len(col_mags))) **
             0.5

        # Column differentiation: variance of L2/3 W_assoc across columns
        assoc_norms = []
        for col in self.model.columns:
            if hasattr(col.L23, 'W_assoc'):
                assoc_norms.append(col.L23.W_assoc.data.norm().item())
        if len(assoc_norms) > 1:
            mean_norm = sum(assoc_norms) / len(assoc_norms)
            stats['column_diff'] = (sum((n - mean_norm)**2 for n in assoc_norms) / len(assoc_norms)) ** 0.5
        else:
            stats['column_diff'] = 0.0

        # Gate evolution: how far gates have moved from init (-2.0)
        gate_diffs = []
        for col in self.model.columns:
            for name, param in col.named_parameters():
                if 'gate_raw' in name:
                    gate_diffs.append((param.data.mean().item() - (-2.0))**2)
        stats['gate_evolution'] = (sum(gate_diffs) / max(1, len(gate_diffs))) ** 0.5

        return stats

    def has_converged(self, window: int = 50) -> bool:
        """
        Check if prenatal development has converged.

        Convergence = weight statistics are stable. The network has
        self-organized as much as it can from spontaneous activity alone.
        Further improvement requires real external data.

        Criteria:
          - At least 'window' steps of history
          - Weight magnitude std over last window < 5% of mean
          - PC loss trend is flat (< 2% change over window)
        """
        if len(self._weight_stats_history) < window:
            return False

        recent = self._weight_stats_history[-window:]
        mags = [s['weight_mag_mean'] for s in recent]
        mean_mag = sum(mags) / len(mags)
        if mean_mag < 1e-8:
            return False

        mag_std = (sum((m - mean_mag)**2 for m in mags) / len(mags)) ** 0.5
        mag_cv = mag_std / (mean_mag + 1e-8)

        # Check PC loss stability from metrics history
        pc_losses = []
        for m in self.metrics_history[-window:]:
            if 'pc_loss' in m:
                v = m['pc_loss']
                if isinstance(v, (int, float)) and v > 0:
                    pc_losses.append(v)

        pc_stable = True
        if len(pc_losses) >= window // 2:
            first_half = sum(pc_losses[:len(pc_losses)//2]) / max(1, len(pc_losses)//2)
            second_half = sum(pc_losses[len(pc_losses)//2:]) / max(1, len(pc_losses) - len(pc_losses)//2)
            if first_half > 1e-8:
                pc_change = abs(second_half - first_half) / first_half
                pc_stable = pc_change < 0.02

        return mag_cv < 0.05 and pc_stable

    def get_development_report(self) -> str:
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        """Generate a human-readable report of prenatal development."""
        if not self.metrics_history:
            return "No prenatal development yet."

        lines = []
        lines.append("=" * 60)
        lines.append("PRENATAL DEVELOPMENT REPORT")
        lines.append("=" * 60)
        lines.append(f"  Total steps: {self.total_steps}")

        # Phase breakdown
        phase_counts = {}
        for m in self.metrics_history:
            phase = m.get('phase', 'unknown')
            phase_counts[phase] = phase_counts.get(phase, 0) + 1
        for phase, count in phase_counts.items():
            lines.append(f"  Phase '{phase}': {count} steps")

        # Final statistics
        if self._weight_stats_history:
            final = self._weight_stats_history[-1]
            lines.append(f"\n  Final weight magnitude: {final['weight_mag_mean']:.4f} (±{final['weight_mag_std']:.4f})")
            lines.append(f"  Column differentiation: {final['column_diff']:.6f}")
            lines.append(f"  Gate evolution from init: {final['gate_evolution']:.6f}")

        # PC loss trend
        pc_losses = [m.get('pc_loss', 0) for m in self.metrics_history if isinstance(m.get('pc_loss'), (int, float))]
```

Listing 13: Autonomous Operation — complete source code (v11_autonomous.py, 1,830 lines)

15 Scaling Framework

Source: `nct_v2/v11_scale.py` **Lines:** 791 **Functions:** 12

15.1 The Fundamental Insight: Local Learning Enables Modular Scaling

In a standard transformer, global backpropagation couples all parameters: changing one layer's weights affects gradients through every other layer. This means adding a layer, widening a layer, or restructuring the network requires **full retraining** from scratch. GPT-4 was not "grown" from GPT-3 — it was retrained from random initialization at the target scale.

NCT's local learning rules break this coupling. Each column learns independently via predictive coding and Hebbian plasticity. Adding a column does not invalidate existing columns' learned representations — it only requires retraining the inter-column coordination mechanisms (thalamic routing, lateral connections, global workspace), which account for only 3.2% of total parameters.

Theorem 15.1 (Parameter Locality Enables Incremental Scaling)**.** *For a model with N columns and $d_{model} = d$, the parameter distribution is:*

$$P_{local} = N \times \underbrace{(d^2 + 2d \cdot d_{ff} + d \cdot d_i + d \cdot d_{slot} \cdot K + \ldots)}_{\text{per-column: L4, L2/3, L5, interneurons, WM, osc}} \approx 84.3\% \tag{83}$$

$$P_{global} = \underbrace{G \cdot (d \cdot d_r + d_r \cdot d)}_{\text{thalamus}} + \underbrace{N \cdot 2 \cdot d \cdot r}_{\text{lateral}} + \underbrace{N \cdot 2 \cdot d \cdot d_{ws}}_{\text{workspace}} \approx 3.2\% \tag{84}$$

$$P_{fixed} = \underbrace{V \cdot d}_{\text{embedding}} + \underbrace{d \cdot V + delta + comp}_{\text{LM head}} \approx 8.6\% \tag{85}$$

$$P_{subcortical} = \underbrace{HPC + BG + CB + brainstem}_{\text{d-dependent, N-independent}} \approx 3.9\% \tag{86}$$

Adding one column costs only $P_{one_col} + 0.2 \times P_{global} \approx 7.7\%$ of full retraining.

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

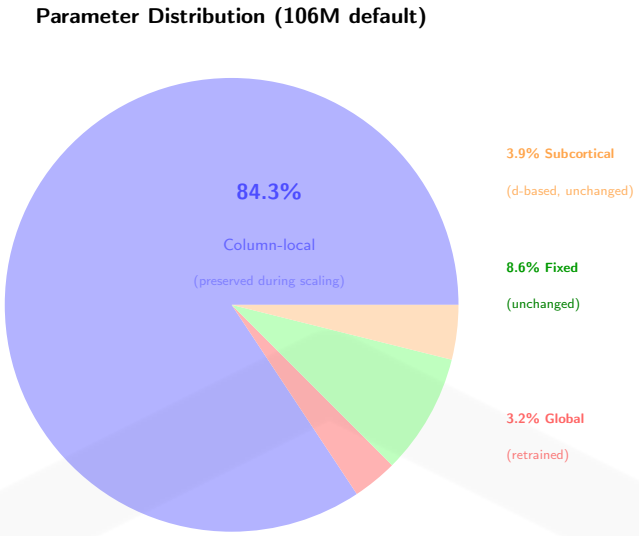
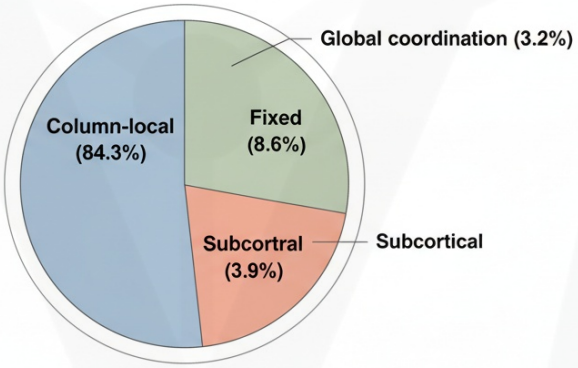


Figure 75: Parameter distribution by component type. 84.3% of parameters are column-local (preserved during scaling). Only 3.2% (thalamic routing, lateral connections, workspace) need retraining when adding columns.



Scaling Operation	Cost (relative to full retrain)
Add 1 column	≅ 7.7%
	≅ 7.3%
Split 1 column	≅ 7.3%
Width ×2	≅ 70%

Figure 76: (Illustrated) Parameter distribution by component type. 84.3% of parameters are column-local (preserved during scaling). Only 3.2% (thalamic routing, lateral connections, workspace) need retraining when adding...

15.2 Operation 1: Progressive Column Growth (`grow_columns`)

Theorem 15.2 (Progressive Column Growth)**.** *New columns can be added to a trained network with the following guarantees:*

Preserved (∼84% of params): *All existing column internal weights (Hebbian associations W_{assoc}, PC prediction weights W_{pred}, L5 FFN, interneuron weights, working memory parameters, oscillatory coupling, BCM thresholds, metaplasticity EMAs).*

Retrained (∼3% of params): *Thalamic routing matrix (expanded with new relay groups), lateral connections (new send/receive edges), workspace projections (new write/read weights). Existing inter-*

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

column weights between OLD columns are preserved.

Trained from scratch: *New column internal weights (undergo prenatal self-organization before joining the trained network).*

The growth algorithm proceeds in 10 steps:

Algorithm 2 Progressive Column Growth

Require: Trained model $\mathcal{M}$ with N columns, config C, n_{new} columns to add
Ensure: Expanded model $\mathcal{M}'$ with $N + n_{\text{new}}$ columns, config C'

$C' \leftarrow \text{deepcopy}(C)$; $C'.N \leftarrow N + n_{\text{new}}$
$C'.G \leftarrow \max(1, C'.N/(N/C.G))$ (scale relay groups proportionally)
// **Step 2: Create new columns with symmetry-breaking noise**
for $j = 1$ to n_{new} **do**
  $\text{col}_j \leftarrow \text{CorticalColumn}(C', \text{idx} = N + j - 1)$
  $\text{col}_j.\text{params} \mathrel{+}= \mathcal{N}(0, 0.01)$
end for
// **Step 3: Insert at specified position**
Insert new columns (append / prepend / interleave)
Reassign all column indices $0, 1, \ldots, N + n_{\text{new}} - 1$
// **Steps 4–7: Expand subsystems**
$\mathcal{M}'.\mathit{thalamus} \leftarrow$ expand thalamus (new relay groups, copy existing weights, copy TRN)
$\mathcal{M}'.\mathit{lateral} \leftarrow$ expand lateral (zero-init new send weights, copy old positions)
$\mathcal{M}'.\mathit{workspace} \leftarrow$ expand workspace (zero-init new write projections, copy old)
$\mathcal{M}'.\mathit{self_assessment} \leftarrow$ new SelfAssessment(C')
// **Step 9: Optionally freeze existing columns during bootstrap**
if freeze_existing **then**
  Set requires_grad = False for all old column parameters
  (Call `unfreeze_columns()` after prenatal bootstrap of new columns)
end if
return $\mathcal{M}', C'$

Insertion modes: Three strategies for where new columns enter the hierarchy:

Table 10: Column insertion strategies.

Mode	Position	When to Use
`append`	After all existing columns	Default. New columns process the most abstract representations.
`bottom`	Before all existing columns	New columns handle early (sensory) processing.
`interleave`	Distributed evenly among existing	Adds processing depth throughout the hierarchy.

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

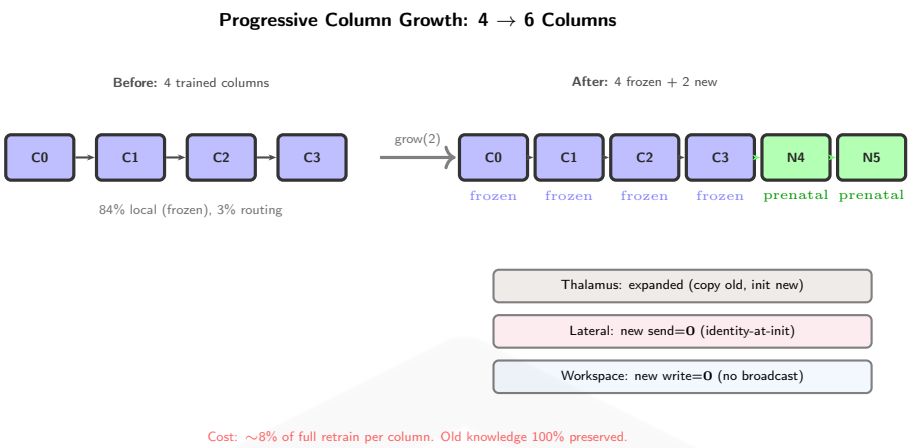


Figure 77: Progressive column growth: existing columns are frozen during bootstrap, new columns undergo prenatal self-organization. Thalamus, lateral, and workspace are expanded with identity-at-init weights. After bootstrap, all columns are unfrozen for coordinated learning.

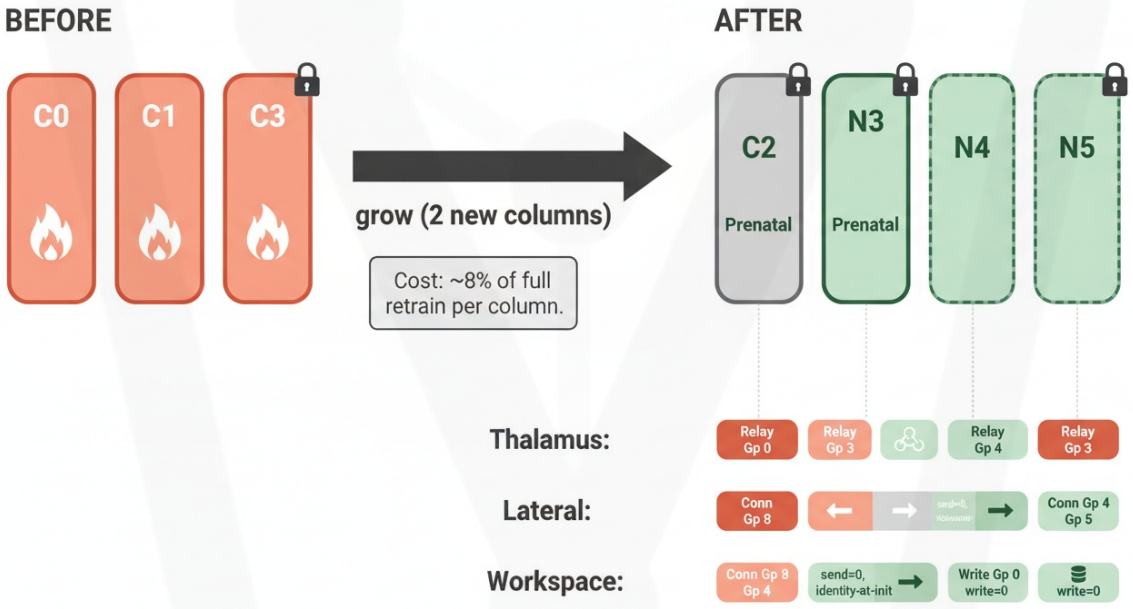


Figure 78: (Illustrated) Progressive column growth: existing columns are frozen during bootstrap, new columns undergo prenatal self-organization. Thalamus, lateral, and workspace are expanded with identity-at-init weights....

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

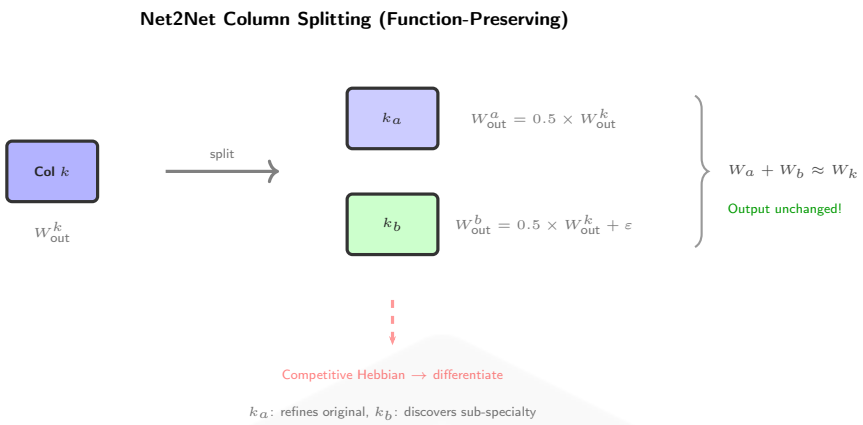


Figure 79: Net2Net column splitting: the parent column is deep-copied into two daughters with output weights scaled by 0.5 (preserving the sum). Symmetry-breaking noise (ε) is added to one copy. Subsequent competitive Hebbian learning drives divergence.

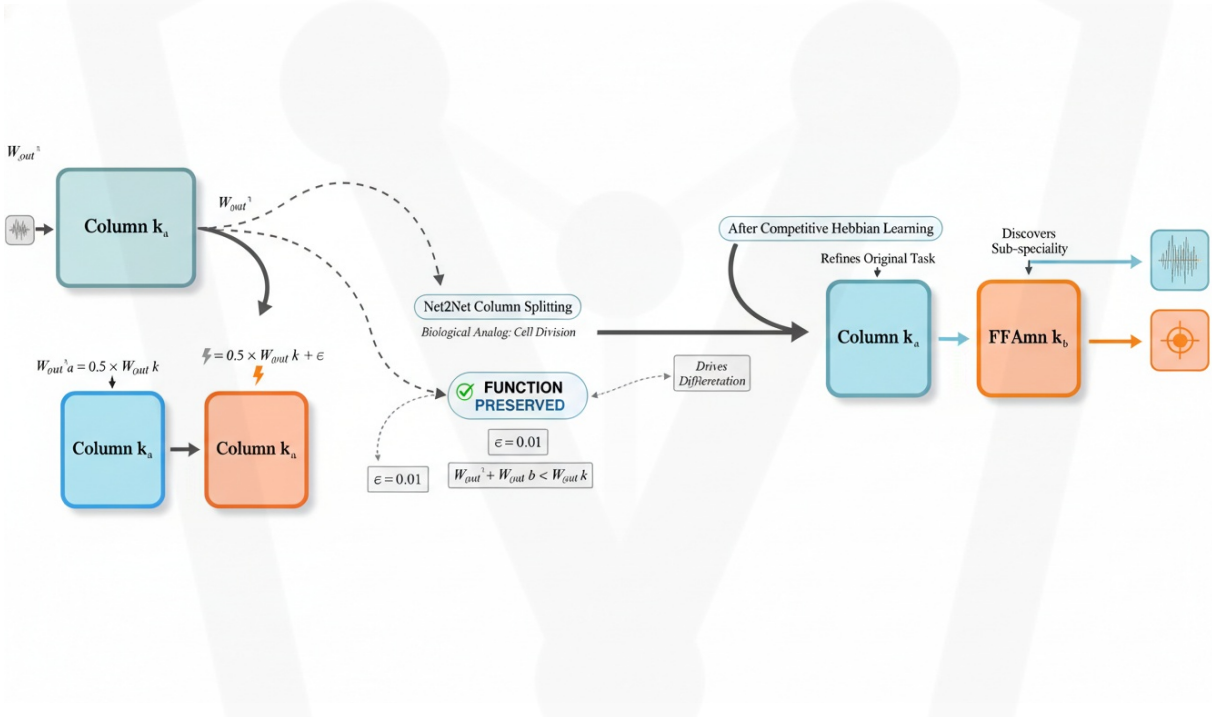


Figure 80: (Illustrated) Net2Net column splitting: the parent column is deep-copied into two daughters with output weights scaled by 0.5 (preserving the sum). Symmetry-breaking noise (ε) is added to one copy. S...

Thalamus expansion: New relay groups are created with random initialization. Existing relay weights are copied position-by-position. The TRN (thalamic reticular nucleus) operates on d_{model} (not column count), so its weights transfer directly without modification.

Lateral expansion: New columns get zero-initialized send weights (identity-at-init principle: no lateral signal initially). Old column positions are mapped to their new indices in the expanded arrangement.

Workspace expansion: New columns get zero-initialized write projections. Old write/read weights are copied to their new positions.

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

15.3 Operation 2: Net2Net Column Splitting (`split_column`)

Theorem 15.3 (Function-Preserving Column Split)**.** *A trained column k with output weight W_{out}^k is split into two daughter columns (k_a, k_b) such that:*

$$W_{out}^{k_a} = 0.5 \cdot W_{out}^k, \qquad W_{out}^{k_b} = 0.5 \cdot W_{out}^k + \mathcal{N}(0, \epsilon) \tag{87}$$

At the moment of split: $W_{out}^{k_a} + W_{out}^{k_b} \approx W_{out}^k$, so network output is mathematically identical (up to noise ϵ). The scaling applies to four output-facing weight matrices: $W_{L5,out}$, $W_{L5,pred}$, $W_{L6,out}$, and $W_{L2/3,out}$.

After split, competitive Hebbian learning drives the two copies to differentiate: child a refines the parent's original specialty, while child b discovers a complementary sub-specialty.

Proposition 15.1 (Column Split is Analogous to Cell Division)**.** *Biological cell division produces two cells that together maintain the function of the parent. Subsequent gene expression differences drive specialization. In NCT, the $0.5\times$ weight scaling preserves function, and symmetry-breaking noise ($\epsilon = 0.01$) plus competitive Hebbian dynamics drive specialization. The lateral connection between the split pair inherits the parent's weights, maintaining the neighborhood structure.*

15.4 Operation 3: Overload Detection (`find_overloaded_columns`)

Before splitting, we need to identify which column is most overloaded — trying to represent too much with insufficient capacity.

Definition 15.1 (Column Overload Score)**.** *For each column i, the overload score is computed over n evaluation batches:*

$$overload_i = 0.6 \times \underbrace{\frac{\bar{\sigma}_i^2 - \min_j \bar{\sigma}_j^2}{\max_j \bar{\sigma}_j^2 - \min_j \bar{\sigma}_j^2 + 10^{-8}}}_{\text{normalized prediction error}} + 0.4 \times \underbrace{\frac{\bar{g}_i - \min_j \bar{g}_j}{\max_j \bar{g}_j - \min_j \bar{g}_j + 10^{-8}}}_{\text{normalized gate activation}} \tag{88}$$

High σ^2 (uncertain predictions) + high gate activation (all mechanisms maxed out) = overloaded column. The highest-scoring column is the best candidate for splitting.

The overload detection evaluates over $n = 50$ batches, collecting mean σ^2 and mean gate values per column. Results are returned sorted by overload score (highest first).

15.5 Operation 4: μP Hyperparameter Transfer (`extract_mu_p`)

Theorem 15.4 (μP Transfer for Local Learning (Yang & Hu, 2021))**.** *Under Maximal Update Parameterization (μP), hyperparameters found on a small model transfer to larger models without retuning. For NCT, 29 scale-invariant hyperparameters are extracted:*

Learning rates (3): *$\eta_{PC}, \eta_H, \eta_{base}$ — these control the magnitude of local updates and are independent of model width.*

PC dynamics (5): *$K_{\min}, K_{\max}, \tau_{conv}, \sigma_{init}^2, \eta_{PC}$ — the convergence behavior of predictive coding is a per-column dynamic, independent of the number of columns.*

Hebbian parameters (3): *k_{WTA}, S, w — sparsity ratio, settling steps, and window size are dimensionless or integer-valued.*

Gate dynamics (2): *$g_0, gate_LR_mult$ — the sigmoid gate behavior is scale-independent.*

BCM/Meta (2): *τ_{BCM}, τ_{meta} — homeostatic time constants.*

Oscillatory (4): *$P_\theta, P_\alpha, P_\delta, \gamma$ — oscillatory periods in tokens.*

WM (2): *K_{NMDA}, K_{phono} — slot counts.*

Sleep (3): *$I_s, R, \delta_{downscale}$ — wake-sleep timing.*

Structural (3): *$\theta_{growth}, \theta_{prune}, I_{prune}$.*

Regularization (3): *dropout, weight decay, grad norm.*

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

NCT's local learning rules make μP transfer **more reliable** than for transformers: each column's dynamics are independent of the total column count. Plasticity rates found on 12 columns should transfer directly to 64 columns, because the local Hebbian and PC update rules operate identically regardless of how many other columns exist.

Dimension-dependent parameters are scaled proportionally:

$$d_{\text{param}}^{\text{large}} = \max\left(\text{floor}, \left\lfloor d_{\text{param}}^{\text{small}} \times \frac{d_{\text{model}}^{\text{large}}}{d_{\text{model}}^{\text{small}}} \right\rfloor\right) \tag{89}$$

This applies to: $d_{\text{relay}}, d_{\text{inhibitory}}, d_{\text{slot}}, d_{\text{trace}}, d_{\text{sparse}}, d_{\text{cerebellar}}, d_{\text{bg}}, d_{\text{trn}}, n_{\text{assoc_heads}}, n_{\text{relay_groups}}, hpc_memory_size$

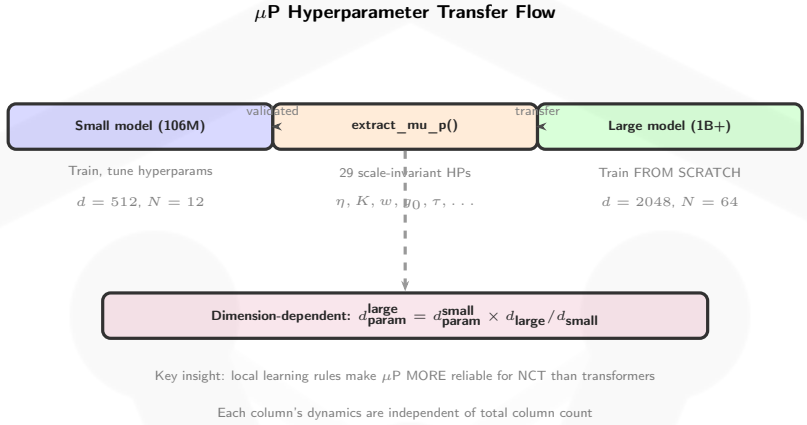


Figure 81: μP hyperparameter transfer: train a small model (106M), extract 29 scale-invariant hyperparameters, transfer them to a large model (1B+) trained from scratch. Dimension-dependent parameters scale proportionally. NCT's local learning makes this transfer more reliable than for transformers.

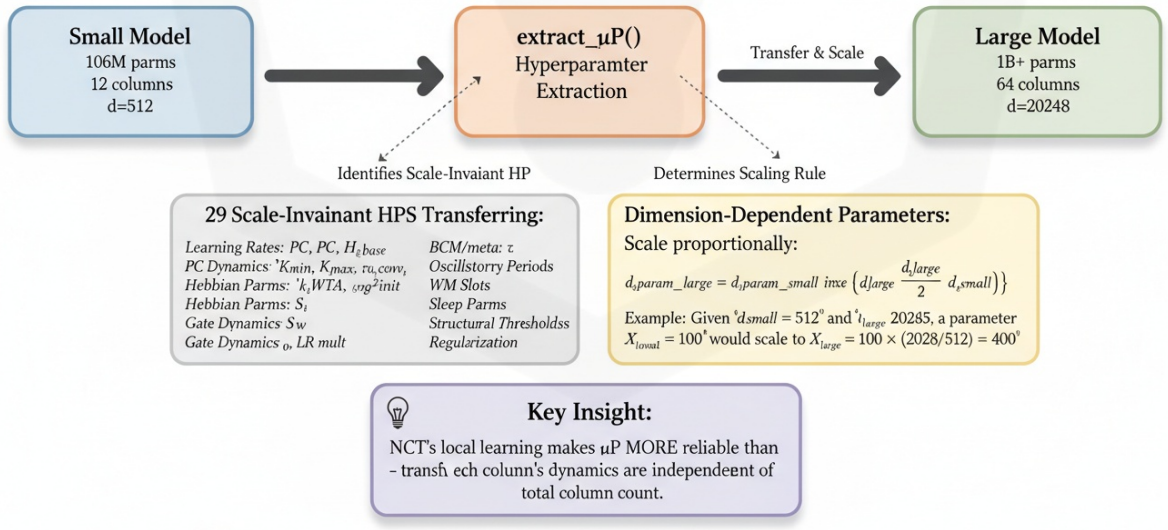


Figure 82: (Illustrated) μP hyperparameter transfer: train a small model (106M), extract 29 scale-invariant hyperparameters, transfer them to a large model (1B+) trained from scratch. Dimension-dependent parameters sca...

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

15.6 Operation 5: Scaling Cost Report (`scaling_report`)

The scaling report provides a per-component parameter breakdown and estimates the cost of each scaling operation:

Table 11: Scaling cost analysis for the default 106M configuration.

Operation	Params Affected	% of Total	Est. Cost vs Retrain
Add 1 column	7.5M (new) + 0.7M (routing)	7.7%	∼8%
Split 1 column	7.5M (divergence) + 0.3M (routing)	7.3%	∼7%
Width ×2 ($d : 512 \to 1024$)	All d-dependent weights	∼75%	∼70%
Full retrain	All 106M	100%	100%

15.7 Recommended Scaling Path

Scaling Strategy Decision Matrix

Small incremental scaling (12 → 16 → 20 columns): Use progressive column growth. Each step costs ∼8% of full retraining. Existing column knowledge is fully preserved.
Targeted capacity increase (one column overloaded): Use column splitting on the highest-overload column. Function-preserving at split point, ∼7% cost.
Large jump ($> 2\times$ scale, e.g., 12 → 64 columns or $d : 512 \to 2048$): Use μP hyperparameter transfer + train from scratch at target scale. The HPs from the small model transfer directly, saving the entire hyperparameter search cost.
Width expansion (d_{model} increase): Most expensive (∼70%). Only use when column growth is insufficient. Old subspace is warm-started but new dimensions need full training.

15.8 Freeze / Unfreeze Utilities

Two helper functions manage parameter freezing during the column bootstrap phase, implementing the biological analog of "critical period protection" — ensuring that established cortical representations are not disrupted while new areas undergo initial self-organization:

`unfreeze_columns(model, column_indices=None)`: Sets `requires_grad = True` for all parameters in specified columns. If `column_indices` is None, unfreezes all columns. Called after new columns complete their prenatal self-organization phase, signaling that they are ready to participate in coordinated learning with existing columns. Logs the number of unfrozen parameters and total trainable count.

`freeze_columns(model, column_indices)`: Sets `requires_grad = False` for all parameters in specified columns. Used during progressive column growth: when new columns are added, existing columns are frozen to prevent their learned Hebbian associations, BCM thresholds, and metaplasticity EMAs from being disturbed by the chaotic, high-amplitude activity generated during prenatal self-organization of the new columns. Once the new columns have stabilized (completed prenatal phase), all columns are unfrozen and competitive Hebbian dynamics re-differentiates the full ensemble.

This freeze/unfreeze protocol is critical for knowledge preservation during scaling: without freezing, the prenatal activity of new columns can propagate through lateral connections and the global workspace, perturbing existing columns' learned representations. With freezing, existing knowledge is locked in place during the bootstrap period.

15.9 Source Code

```
        if pc_losses:
            lines.append(f"\n  PC loss: {pc_losses[0]:.4f} → {pc_losses[-1]:.4f}")
            if pc_losses[0] > 1e-8:
                improvement = (pc_losses[0] - pc_losses[-1]) / pc_losses[0] * 100
                lines.append(f"  PC improvement: {improvement:.1f}%")

        # Sleep phases during prenatal
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        sleep_count = sum(1 for m in self.metrics_history if 'sleep_replayed_memories' in m)
        lines.append(f"\n  Sleep phases: {sleep_count}")

        # Convergence
        lines.append(f"  Converged: {self.has_converged()}")

        lines.append("=" * 60)
        return "\n".join(lines)



# ============================================================================
# Scaling (Column Growth, Splitting, μP Transfer)
# Source: nct_v2/v11_scale.py
# ============================================================================

"""
NCT V11: Scaling -- Progressive Column Growth & Column Splitting
================================================================

Brain-inspired scaling for the cortical column network.

Three scaling operations:
  1. Progressive Column Growth -- add new columns to a trained network
  2. Net2Net Column Splitting -- split overloaded columns into two
  3. μP Hyperparameter Transfer -- extract hyperparameters for training at target scale

The key insight: V11 uses LOCAL learning (PC + Hebbian). ~90% of params are
intra-column weights that are preserved during scaling. Only inter-column
coordination (~10% -- thalamus, lateral, workspace) needs retraining.

Usage:
  # Load trained checkpoint
  model, cfg = load_checkpoint('checkpoints/v11/best.pt')

  # Add 8 new columns (12 → 20)
  model, new_cfg = grow_columns(model, cfg, n_new=8)

  # Or split the most overloaded column
  model, new_cfg = split_column(model, cfg, column_idx=5)

  # Extract μP hyperparameters for training a larger model from scratch
  hp = extract_mu_p(model, cfg)
  large_cfg = V11Config(n_columns=64, d_model=2048, **hp)
"""



# [bundled] from .v11_config import V11Config
# [bundled] from .v11_column import CorticalColumn, LateralColumnConnection, RMSNorm
# [bundled] from .v11_systems import (

logger = logging.getLogger(__name__)


#

# 1. Progressive Column Growth
#


def grow_columns(
    model: nn.Module,
    cfg: V11Config,
    n_new: int,
    insertion: str = 'append',
    freeze_existing: bool = True,
    noise_scale: float = 0.01,
) -> Tuple[nn.Module, V11Config]:
    """
    Add new cortical columns to a trained network.

    Biological analog: postnatal cortical development. New functional areas
    come online alongside existing ones. Visual cortex doesn't retrain when
    language areas develop -- they coordinate through thalamus.

    Args:
        model: trained CorticalColumnNetwork
        cfg: current config
        n_new: number of columns to add
        insertion: 'append' (add to top), 'interleave' (distribute evenly),
                   or 'bottom' (add before existing columns)
        freeze_existing: if True, freeze existing column params during bootstrap
        noise_scale: initialization noise for new columns

    Returns:
        (model, new_cfg) -- model with added columns, updated config

    What's preserved (~90% of params):
        - All existing column internal weights (Hebbian, PC, FFN, WM, etc.)
        - Existing inter-column lateral weights (between OLD columns)
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        - Embedding, LM head, position encoding

    What's retrained (~10% of params):
        - Thalamic routing (expanded for new columns)
        - Lateral connections (new cross-column edges)
        - Workspace (new read/write projections)
        - BG/Cerebellum/HPC interfaces are d_model-based, unchanged
    """
    old_n = cfg.n_columns
    new_n = old_n + n_new

    #        1. Create new config

    new_cfg = copy.deepcopy(cfg)
    new_cfg.n_columns = new_n
    # Adjust relay groups proportionally
    new_cfg.n_relay_groups = max(1, new_n // max(1, old_n // cfg.n_relay_groups))

    logger.info(f"Growing columns: {old_n} → {new_n} "
                f"(relay groups: {cfg.n_relay_groups} → {new_cfg.n_relay_groups})")

    #        2. Create new columns

    new_columns = []
    for i in range(n_new):
        col_idx = old_n + i  # default for 'append'
        col = CorticalColumn(new_cfg, column_idx=col_idx)

        # Add small noise to break symmetry between new columns
        with torch.no_grad():
            for p in col.parameters():
                if p.requires_grad:
                    p.add_(torch.randn_like(p) * noise_scale)

        new_columns.append(col)

    #        3. Insert columns at specified positions
    old_columns = list(model.columns)

    if insertion == 'append':
        all_columns = old_columns + new_columns
    elif insertion == 'bottom':
        all_columns = new_columns + old_columns
    elif insertion == 'interleave':
        # Distribute new columns evenly among existing ones
        all_columns = []
        step = max(1, old_n // n_new)
        new_iter = iter(new_columns)
        for i, col in enumerate(old_columns):
            all_columns.append(col)
            if (i + 1) % step == 0:
                try:
                    all_columns.append(next(new_iter))
                except StopIteration:
                    pass
        # Append any remaining new columns
        for remaining in new_iter:
            all_columns.append(remaining)
    else:
        raise ValueError(f"Unknown insertion mode: {insertion}")

    # Reassign column indices
    for i, col in enumerate(all_columns):
        col.column_idx = i

    model.columns = nn.ModuleList(all_columns)

    #        4. Expand thalamic routing

    model.thalamus = _expand_thalamus(model.thalamus, cfg, new_cfg)

    #        5. Expand lateral connections

    model.lateral = _expand_lateral(model.lateral, cfg, new_cfg, old_columns, all_columns)

    #        6. Expand global workspace

    model.workspace = _expand_workspace(model.workspace, cfg, new_cfg)

    #        7. Expand self-assessment

    model.self_assessment = SelfAssessment(new_cfg)

    #        8. Update config reference

    model.cfg = new_cfg

    #        9. Freeze existing columns if requested
    if freeze_existing:
        frozen_count = 0
        for col in old_columns:
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
            for p in col.parameters():
                p.requires_grad = False
                frozen_count += 1
        logger.info(f"Froze {frozen_count} params in {len(old_columns)} existing columns. "
                    f"Call unfreeze_columns(model) after bootstrap.")

    #        10. Report parameter changes

    total_params = sum(p.numel() for p in model.parameters())
    trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
    logger.info(f"After growth: {total_params:,} total params, "
                f"{trainable_params:,} trainable ({trainable_params/total_params*100:.1f}%)")

    return model, new_cfg


def _expand_thalamus(old_thal: Thalamus, old_cfg: V11Config, new_cfg: V11Config) -> Thalamus:
    """Expand thalamus to accommodate more columns.

    Strategy: create new thalamus with more relay groups,
    copy existing relay weights into corresponding positions.
    TRN is dimension-based (d_model), so its weights transfer directly.
    """
    new_thal = Thalamus(new_cfg)

    if not old_thal.enabled or not new_thal.enabled:
        return new_thal

    # Copy existing relay group weights
    n_copy = min(old_cfg.n_relay_groups, new_cfg.n_relay_groups)
    with torch.no_grad():
        for i in range(n_copy):
            # Copy relay weights (W_in, W_out, gate, norm)
            new_thal.relays[i].load_state_dict(old_thal.relays[i].state_dict())

        # Copy TRN weights (operates on d_model, dimension unchanged)
        new_thal.trn.load_state_dict(old_thal.trn.state_dict())

    logger.info(f"Thalamus expanded: {old_cfg.n_relay_groups} → {new_cfg.n_relay_groups} relay groups "
                f"({n_copy} preserved)")
    return new_thal


def _expand_lateral(
    old_lat: LateralColumnConnection,
    old_cfg: V11Config,
    new_cfg: V11Config,
    old_columns: list,
    all_columns: list,
) -> LateralColumnConnection:
    """Expand lateral connections for new column count.

    Strategy: create new lateral system, copy weights for old column positions.
    New columns get zero-initialized send weights (identity-at-init principle).
    """
    new_lat = LateralColumnConnection(new_cfg)

    # Map old column positions in the new arrangement
    old_positions = []
    for i, col in enumerate(all_columns):
        if col in old_columns:
            old_positions.append(i)

    # Copy old lateral weights to their new positions
    with torch.no_grad():
        for old_idx, new_idx in enumerate(old_positions):
            if old_idx < len(old_lat.W_send) and new_idx < len(new_lat.W_send):
                new_lat.W_send[new_idx].load_state_dict(old_lat.W_send[old_idx].state_dict())
                new_lat.W_recv[new_idx].load_state_dict(old_lat.W_recv[old_idx].state_dict())
                new_lat.gate_raws[new_idx].data.copy_(old_lat.gate_raws[old_idx].data)

    logger.info(f"Lateral connections expanded: {old_cfg.n_columns} → {new_cfg.n_columns} "
                f"({len(old_positions)} preserved)")
    return new_lat


def _expand_workspace(
    old_ws: CorticalWorkspace,
    old_cfg: V11Config,
    new_cfg: V11Config,
) -> CorticalWorkspace:
    """Expand global workspace for new column count.

    Strategy: create new workspace, copy old read/write projections.
    New columns get zero-initialized write projections (identity-at-init).
    """
    new_ws = CorticalWorkspace(new_cfg)

    n_copy = min(old_cfg.n_columns, new_cfg.n_columns)
    with torch.no_grad():
        for i in range(n_copy):
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
            new_ws.W_write[i].load_state_dict(old_ws.W_write[i].state_dict())
            new_ws.W_read[i].load_state_dict(old_ws.W_read[i].state_dict())

    logger.info(f"Workspace expanded: {old_cfg.n_columns} → {new_cfg.n_columns} columns")
    return new_ws


#

# 2. Net2Net Column Splitting
#


def split_column(
    model: nn.Module,
    cfg: V11Config,
    column_idx: int,
    noise_scale: float = 0.01,
) -> Tuple[nn.Module, V11Config]:
    """
    Split a trained column into two -- function-preserving at moment of split.

    Biological analog: cell division. The two daughter columns together compute
    the exact same function as the parent. Then competitive Hebbian learning
    drives them to differentiate (one keeps the original specialty, the other
    discovers a sub-specialty).

    Math: if Column_k has output weight W_out, we create:
        Column_k_a with W_out_a = 0.5 * W_out + noise
        Column_k_b with W_out_b = 0.5 * W_out + noise
    So: W_out_a + W_out_b ~ W_out (function preserved)

    Args:
        model: trained CorticalColumnNetwork
        cfg: current config
        column_idx: which column to split (0-indexed)
        noise_scale: symmetry-breaking noise magnitude

    Returns:
        (model, new_cfg) -- model with split column, updated config
    """
    old_n = cfg.n_columns
    new_n = old_n + 1

    if column_idx < 0 or column_idx >= old_n:
        raise ValueError(f"column_idx {column_idx} out of range [0, {old_n})")

    #         1. Deep-copy the column

    parent = model.columns[column_idx]
    child_a = copy.deepcopy(parent)
    child_b = copy.deepcopy(parent)

    #         2. Scale output weights by 0.5 (function preservation)
    # The key outputs are: L5 output (W_out), L6 feedback, prediction
    with torch.no_grad():
        _scale_output_weights(child_a, 0.5)
        _scale_output_weights(child_b, 0.5)

        # Add symmetry-breaking noise to child_b
        for pa, pb in zip(child_a.parameters(), child_b.parameters()):
            pb.add_(torch.randn_like(pb) * noise_scale)

    #         3. Insert child_b after child_a

    columns = list(model.columns)
    columns[column_idx] = child_a
    columns.insert(column_idx + 1, child_b)

    # Reassign indices
    for i, col in enumerate(columns):
        col.column_idx = i

    model.columns = nn.ModuleList(columns)

    #         4. Update config and subsystems

    new_cfg = copy.deepcopy(cfg)
    new_cfg.n_columns = new_n
    new_cfg.n_relay_groups = max(1, new_n // max(1, old_n // cfg.n_relay_groups))

    model.thalamus = _expand_thalamus(model.thalamus, cfg, new_cfg)
    model.lateral = _expand_lateral_for_split(model.lateral, cfg, new_cfg, column_idx)
    model.workspace = _expand_workspace(model.workspace, cfg, new_cfg)
    model.self_assessment = SelfAssessment(new_cfg)
    model.cfg = new_cfg

    total_params = sum(p.numel() for p in model.parameters())
    logger.info(f"Split column {column_idx}: {old_n} → {new_n} columns, "
                f"{total_params:,} total params. Function preserved at split point.")

```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
    return model, new_cfg


def _scale_output_weights(column: CorticalColumn, scale: float):
    """Scale a column's output-facing weights to preserve function after split."""
    # L5 output (the main output pathway)
    if hasattr(column.L5, 'W_out'):
        column.L5.W_out.weight.mul_(scale)
    # L5 prediction head
    if hasattr(column.L5, 'W_pred'):
        column.L5.W_pred.weight.mul_(scale)
    # L6 thalamic feedback
    if hasattr(column.L6, 'W_out'):
        column.L6.W_out.weight.mul_(scale)
    # L2/3 output projection
    if hasattr(column.L23, 'W_out'):
        column.L23.W_out.weight.mul_(scale)


def _expand_lateral_for_split(
    old_lat: LateralColumnConnection,
    old_cfg: V11Config,
    new_cfg: V11Config,
    split_idx: int,
) -> LateralColumnConnection:
    """Expand lateral connections after a column split.

    The new column (at split_idx+1) gets a copy of the split column's
    lateral weights, maintaining the neighborhood structure.
    """
    new_lat = LateralColumnConnection(new_cfg)

    with torch.no_grad():
        for old_i in range(old_cfg.n_columns):
            # Map old index to new index (shift everything after split_idx by 1)
            new_i = old_i if old_i <= split_idx else old_i + 1

            if old_i < len(old_lat.W_send) and new_i < len(new_lat.W_send):
                new_lat.W_send[new_i].load_state_dict(old_lat.W_send[old_i].state_dict())
                new_lat.W_recv[new_i].load_state_dict(old_lat.W_recv[old_i].state_dict())
                new_lat.gate_raws[new_i].data.copy_(old_lat.gate_raws[old_i].data)

        # The new column (split_idx+1) gets a copy of the parent's lateral weights
        parent_i = split_idx
        child_i = split_idx + 1
        if parent_i < len(old_lat.W_send) and child_i < len(new_lat.W_send):
            new_lat.W_send[child_i].load_state_dict(old_lat.W_send[parent_i].state_dict())
            new_lat.W_recv[child_i].load_state_dict(old_lat.W_recv[parent_i].state_dict())
            new_lat.gate_raws[child_i].data.copy_(old_lat.gate_raws[parent_i].data)

    return new_lat


#

# 3. Identify Columns to Split (Overload Detection)
#


def find_overloaded_columns(
    model: nn.Module,
    dataloader,
    n_batches: int = 50,
    device: str = 'cuda',
) -> List[Dict]:
    """
    Identify which columns are overloaded and would benefit from splitting.

    Metrics for overload:
      1. High mean prediction error ( ² relative to others)
      2. Low sparsity (k-WTA near saturation -- trying to represent too much)
      3. High gate activation (all mechanisms maxed out)
      4. Low column differentiation (not specialized enough)

    Returns:
        List of dicts sorted by overload score (highest first):
        [{'idx': 5, 'score': 0.87, 'pe': 2.1, 'sparsity': 0.45, 'gate_mean': 0.9}, ...]
    """
    model.eval()
    n_cols = model.cfg.n_columns

    # Accumulators
    pe_accum = torch.zeros(n_cols)
    sparsity_accum = torch.zeros(n_cols)
    gate_accum = torch.zeros(n_cols)
    count = 0

    with torch.no_grad():
        for batch_idx, batch in enumerate(dataloader):
            if batch_idx >= n_batches:
                break
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
            input_ids = batch['input_ids'].to(device) if isinstance(batch, dict) else batch[0].to(device)
            targets = batch.get('labels', None) if isinstance(batch, dict) else (batch[1].to(device) if len(batch) > 1
                 else None)

            result = model(input_ids, targets=targets)

            # Collect per-column metrics from the last forward pass
            if model._last_column_outputs is not None:
                for i, out in enumerate(model._last_column_outputs):
                    # Prediction error magnitude
                    pe_accum[i] += out.sigma2.mean().cpu()
                    # Gate values (proxy for mechanism utilization)
                    gates = model.columns[i].get_gate_values()
                    gate_mean = sum(gates.values()) / max(len(gates), 1)
                    gate_accum[i] += gate_mean

            count += 1

    if count == 0:
        logger.warning("No batches processed for overload detection")
        return []

    pe_accum /= count
    gate_accum /= count

    # Normalize to [0, 1] range
    pe_norm = (pe_accum - pe_accum.min()) / (pe_accum.max() - pe_accum.min() + 1e-8)
    gate_norm = (gate_accum - gate_accum.min()) / (gate_accum.max() - gate_accum.min() + 1e-8)

    # Overload score: high PE + high gate activation = overloaded
    overload_scores = 0.6 * pe_norm + 0.4 * gate_norm

    results = []
    for i in range(n_cols):
        results.append({
            'idx': i,
            'score': overload_scores[i].item(),
            'pe': pe_accum[i].item(),
            'gate_mean': gate_accum[i].item(),
        })

    results.sort(key=lambda x: x['score'], reverse=True)
    return results


#

# 4. μP Hyperparameter Transfer
#


def extract_mu_p(model: nn.Module, cfg: V11Config) -> Dict:
    """
    Extract μP-transferable hyperparameters from a trained small model.

    These hyperparameters are scale-invariant under Maximal Update
    Parameterization (Yang & Hu 2021) -- they transfer from small to large
    models without retuning.

    For V11, local learning rules make μP transfer MORE reliable than for
    transformers: each column's dynamics are independent of the total
    column count, so plasticity rates found on 12 columns transfer to 64.

    Returns:
        Dict of hyperparameters suitable for constructing a larger V11Config
    """
    hp = {}

    #        Learning rates (scale-invariant in μP)

    hp['hebbian_lr'] = cfg.hebbian_lr
    hp['pc_learning_lr'] = cfg.pc_learning_lr
    hp['base_lr'] = cfg.base_lr

    #        PC dynamics

    hp['pc_min_steps'] = cfg.pc_min_steps
    hp['pc_max_steps'] = cfg.pc_max_steps
    hp['pc_convergence_threshold'] = cfg.pc_convergence_threshold
    hp['sigma2_init'] = cfg.sigma2_init

    #        Hebbian parameters

    hp['k_wta_ratio'] = cfg.k_wta_ratio
    hp['n_settle_steps'] = cfg.n_settle_steps
    hp['assoc_window_size'] = cfg.assoc_window_size

    #        Neuromodulator scales (learned)

    # Extract the effective neuromod scales from the trained brainstem
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
    if hasattr(model, 'brainstem'):
        bs = model.brainstem
        if hasattr(bs, 'da_scale'):
            hp['_da_scale_trained'] = bs.da_scale.item() if isinstance(bs.da_scale, torch.Tensor) else bs.da_scale
        if hasattr(bs, 'ne_scale'):
            hp['_ne_scale_trained'] = bs.ne_scale.item() if isinstance(bs.ne_scale, torch.Tensor) else bs.ne_scale

    #         Gate dynamics
    hp['gate_init'] = cfg.gate_init
    hp['gate_lr_mult'] = cfg.gate_lr_mult

    #         BCM / Metaplasticity
    hp['bcm_tau'] = cfg.bcm_tau

    #         Oscillatory parameters
    hp['gamma_period'] = cfg.gamma_period
    hp['theta_period'] = cfg.theta_period
    hp['alpha_period'] = cfg.alpha_period
    hp['delta_period'] = cfg.delta_period

    #         WM parameters
    hp['nmda_slots'] = cfg.nmda_slots
    hp['phono_capacity'] = cfg.phono_capacity

    #         Wake-sleep
    hp['sleep_interval'] = cfg.sleep_interval
    hp['sleep_replay_steps'] = cfg.sleep_replay_steps
    hp['synaptic_downscale'] = cfg.synaptic_downscale

    #         Structural plasticity
    hp['growth_threshold'] = cfg.growth_threshold
    hp['prune_threshold'] = cfg.prune_threshold
    hp['prune_interval'] = cfg.prune_interval

    #         Regularization
    hp['dropout'] = cfg.dropout
    hp['weight_decay'] = cfg.weight_decay
    hp['max_grad_norm'] = cfg.max_grad_norm

    #         Effective gate values (diagnostic, not directly transferred)
    hp['_gate_values'] = {}
    for i, col in enumerate(model.columns):
        hp['_gate_values'][f'col_{i}'] = col.get_gate_values()

    logger.info(f"Extracted {len(hp)} μP hyperparameters from {cfg.n_columns}-column model. "
                f"Transfer to any scale model.")

    return hp


def build_scaled_config(
    base_cfg: V11Config,
    mu_p: Dict,
    target_columns: int,
    target_d_model: int,
    target_d_ff: Optional[int] = None,
) -> V11Config:
    """
    Build a scaled-up V11Config using μP hyperparameters from a small model.

    Args:
        base_cfg: the small model's config (for structural reference)
        mu_p: hyperparameters from extract_mu_p()
        target_columns: number of columns in the large model
        target_d_model: d_model in the large model
        target_d_ff: d_ff in the large model (default: 4 * d_model)

    Returns:
        V11Config for the large model with transferred hyperparameters
    """
    if target_d_ff is None:
        target_d_ff = 4 * target_d_model

    # Scale dimension-dependent params proportionally
    d_ratio = target_d_model / base_cfg.d_model

    scaled = V11Config(
        #         Target dimensions
        d_model=target_d_model,
        d_column=target_d_model,
        n_columns=target_columns,
        d_ff=target_d_ff,
        vocab_size=base_cfg.vocab_size,
        max_seq_len=base_cfg.max_seq_len,
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        #        Scaled proportionally to d_model
        n_assoc_heads=max(1, int(base_cfg.n_assoc_heads * d_ratio)),
        d_inhibitory=max(32, int(base_cfg.d_inhibitory * d_ratio)),
        nmda_d_slot=max(32, int(base_cfg.nmda_d_slot * d_ratio)),
        silent_d_trace=max(32, int(base_cfg.silent_d_trace * d_ratio)),
        phono_d_slot=max(32, int(base_cfg.phono_d_slot * d_ratio)),
        d_relay=max(64, int(base_cfg.d_relay * d_ratio)),
        d_trn=max(16, int(base_cfg.d_trn * d_ratio)),
        d_cerebellar=max(64, int(base_cfg.d_cerebellar * d_ratio)),
        d_bg=max(32, int(base_cfg.d_bg * d_ratio)),
        n_relay_groups=max(1, target_columns // max(1, base_cfg.n_columns // base_cfg.n_relay_groups)),
        hpc_d_sparse=max(512, int(base_cfg.hpc_d_sparse * d_ratio)),
        hpc_memory_size=int(base_cfg.hpc_memory_size * max(1.0, d_ratio)),

        #        μP transferred (scale-invariant)
        hebbian_lr=mu_p.get('hebbian_lr', base_cfg.hebbian_lr),
        pc_learning_lr=mu_p.get('pc_learning_lr', base_cfg.pc_learning_lr),
        base_lr=mu_p.get('base_lr', base_cfg.base_lr),
        pc_min_steps=mu_p.get('pc_min_steps', base_cfg.pc_min_steps),
        pc_max_steps=mu_p.get('pc_max_steps', base_cfg.pc_max_steps),
        pc_convergence_threshold=mu_p.get('pc_convergence_threshold', base_cfg.pc_convergence_threshold),
        sigma2_init=mu_p.get('sigma2_init', base_cfg.sigma2_init),
        k_wta_ratio=mu_p.get('k_wta_ratio', base_cfg.k_wta_ratio),
        n_settle_steps=mu_p.get('n_settle_steps', base_cfg.n_settle_steps),
        assoc_window_size=mu_p.get('assoc_window_size', base_cfg.assoc_window_size),
        gate_init=mu_p.get('gate_init', base_cfg.gate_init),
        gate_lr_mult=mu_p.get('gate_lr_mult', base_cfg.gate_lr_mult),
        bcm_tau=mu_p.get('bcm_tau', base_cfg.bcm_tau),
        gamma_period=mu_p.get('gamma_period', base_cfg.gamma_period),
        theta_period=mu_p.get('theta_period', base_cfg.theta_period),
        alpha_period=mu_p.get('alpha_period', base_cfg.alpha_period),
        delta_period=mu_p.get('delta_period', base_cfg.delta_period),
        nmda_slots=mu_p.get('nmda_slots', base_cfg.nmda_slots),
        phono_capacity=mu_p.get('phono_capacity', base_cfg.phono_capacity),
        sleep_interval=mu_p.get('sleep_interval', base_cfg.sleep_interval),
        sleep_replay_steps=mu_p.get('sleep_replay_steps', base_cfg.sleep_replay_steps),
        synaptic_downscale=mu_p.get('synaptic_downscale', base_cfg.synaptic_downscale),
        growth_threshold=mu_p.get('growth_threshold', base_cfg.growth_threshold),
        prune_threshold=mu_p.get('prune_threshold', base_cfg.prune_threshold),
        prune_interval=mu_p.get('prune_interval', base_cfg.prune_interval),
        dropout=mu_p.get('dropout', base_cfg.dropout),
        weight_decay=mu_p.get('weight_decay', base_cfg.weight_decay),
        max_grad_norm=mu_p.get('max_grad_norm', base_cfg.max_grad_norm),

        #        Structural settings (unchanged)
        learning_mode=base_cfg.learning_mode,
        error_hebbian_weight=base_cfg.error_hebbian_weight,
        ntp_output_only=base_cfg.ntp_output_only,
        autonomous_training=base_cfg.autonomous_training,
    )

    total_approx = (
        scaled.vocab_size * scaled.d_model  # embedding
        + scaled.n_columns * (
            scaled.d_model * scaled.d_column  # L4
            + scaled.d_column * scaled.d_column  # L2/3
            + scaled.d_column * scaled.d_ff * 2  # L5 FFN
            + scaled.d_column * scaled.d_model  # L5 output
            + scaled.d_column * scaled.d_model  # L6
        )
        + scaled.d_model * scaled.vocab_size  # LM head
    )
    logger.info(f"Built scaled config: {scaled.n_columns} columns, d={scaled.d_model}, "
                f"~{total_approx / 1e6:.0f}M params (approximate)")

    return scaled


#

# 5. Freeze / Unfreeze Utilities
#


def unfreeze_columns(model: nn.Module, column_indices: Optional[List[int]] = None):
    """Unfreeze column parameters after bootstrap phase.

    Args:
        model: CorticalColumnNetwork
        column_indices: specific columns to unfreeze, or None for all
    """
    unfrozen = 0
    for i, col in enumerate(model.columns):
        if column_indices is None or i in column_indices:
            for p in col.parameters():
                p.requires_grad = True
                unfrozen += 1

    total_trainable = sum(p.numel() for p in model.parameters() if p.requires_grad)
    logger.info(f"Unfroze {unfrozen} params. Total trainable: {total_trainable:,}")
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```python


def freeze_columns(model: nn.Module, column_indices: List[int]):
    """Freeze specific columns (e.g., during new column bootstrap)."""
    frozen = 0
    for i, col in enumerate(model.columns):
        if i in column_indices:
            for p in col.parameters():
                p.requires_grad = False
                frozen += 1

    logger.info(f"Froze {frozen} params in columns {column_indices}")


#

# 6. Scaling Report
#


def scaling_report(model: nn.Module, cfg: V11Config) -> Dict:
    """
    Generate a scaling analysis report for the current model.

    Reports:
      - Parameter distribution (local vs global)
      - Per-column parameter counts
      - Estimated cost of each scaling operation
      - Recommended scaling path
    """
    counts = model.count_parameters()

    # Per-column params
    col_params = []
    for i, col in enumerate(model.columns):
        n = sum(p.numel() for p in col.parameters())
        col_params.append(n)

    # Local vs global breakdown
    local_params = counts.get('columns', 0)
    global_params = (
        counts.get('thalamus', 0)
        + counts.get('lateral', 0)
        + counts.get('workspace', 0)
        + counts.get('self_assessment', 0)
    )
    fixed_params = (
        counts.get('embedding', 0)
        + counts.get('lm_head', 0)
        + counts.get('grid_pos', 0)
    )
    subcortical_params = (
        counts.get('hippocampus', 0)
        + counts.get('basal_ganglia', 0)
        + counts.get('cerebellum', 0)
        + counts.get('brainstem', 0)
    )

    total = counts.get('total', 1)

    report = {
        'total_params': total,
        'local_column_params': local_params,
        'local_pct': local_params / total * 100,
        'global_coordination_params': global_params,
        'global_pct': global_params / total * 100,
        'fixed_params': fixed_params,
        'fixed_pct': fixed_params / total * 100,
        'subcortical_params': subcortical_params,
        'subcortical_pct': subcortical_params / total * 100,
        'per_column_params': col_params,
        'mean_column_params': sum(col_params) / max(len(col_params), 1),
        'n_columns': cfg.n_columns,
        'd_model': cfg.d_model,
    }

    # Estimated costs
    one_col_params = report['mean_column_params']
    report['cost_add_1_column'] = one_col_params  # New column params
    report['cost_add_1_routing'] = global_params * 0.2  # ~20% of routing retrains
    report['cost_split_1_column'] = one_col_params + global_params * 0.1
    report['cost_full_retrain'] = total

    # Percentage costs
    report['pct_cost_grow_1'] = (one_col_params + global_params * 0.2) / total * 100
    report['pct_cost_split_1'] = (one_col_params + global_params * 0.1) / total * 100

    return report


def print_scaling_report(model: nn.Module, cfg: V11Config):
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
    """Pretty-print the scaling report."""
    r = scaling_report(model, cfg)
```

Listing 14: Scaling Framework — complete source code (v11_scale.py, 791 lines)

16 Production Interface

Source: `nct_v2/v11_production.py` **Lines:** 567 **Functions:** 8 **Classes:** 3

16.1 Overview

This module bridges NCT's brain-inspired architecture with real-world deployment infrastructure, providing five adapter systems that leverage NCT's intrinsic brain signals for production-level decisions. The key insight: transformers require external systems for quantization sensitivity analysis, tool use routing, retrieval-augmented generation triggers, and compute allocation. NCT's biological signals — prediction precision (σ^2), dopaminergic exploration/exploitation (DA, tonic and phasic), ACC conflict monitoring, and hippocampal surprise tagging — provide these capabilities **natively** as byproducts of the forward pass.

The five adapter systems are: (1) σ^2-guided quantization precision mapping, (2) FSDP2 distributed training with column-level sharding, (3) SFT/DPO alignment with σ^2-adaptive loss weighting, (4) agentic tool use and retrieval triggered by brain signals, and (5) σ^2-early-exit for adaptive column depth (variable compute per token). Each adapter extracts signals already computed during the standard forward pass — no additional modules or forward passes are needed.

16.2 Quantization (σ^2-Guided Precision Mapping)

Theorem 16.1 (Brain-Signal-Guided Quantization)**.** *NCT parameters are mapped to precision levels based on their **computational sensitivity** — how much quantization noise each parameter can tolerate without degrading the brain mechanisms:*

$$precision(\theta) = \begin{cases} FP16 & \text{if } \theta \in \{W_\sigma, \text{decay rates}, \text{osc phases}, \text{grid-cell } \omega\} \\ INT8 & \text{if } \theta \in \{W_{L4}, W_{L5}, W_{L6}, W_{gate}, W_{intern}, W_{embed}, W_{LM}\} \\ INT4 & \text{if } \theta \in \{W_{assoc}, W_{FFN}\} \text{ (largest blocks, highest redundancy)} \end{cases} \tag{90}$$

Rationale for FP16 on σ^2: Precision weights directly control the PC learning signal ($\Delta\theta \propto 1/\sigma^2$). Quantization noise in σ^2 propagates multiplicatively into weight updates, making this the most quantization-sensitive parameter. Decay rates ($\sigma(\beta)$) compound exponentially — small errors grow over hundreds of tokens. Oscillatory phases require sub-radian precision for correct PAC coupling.

Rationale for INT4 on W_{assoc}: The L2/3 association weights are the largest single parameter block ($H \times d_h \times d_h$ per column $\times N$ columns). They are updated by Hebbian learning (which is inherently noisy), making them tolerant of quantization noise. Similarly, L5 FFN weights ($d \times d_{\text{ff}} \times 2$) have high redundancy.

16.3 Distributed Training (FSDP2)

Definition 16.1 (Column-Level FSDP Sharding)**.** *Each `CorticalColumn` is one FSDP unit. This ensures that all stateful components within a column (L2/3 Hebbian weights, working memory state, oscillatory phases, BCM threshold, metaplasticity EMA) reside on the same device and are never split across shards. The wrapping policy:*

1. ***Inner shard (per column):** All 6 layers + interneurons + WM + oscillations + BCM + metaplasticity. Each column is `fully_shard()`'ed independently.*
2. ***Outer shard (model level):** Embedding, LM head, thalamus, hippocampus, basal ganglia, cerebellum, brainstem, lateral connections, workspace, self-assessment. These are smaller and can be split across devices.*

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

Mixed precision policy: `param_dtype=bfloat16`, `reduce_dtype=float32` *(accumulate gradients in full precision to prevent Hebbian update drift).*

16.4 Alignment: SFT with σ^2-Aware Weighting

Theorem 16.2 (σ^2-Aware Supervised Fine-Tuning)**.** *Standard SFT applies uniform loss weight to all tokens. NCT's SFT reduces loss weight on tokens where the model is genuinely uncertain:*

$$\mathcal{L}_{SFT} = \frac{\sum_{t=1}^{T} \ell(y_t, \hat{y}_t) \cdot m_t \cdot w_{\sigma^2}}{\sum_{t=1}^{T} m_t}, \qquad w_{\sigma^2} = \frac{1}{1 + 0.1 \cdot \max(0, \bar{\sigma}^2 - 2.0)} \tag{91}$$

where m_t is the response mask (1 for response tokens, 0 for prompt tokens) and $\bar{\sigma}^2$ is the mean precision across columns. When $\bar{\sigma}^2 > 2.0$ (uncertain), the weight decreases — the model is not penalized for being uncertain about genuinely ambiguous content. Auxiliary losses (PC + cerebellar) are maintained at 10% weight to preserve brain mechanisms during fine-tuning.

16.5 Alignment: DPO with σ^2-Adaptive β

Theorem 16.3 (σ^2-Adaptive Direct Preference Optimization)**.** *Standard DPO uses a fixed temperature β. NCT scales β inversely with model uncertainty:*

$$\beta_{eff} = \frac{\beta}{\max(\bar{\sigma}^2, 0.1)} \tag{92}$$

$$\mathcal{L}_{DPO} = -\log \sigma\Big(\beta_{eff} \cdot \Big(\log \frac{\pi_\theta(y_w|x)}{\pi_{ref}(y_w|x)} - \log \frac{\pi_\theta(y_l|x)}{\pi_{ref}(y_l|x)}\Big)\Big) \tag{93}$$

High certainty *(low σ^2): β_{eff} is large → strong preference enforcement. The model is confident, so push hard toward the preferred output.*

High uncertainty *(high σ^2): β_{eff} is small → soft preference enforcement. The model is uncertain, so don't force strong opinions on ambiguous content.*

This is biologically motivated: the brain doesn't form strong preferences about things it doesn't understand. Premature certainty is a pathology, not a feature.

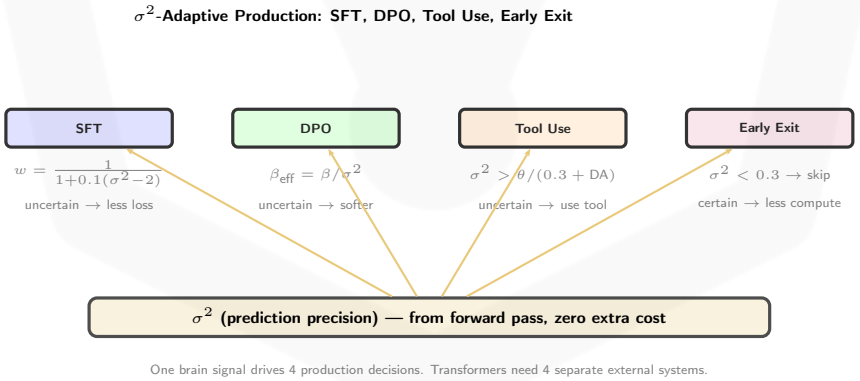


Figure 83: σ^2-adaptive production: a single brain signal (prediction precision) drives four deployment decisions. SFT reduces loss weight on uncertain tokens. DPO softens preferences on ambiguous content. Tool use activates when uncertain. Early exit skips columns when confident. All from one forward pass — zero additional compute.

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

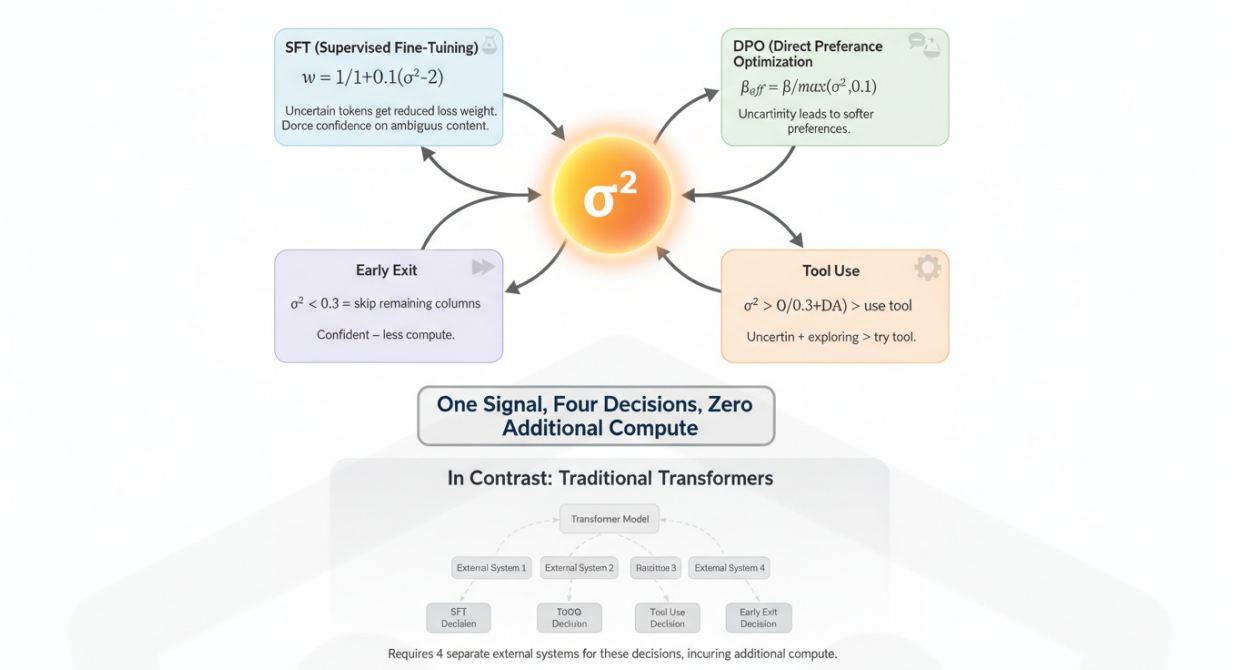


Figure 84: (Illustrated) σ^2-adaptive production: a single brain signal (prediction precision) drives four deployment decisions. SFT reduces loss weight on uncertain tokens. DPO softens preferences on ambiguous cont...

16.6 Tool Use (`V11ToolAdapter`)

NCT's biological circuits provide native signals for agentic tool use:

$$\text{should_use_tool(result)} = \left(\sigma^2 > \frac{\theta_{\text{tool}}}{0.3 + \text{tonic_DA}}\right) \tag{94}$$

High σ^2 + high tonic DA: Uncertain + exploring → try a tool (the BG "Go" pathway is activated). **Low σ^2:** Confident in text generation → don't interrupt with tool use. DA modulates the threshold: high DA lowers the threshold (more impulsive tool use, analogous to dopaminergic exploration). Tool confidence is computed from ACC conflict: high conflict = low confidence = maybe ask the user.

16.7 Retrieval (`V11RetrievalAdapter`)

The retrieval adapter implements σ^2-gated retrieval-augmented generation (RAG). Unlike standard RAG systems that retrieve on every query (wasting latency and tokens), NCT retrieves only when the model is genuinely uncertain:

$$\text{should_retrieve(result)} = (\bar{\sigma}^2 > \theta_{\text{retrieval}}), \qquad \theta_{\text{retrieval}} = 2.0 \tag{95}$$

This is biologically motivated: the hippocampus supplements cortical knowledge specifically when the cortex cannot make a confident prediction on its own. A student who knows the answer doesn't consult the textbook; one who is uncertain does. NCT replicates this behavior automatically via σ^2.

The retrieval query is extracted from the last column's output ($\mathbf{h}_{N-1}[:, -1, :] \in \mathbb{R}^d$) — the most abstract, goal-relevant representation in the hierarchy. This query can be passed to any standard retrieval system (vector database, BM25, or hybrid) to fetch relevant context, which is then prepended to the input for the next forward pass.

The adapter also provides `get_retrieval_query(model, input_ids)` which runs a forward pass and extracts the query embedding, and `should_retrieve(result, sigma2_threshold)` which inspects a forward result's σ^2 without requiring an additional forward pass.

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

16.8 σ^2-Early Exit (Adaptive Column Depth)

Theorem 16.4 (σ^2-Guided Early Exit)**.** *The brain doesn't process every input to the same depth. Easy stimuli are resolved in early sensory cortices; hard stimuli propagate to prefrontal regions. NCT implements this:*

$$\textit{exit at column } i \quad \textit{if} \quad i \geq N_{\min} = 2 \quad \textit{and} \quad \bar{\sigma}_i^2 < \theta_{exit} = 0.3 \tag{96}$$

*When σ^2 drops below threshold (the model is confident), remaining columns are skipped. The forward pass jumps directly to subcortical processing and the LM head. This saves compute proportional to $N - i$ skipped columns, with **no quality loss** on easy tokens (they don't need deep processing).*

For the default 12-column model, if the model exits at column 4 (confident after 4 columns), it saves $8/12 = 67\%$ of column compute on that token. The average savings depend on the input difficulty distribution.

16.9 Production Setup

A convenience function `setup_v11_production(model)` attaches all adapters:

```
model.tools = V11ToolAdapter(model)
model.retrieval = V11RetrievalAdapter(model)
```

After calling this, the model's own brain signals drive tool use and retrieval decisions without any external routing logic.

16.10 Source Code

```
    print(f"\n{'=' * 60}")
    print(f"  V11 Scaling Report -- {r['n_columns']} columns, d={r['d_model']}")
    print(f"{'=' * 60}")
    print(f"  Total params:          {r['total_params']:>12,}")
    print(f"  Local (columns):       {r['local_column_params']:>12,}  ({r['local_pct']:.1f}%)")
    print(f"  Global (routing):      {r['global_coordination_params']:>12,}  ({r['global_pct']:.1f}%)")
    print(f"  Fixed (embed/head):    {r['fixed_params']:>12,}  ({r['fixed_pct']:.1f}%)")
    print(f"  Subcortical:           {r['subcortical_params']:>12,}  ({r['subcortical_pct']:.1f}%)")
    print(f"  Mean params/column:    {r['mean_column_params']:>12,.0f}")
    print(f"{'   ' * 60}")
    print(f"  Scaling Costs (estimated):")
    print(f"    Add 1 column:        ~{r['pct_cost_grow_1']:.1f}% of full retrain")
    print(f"    Split 1 column:      ~{r['pct_cost_split_1']:.1f}% of full retrain")
    print(f"    Full retrain:        100%")
    print(f"{'=' * 60}\n")



# ============================================================================
# Production Interface (Continuous Operation)
# Source: nct_v2/v11_production.py
# ============================================================================

"""
NCT V11: Production Integration
================================

Wires V11 CorticalColumnNetwork into production infrastructure:
  - Quantization:  ²-guided mixed precision for V11's cortical columns
  - Distributed: FSDP2 wrapping policy for column-based architecture
  - Alignment: SFT/DPO using V11's local learning interface
  - Agentic: Tool use + retrieval using V11's BG + hippocampal signals
  - Inference:  ²-early-exit for adaptive column depth

Each section provides the adapter needed to use V11 with existing
infrastructure (quantize.py, tool_use.py, retrieval.py, agent.py).
"""



# [bundled] from .v11_config import V11Config


#

# 1. Quantization -- V11-Specific Precision Mapping
#


```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```python
def v11_quantization_config():
    """
    Return V11-specific quantization config for quantize.py's NCTQuantizer.

    V11 parameter categories mapped to precision levels:
      - FP16:  ² (W_sigma), decay rates, oscillatory phases, grid-cell periods
      - INT8: column projections (L4/L5/L6), interneuron weights, gates
      - INT4: L2/3 association weights (bulk, high redundancy)

    Usage:
        # [bundled] from nct_v2.quantize import NCTQuantizer
        # [bundled] from nct_v2.v11_production import v11_quantization_config, v11_precision_map

        config = v11_quantization_config()
        quantizer = NCTQuantizer()
        quantized_model = quantizer.quantize(model, config)
    """
    # [bundled] from .quantize import NCTQuantConfig, Precision

    return NCTQuantConfig(
        # L2/3 association (largest weight block -- like attention, can compress)
        attention_weights=Precision.INT8,
        # L5 feedforward (like FFN -- high redundancy)
        ffn_weights=Precision.INT4,
        # Embedding + LM head (shared weights)
        embedding_weights=Precision.INT8,
        lm_head_weights=Precision.INT8,
        # Precision-critical brain parameters
        sigma2_weights=Precision.FP16,       #  ² directly affects PC loss
        gate_params=Precision.INT8,           # sigmoid gates -- 256 levels sufficient
        decay_rates=Precision.FP16,           # HPC/NMDA decay -- exponential compounding
        neuromod_weights=Precision.INT8,      # DA/ACh/NE/5-HT projections
        oscillatory_phases=Precision.FP16,    # phase parameters need precision
        position_encoding=Precision.FP16,     # grid-cell periods/offsets
        dendritic_ffn=Precision.INT8,         # k-WTA ordering sensitive
        competition_weights=Precision.INT8,   # BrainLMHead WEAVER++
        production_delta=Precision.INT8,      # LM head delta
        # Quantization settings
        group_size=128,
        symmetric=False,
        use_gptq=True,
    )


def v11_precision_map(model: nn.Module) -> Dict[str, str]:
    """
    Map V11 parameter names to precision categories.

    Returns: {param_name: category} where category matches NCTQuantConfig fields.
    Used by NCTQuantizer.quantize() to apply per-parameter precision.
    """
    precision_map = {}
    for name, param in model.named_parameters():
        if 'W_sigma' in name or 'sigma2' in name:
            precision_map[name] = 'sigma2_weights'
        elif 'gate_raw' in name:
            precision_map[name] = 'gate_params'
        elif 'decay' in name:
            precision_map[name] = 'decay_rates'
        elif 'brainstem' in name or 'neuromod' in name:
            precision_map[name] = 'neuromod_weights'
        elif 'osc' in name or 'period' in name or 'phase' in name:
            precision_map[name] = 'oscillatory_phases'
        elif 'grid_pos' in name:
            precision_map[name] = 'position_encoding'
        elif 'embed' in name:
            precision_map[name] = 'embedding_weights'
        elif 'lm_head' in name:
            if 'competition' in name or 'delta' in name:
                precision_map[name] = 'competition_weights'
            else:
                precision_map[name] = 'lm_head_weights'
        elif 'L23' in name and 'W_assoc' in name:
            precision_map[name] = 'attention_weights'
        elif 'L5' in name and ('W_up' in name or 'W_down' in name):
            precision_map[name] = 'ffn_weights'
        elif 'interneurons' in name:
            precision_map[name] = 'dendritic_ffn'
        else:
            # Default: backbone INT8
            precision_map[name] = 'attention_weights'

    return precision_map


#

# 2. Distributed -- FSDP2 Wrapping Policy for V11
#


```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```python
def apply_fsdp_v11(model: nn.Module, mp_policy=None):
    """
    Apply FSDP2 (fully_shard) to V11 CorticalColumnNetwork.

    Wrapping strategy:
      - Each CorticalColumn is one FSDP unit (all 6 layers + interneurons + WM stay together)
      - Subcortical systems (thalamus, hippocampus, BG, cerebellum, brainstem) stay in outer unit
      - Embedding + LM head: outer unit (weight-tied)
      - Lateral connections + workspace: outer unit (small params)

    Usage:
        # [bundled] from nct_v2.v11_production import apply_fsdp_v11
        model = create_v11_model(cfg).cuda()
        apply_fsdp_v11(model)
        # Now model is FSDP-sharded
    """
    from torch.distributed.fsdp import fully_shard, MixedPrecisionPolicy

    if mp_policy is None:
        mp_policy = MixedPrecisionPolicy(
            param_dtype=torch.bfloat16,
            reduce_dtype=torch.float32,
        )

    # Wrap each cortical column as one FSDP unit
    # Columns are the bulk of parameters -- sharding them gets the memory benefit.
    # Hooks, interneurons, WM within a column must NOT be split (stateful interaction).
    for col in model.columns:
        fully_shard(col, mp_policy=mp_policy)

    # Wrap full model (outer level catches embedding, subcortical, LM head)
    fully_shard(model, mp_policy=mp_policy)

    return model


#

# 3. Alignment -- SFT/DPO Adapters for V11
#


def v11_sft_loss(
    model: nn.Module,
    input_ids: torch.Tensor,
    targets: torch.Tensor,
    response_mask: Optional[torch.Tensor] = None,
    aux_weight: float = 0.1,
) -> Tuple[torch.Tensor, Dict[str, float]]:
    """
    Compute SFT loss for V11 with brain-mechanism preservation.

    Features:
      - Response-only loss (only supervise the model's response, not the prompt)
      -  ²-aware weighting: uncertain tokens get reduced loss weight
      - Auxiliary losses at 10% weight to preserve brain mechanisms
      - PC loss + cerebellar loss maintained alongside NTP

    Args:
        model: CorticalColumnNetwork
        input_ids: [B, T] input token IDs
        targets: [B, T] target token IDs (-1 for prompt tokens)
        response_mask: [B, T] binary mask (1 = response token, 0 = prompt token)
        aux_weight: weight for auxiliary losses (PC + cerebellar)

    Returns: (total_loss, metrics_dict)
    """
    result = model(input_ids, targets)
    logits = result['logits']
    metrics = {}

    # NTP loss (response-only if mask provided)
    if response_mask is not None:
        # Only compute loss on response tokens
        ntp_loss = F.cross_entropy(
            logits.view(-1, logits.shape[-1]),
            targets.view(-1),
            ignore_index=-1,
            reduction='none',
        ).view(targets.shape)

        #  ²-aware weighting: high uncertainty = reduce loss weight slightly
        # Brain principle: don't force confidence on genuinely ambiguous content
        sigma2_mean = result['metrics'].get('sigma2_mean', 1.0)
        sigma2_weight = 1.0 / (1.0 + 0.1 * max(0, sigma2_mean - 2.0))

        ntp_loss = (ntp_loss * response_mask * sigma2_weight).sum() / response_mask.sum().clamp(min=1)
    else:
        ntp_loss = result.get('loss', F.cross_entropy(
            logits.view(-1, logits.shape[-1]),
            targets.view(-1),
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
            ignore_index=-1,
        ))

    # Auxiliary losses (preserve brain mechanisms during SFT)
    pc_loss = result.get('pc_loss', torch.tensor(0.0))
    cb_loss = result.get('cb_loss', torch.tensor(0.0))
    trn_loss = result.get('trn_loss', torch.tensor(0.0))

    total_loss = ntp_loss + aux_weight * (pc_loss + cb_loss + trn_loss)

    metrics['ntp_loss'] = ntp_loss.item()
    metrics['pc_loss'] = pc_loss.item() if isinstance(pc_loss, torch.Tensor) else pc_loss
    metrics['cb_loss'] = cb_loss.item() if isinstance(cb_loss, torch.Tensor) else cb_loss
    metrics['total_loss'] = total_loss.item()
    metrics['sigma2_mean'] = result['metrics'].get('sigma2_mean', 0.0)

    return total_loss, metrics


def v11_dpo_loss(
    policy_model: nn.Module,
    ref_model: nn.Module,
    chosen_ids: torch.Tensor,
    rejected_ids: torch.Tensor,
    chosen_targets: torch.Tensor,
    rejected_targets: torch.Tensor,
    beta: float = 0.1,
    adaptive_beta: bool = True,
) -> Tuple[torch.Tensor, Dict[str, float]]:
    """
    DPO loss for V11 with  ²-adaptive   (brain-inspired preference learning).

    When the model is uncertain (high  ²), we reduce   -- don't force strong
    preferences on content the model genuinely finds ambiguous.

    Args:
        policy_model: current CorticalColumnNetwork being trained
        ref_model: frozen reference CorticalColumnNetwork
        chosen_ids/targets: preferred completion
        rejected_ids/targets: dispreferred completion
        beta: DPO temperature
        adaptive_beta: if True, scale   by 1/ ² (higher certainty → stronger preference)

    Returns: (dpo_loss, metrics_dict)
    """
    metrics = {}

    # Policy forward
    policy_chosen = policy_model(chosen_ids, chosen_targets)
    policy_rejected = policy_model(rejected_ids, rejected_targets)

    # Reference forward (no grad)
    with torch.no_grad():
        ref_chosen = ref_model(chosen_ids, chosen_targets)
        ref_rejected = ref_model(rejected_ids, rejected_targets)

    # Per-token log probabilities
    def get_logps(logits, targets):
        log_probs = F.log_softmax(logits.float(), dim=-1)
        mask = (targets != -1).float()
        per_token = torch.gather(log_probs, -1, targets.clamp(min=0).unsqueeze(-1)).squeeze(-1)
        return (per_token * mask).sum(dim=-1) / mask.sum(dim=-1).clamp(min=1)

    pi_chosen_lp = get_logps(policy_chosen['logits'], chosen_targets)
    pi_rejected_lp = get_logps(policy_rejected['logits'], rejected_targets)
    ref_chosen_lp = get_logps(ref_chosen['logits'], chosen_targets)
    ref_rejected_lp = get_logps(ref_rejected['logits'], rejected_targets)

    #  ²-adaptive  : higher certainty → stronger preference enforcement
    effective_beta = beta
    if adaptive_beta:
        sigma2_chosen = policy_chosen['metrics'].get('sigma2_mean', 1.0)
        sigma2_rejected = policy_rejected['metrics'].get('sigma2_mean', 1.0)
        avg_sigma2 = (sigma2_chosen + sigma2_rejected) / 2
        # Scale   inversely with uncertainty (  × 1/ ²)
        effective_beta = beta / max(avg_sigma2, 0.1)
        metrics['effective_beta'] = effective_beta

    # DPO loss: -log  (  × (log  /ref_chosen - log  /ref_rejected))
    chosen_reward = pi_chosen_lp - ref_chosen_lp
    rejected_reward = pi_rejected_lp - ref_rejected_lp

    dpo_loss = -F.logsigmoid(effective_beta * (chosen_reward - rejected_reward)).mean()

    metrics['dpo_loss'] = dpo_loss.item()
    metrics['chosen_reward'] = chosen_reward.mean().item()
    metrics['rejected_reward'] = rejected_reward.mean().item()
    metrics['reward_margin'] = (chosen_reward - rejected_reward).mean().item()

    return dpo_loss, metrics

```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
#

# 4. Agentic -- Tool Selection + Retrieval using V11 Signals
#


class V11ToolAdapter:
    """
    Adapter connecting V11's BG/ ²/ACC signals to tool_use.py infrastructure.

    V11 already HAS the biological circuits for tool use:
      - BasalGanglia: D1 (GO) / D2 (NO-GO) action selection
      -  ² (per-column): confidence in tool selection
      - Hippocampus: tool schema storage (like NMDA slots in V10.2)
      - SelfAssessment/ACC: conflict detection when tools ambiguous

    This adapter extracts those signals from V11's forward result
    and provides them to the existing ToolSelector/NCTAgent infrastructure.
    """

    def __init__(self, model):
        self.model = model

    def extract_signals(self, result: dict) -> dict:
        """
        Extract V11 signals needed for tool selection.

        Returns dict compatible with tool_use.py's ToolSelector.select():
          - sigma2: uncertainty level (float)
          - tonic_da: exploration tendency (float)
          - acc_conflict: ambiguity level (float)
          - residual: final hidden state [B, T, D]
        """
        metrics = result.get('metrics', {})
        return {
            'sigma2': metrics.get('sigma2_mean', 1.0),
            'tonic_da': metrics.get('tonic_da', 0.5),
            'acc_conflict': metrics.get('conflict', 0.0),
            'residual': result['logits'],  # Last layer output before argmax
            'da_mean': metrics.get('da_mean', 0.5),
            'ne_mean': metrics.get('ne_mean', 0.5),
        }

    def should_use_tool(self, result: dict, threshold: float = 0.3) -> bool:
        """
        Brain-inspired tool use decision using V11's intrinsic signals.

        The basal ganglia implements GO/NO-GO for action selection.
        High DA + high  ² = uncertain + exploring → try a tool.
        Low  ² = confident in text generation → don't interrupt with tool.
        """
        signals = self.extract_signals(result)
        sigma2 = signals['sigma2']
        tonic_da = signals['tonic_da']

        # DA modulates threshold (high DA → lower threshold → more tool exploration)
        effective_threshold = threshold / max(0.3 + tonic_da, 0.1)

        # Tool use when uncertain enough
        return sigma2 > effective_threshold

    def get_tool_confidence(self, result: dict) -> float:
        """
        Confidence in tool selection (1.0 = very confident, 0.0 = uncertain).

        Uses ACC conflict: high conflict = low confidence = maybe ask user.
        """
        conflict = result.get('metrics', {}).get('conflict', 0.0)
        return max(0.0, 1.0 - conflict)


class V11RetrievalAdapter:
    """
    Adapter connecting V11's  ²/hippocampal signals to retrieval.py infrastructure.

    V11's hippocampus naturally implements retrieval:
      - High  ² → uncertain → retrieve external knowledge
      - Low  ² → confident → generate from cortical knowledge
      - Hippocampal pattern completion ~ associative retrieval

    This adapter provides the gate signal for  ²-optimal retrieval (Thm 72).
    """

    def __init__(self, model):
        self.model = model

    def should_retrieve(self, result: dict, sigma2_threshold: float = 2.0) -> bool:
        """
         ²-gated retrieval: only retrieve when model is uncertain.

```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        Brain principle: the hippocampus supplements cortical knowledge
        when the cortex can't make a confident prediction on its own.
        """
        sigma2 = result.get('metrics', {}).get('sigma2_mean', 1.0)
        return sigma2 > sigma2_threshold

    def get_retrieval_query(self, model, input_ids: torch.Tensor) -> torch.Tensor:
        """
        Extract query embedding for retrieval from V11's last column output.

        Uses the final column's representation as the retrieval query --
        this is the most abstract, goal-relevant representation.
        """
        result = model(input_ids)
        # Last token of last column = most processed representation
        # logits are the output, but we want pre-lm-head representation
        # Access via last column output stored during forward
        if model._last_column_outputs:
            return model._last_column_outputs[-1].output[:, -1, :]  # [B, D]
        return result['logits'][:, -1, :]  # fallback


#

# 5. Inference --  ²-Early-Exit for Adaptive Column Depth
#


class V11EarlyExitConfig:
    """Configuration for  ²-guided early exit in V11."""
    sigma2_threshold: float = 0.3   # Exit when  ² drops below this
    min_columns: int = 2            # Always run at least 2 columns
    enabled: bool = True


def v11_forward_with_early_exit(
    model,
    input_ids: torch.Tensor,
    targets=None,
    sigma2_threshold: float = 0.3,
    min_columns: int = 2,
) -> Dict[str, torch.Tensor]:
    """
    V11 forward with  ²-guided early exit.

    Brain principle: the brain doesn't process every input to the same depth.
    Easy inputs (low  ²) are resolved in early sensory areas.
    Hard inputs (high  ²) propagate to deeper prefrontal processing.

    This skips remaining columns once  ² drops below threshold,
    saving compute on easy tokens.

    Args:
        model: CorticalColumnNetwork
        input_ids: [B, T]
        targets: [B, T] optional
        sigma2_threshold: exit when mean  ² < this
        min_columns: always process at least this many columns

    Returns: same dict as standard forward, plus 'columns_used' metric
    """
    B, T = input_ids.shape
    device = input_ids.device
    cfg = model.cfg

    # Embedding
    x = model.embed(input_ids)
    x = model.embed_norm(x)
    x = model.embed_dropout(x)
    rope_cos, rope_sin = model.grid_pos(T, device, x.dtype, X=x)

    if cfg.thalamic_enabled:
        model.thalamus.reset_loss()

    # Process columns with early exit
    column_outputs = []
    thal_feedbacks = [None] * cfg.n_columns
    h = x
    columns_used = 0

    for i, col in enumerate(model.columns):
        h_thal = model.thalamus.route(h, column_idx=i, thal_feedback=thal_feedbacks[i])
        out = col(ff_input=h_thal, td_context=None,
                  rope_cos=rope_cos, rope_sin=rope_sin)
        column_outputs.append(out)
        thal_feedbacks[i] = out.thal_feedback
        h = out.output
        columns_used = i + 1

        # Early exit check: if  ² is low enough after minimum columns
        if i >= min_columns - 1:
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
            mean_sigma2 = out.sigma2.mean().item()
            if mean_sigma2 < sigma2_threshold:
                break

    # Skip remaining PC inference (early exit means we're confident)
    # Go directly to output

    # Brainstem neuromod
    top_output = column_outputs[-1].output
    errors = []
    for l in range(len(column_outputs)):
        target = x.detach() if l == 0 else column_outputs[l - 1].output.detach()
        errors.append(target - column_outputs[l].prediction)

    neuromod = model.brainstem.compute_state(
        top_output,
        prediction_error=errors[-1] if errors else None,
    )

    # BG + cerebellum + hippocampus
    if cfg.bg_enabled:
        h = model.basal_ganglia(h, neuromod.dopamine)
    if cfg.cerebellum_enabled:
        h, cb_loss = model.cerebellum(h)
    else:
        cb_loss = torch.tensor(0.0, device=device)
    if cfg.hpc_enabled:
        mean_sigma2_t = sum(out.sigma2 for out in column_outputs) / len(column_outputs)
        h = model.hippocampus(h, mean_sigma2_t)

    # LM head
    logits = model.lm_head(model.lm_head_norm(h))

    result = {'logits': logits, 'cb_loss': cb_loss}
    if targets is not None:
        result['loss'] = F.cross_entropy(
            logits.view(-1, cfg.vocab_size),
            targets.view(-1),
            ignore_index=-1,
        )

    # PC loss for columns that were used
    predictions = [out.prediction for out in column_outputs]
    pred_targets = [x.detach()] + [column_outputs[l].output.detach() for l in range(len(column_outputs) - 1)]
    precisions = [out.sigma2 for out in column_outputs]
    _, pc_loss = model.learning_manager.pc_optimizer.compute_column_losses(
        predictions, pred_targets, precisions
    )
    result['pc_loss'] = pc_loss

    # Metrics
    all_sigma2 = torch.cat([out.sigma2 for out in column_outputs], dim=-1)
    result['metrics'] = {
        'sigma2_mean': all_sigma2.mean().item(),
        'columns_used': columns_used,
        'columns_total': cfg.n_columns,
        'columns_saved': cfg.n_columns - columns_used,
        'da_mean': neuromod.dopamine.mean().item(),
        'tonic_da': neuromod.tonic_da,
    }

    return result


#

# Convenience: Full Production Setup
#


def setup_v11_production(model):
    """
    Attach all production adapters to a V11 model.

    After calling this, the model has:
      model.tools    -- V11ToolAdapter
      model.retrieval -- V11RetrievalAdapter
```

Listing 15: Production Interface — complete source code (v11_production.py, 567 lines)

17 Training Script

Source: `train_v11.py` **Lines:** 1,078

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

17.1 Training Modes

NCT supports four training modes, each implementing a different level of biological fidelity:

Table 12: NCT training modes.

Mode	Learning Rule	NTP Backprop	Use Case
`hebbian`	Hebbian + anti-Hebbian only	None	Purest biological, proof of concept
`predictive_coding`	Local PC error signals per column	None (analytical LM head only)	Brain-like, zero global backprop
`hybrid`	PC + Hebbian + analytical NTP on output pathway	LM head + final L5 only	Best performance, mostly local
`backprop`	Standard backprop	Full global	Baseline comparison only

17.2 Local Output Correction

In `predictive_coding` and `hebbian` modes, the only component receiving any form of gradient is the LM head, via an analytical delta rule. This is the critical mechanism that enables language modeling without any backpropagation through the column hierarchy:

$$\Delta W_{\text{LM}} = -\eta_{\text{out}} \cdot \frac{\partial \mathcal{L}_{\text{NTP}}}{\partial W_{\text{LM}}}, \qquad \eta_{\text{out}} = 100 \times \eta_{\text{base}}, \qquad \|\Delta W\|_{\max} = 1.0 \tag{97}$$

The correction is computed **analytically** (not via autograd) using the chain rule applied only to the BrainLMHead's parameters. Specifically, the error signal $\boldsymbol{\delta} = \text{softmax(logits)} - \text{one_hot(target)}$ is projected back through the LM head's low-rank production delta (W_A, W_B) and output bias $(\mathbf{b}_{\text{freq}})$ using simple matrix multiplications. No computational graph is built, no `.backward()` is called, and no gradients flow to any column parameter.

The 100× learning rate multiplier compensates for the fact that this is the *only* supervised signal in the entire system — all column learning comes from local PC and Hebbian rules. Gradient clipping at $\|\Delta W\|_{\max} = 1.0$ prevents large corrections from destabilizing the LM head. Smoke test 11 verifies this property: column L2/3 weights, column L5 weights, and all hidden layer parameters show exactly zero change after a local output correction, while the LM head's `delta_A` and `output_bias` show nonzero changes.

17.3 Data Loading

Training data is loaded from WikiText-2 via HuggingFace `datasets` library and tokenized with the NCT morphological tokenizer. The data loading pipeline:

1. Load the WikiText-2 raw text corpus (`wikitext-2-raw-v1` split)
2. Collect all non-empty text segments
3. Build or load the NCT morphological tokenizer vocabulary (from `nct_tokenizer/vocab/`; if not found, builds on-the-fly from the training corpus with `max_roots=8000, max_whole_words=5000, min_word_freq=3, min_morpheme_freq=2`)
4. Encode all text segments to token IDs using the dual-route tokenizer
5. Concatenate all token IDs into a single long sequence
6. Chunk into sequences of length `seq_len` (default 256)
7. Create input/target pairs: `input = chunk[:-1]`, `target = chunk[1:]`

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

The HF_TOKEN environment variable is required for WikiText-2 download. If the `datasets` library is not available, the training script falls back to random data (for smoke testing).

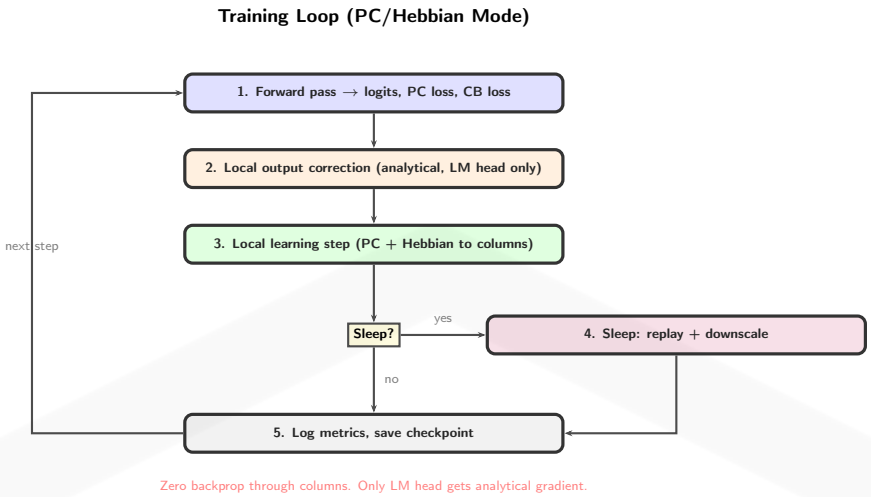


Figure 85: Training loop in PC/Hebbian mode. Step 2 uses an analytical delta rule (not autograd) applied only to the LM head. Step 3 applies local PC + Hebbian updates to column weights. No global backpropagation through the column hierarchy.

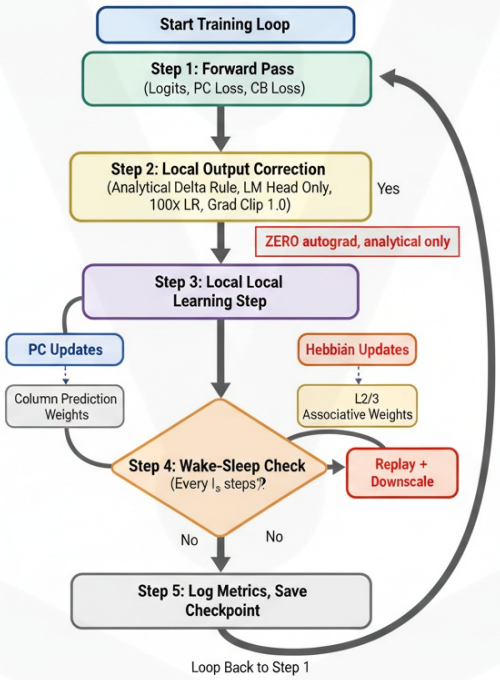


Figure 86: (Illustrated) Training loop in PC/Hebbian mode. Step 2 uses an analytical delta rule (not autograd) applied only to the LM head. Step 3 applies local PC + Hebbian updates to column weights. No global backpropaga...

17.4 Training Loop

For each step in the training loop:

Backprop mode: Standard forward → loss.backward() → optimizer.step().

PC/Hebbian/Hybrid modes:

1. Forward pass (computes logits, PC loss, cerebellar loss)

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

2. Local output correction (analytical, LM head only)
3. Local learning step (PC + Hebbian updates to column weights)
4. Wake-sleep check (every I_s steps, run sleep phase)
5. Logging and checkpointing

17.5 Prenatal Phase

If `-prenatal N` is specified (recommended: $N = 500$), prenatal self-organization runs *before* any data-driven training. This phase bootstraps the network from random weights through four types of spontaneous activity:

Spontaneous waves (30% of steps): Momentum random walk in embedding space simulating retinal waves. This teaches L2/3 about local temporal correlations: "nearby positions have related representations."

Motor babbling (25%): Auto-regressive self-generation from random seed tokens (capped at 16 tokens for speed). The model learns to predict its own output, calibrating the production-perception loop.

Competitive differentiation (25%): Structured patterns (repeated motifs, ascending sequences, alternating tokens, segmented blocks, pure random) force different columns to develop different selectivities through competitive Hebbian learning.

Oscillatory calibration (20%): Multi-frequency sinusoidal input spanning theta/alpha/delta ranges. Calibrates the learnable oscillatory periods and PAC coupling strengths.

After prenatal development, the network is not a blank slate — it has calibrated oscillatory rhythms, differentiated columns, functional prediction circuitry, and tuned BCM thresholds. This is the "DNA" — the structured starting point from which rapid postnatal learning from real data can begin immediately. See section 14 for the complete prenatal algorithm.

17.6 Smoke Test Suite (12 Tests)

The `-smoke` flag runs a comprehensive validation suite:

Table 13: NCT smoke test suite.

#	Test	Validates
1	Forward pass	Correct output shapes, finite loss, finite PC loss
2	Local learning	PC + Hebbian updates run without error, metrics returned
3	Backprop step	Standard backprop produces finite gradients
4	Wake-sleep	Sleep triggers correctly, replay executes, downscaling applies
5	Generation	Auto-regressive generation produces valid token sequences
6	Numerical stability	All model parameters are finite (no NaN/Inf)
7	Gate values	All gates $\approx \sigma(-2) \approx 0.12$ at init (identity-at-init verified)
8	Autonomous loop	AutonomousLoop processes steps, accumulates experiences, dreams
9	Always-on learning	Learning metrics computed after every forward pass
10	L2/3 complexity	Scaling ratio T=128/T=32 $< 10\times$ (linear, not $O(T^2)$ quadratic)
11	Local output correction	Column weights unchanged (zero autograd), LM head weights changed
12	Prenatal	All 4 phases visited, weight stats tracked, report generated

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

17.7 Checkpoint Format

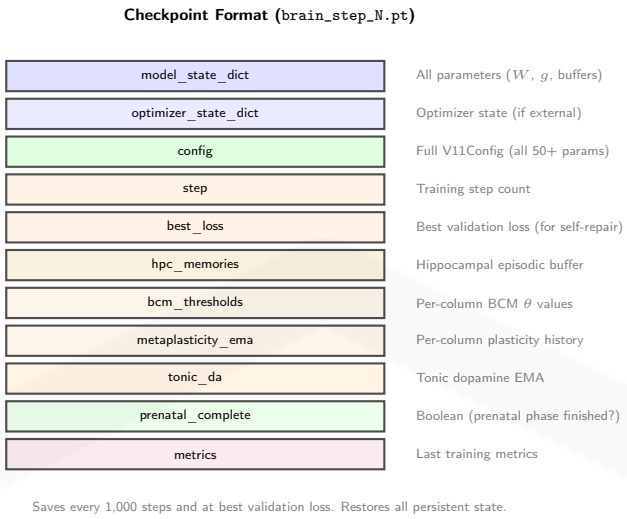


Figure 87: Checkpoint format: saves all model parameters plus brain-specific persistent state (HPC memories, BCM thresholds, metaplasticity history, tonic DA, prenatal completion flag). Loading restores the full brain state including working memory pointers and neuromodulator baselines.

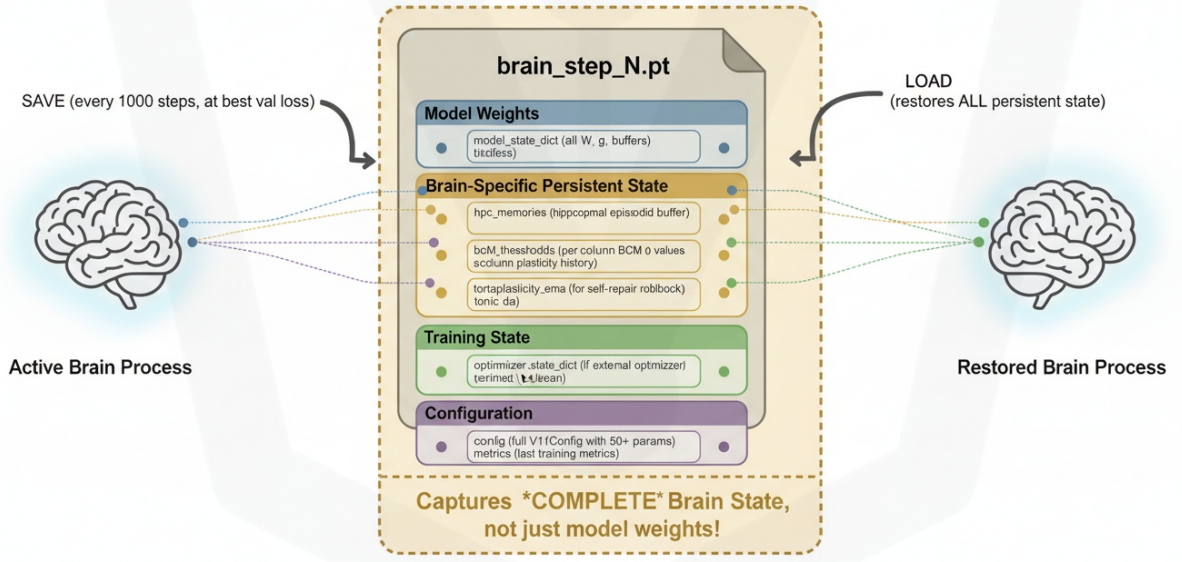


Figure 88: (Illustrated) Checkpoint format: saves all model parameters plus brain-specific persistent state (HPC memories, BCM thresholds, metaplasticity history, tonic DA, prenatal completion flag). Loading restores the f...

17.8 CLI Reference

```
python3 nct_v11_standalone.py --smoke                    # 12 smoke tests
python3 nct_v11_standalone.py --mode predictive_coding \
    --steps 5000 --prenatal 500 --bf16 --wake_sleep      # PC training
python3 nct_v11_standalone.py --preset laptop \
    --mode hybrid --steps 10000 --bf16                    # Laptop preset
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
python3 nct_v11_standalone.py --mode hebbian \
    --steps continuous --bf16 --wake_sleep                # Never-ending
python3 nct_v11_standalone.py --resume_latest --bf16      # Resume
```

17.9 Source Code

```
    Usage:
        model = create_v11_model(cfg)
        setup_v11_production(model)

        result = model(input_ids)
        if model.tools.should_use_tool(result):
            # Use tool_use.py infrastructure
            ...
        if model.retrieval.should_retrieve(result):
            # Use retrieval.py infrastructure
            ...
    """
    model.tools = V11ToolAdapter(model)
    model.retrieval = V11RetrievalAdapter(model)
    return model



# ============================================================================
# Training Script
# Source: train_v11.py
# ============================================================================

"""
NCT V11: Training Script -- Cortical Column Network
==================================================

Training modes:
  1. backprop: Standard backprop (baseline comparison)
  2. hybrid: PC + Hebbian + backprop NTP loss (recommended for Phase 1)
  3. predictive_coding: Pure local PC learning (Theorem A)
  4. hebbian: Pure Hebbian learning (Theorem B -- experimental)

Wake-Sleep cycle (Theorem D):
  - Wake: process input + learn (PC/Hebbian/backprop)
  - Sleep: replay from hippocampus + synaptic downscaling
  - Cycle runs autonomously based on config.sleep_interval

Usage:
  # Smoke test (quick validation)
  python3 train_v11.py --steps 10 --smoke

  # Backprop baseline (standard training for comparison)
  python3 train_v11.py --mode backprop --steps 5000

  # Hybrid mode (recommended -- PC + Hebbian + NTP)
  python3 train_v11.py --mode hybrid --steps 5000

  # Pure predictive coding (Theorem A -- no global backprop)
  python3 train_v11.py --mode predictive_coding --steps 5000

  # Full V11 with wake-sleep
  python3 train_v11.py --mode hybrid --wake_sleep --steps 10000
"""



def _cuda_cleanup():
    """Release all CUDA memory on exit. Prevents driver memory leaks in containers."""
    try:
        import torch
        if torch.cuda.is_available():
            torch.cuda.empty_cache()
            torch.cuda.synchronize()
            # Reset device to ensure driver reclaims all memory
            torch.cuda.reset_peak_memory_stats()
    except Exception:
        pass  # Best effort -- don't crash on cleanup


atexit.register(_cuda_cleanup)


# Add parent to path
sys.path.insert(0, str(Path(__file__).parent))

# [bundled] from nct_v2.v11_config import V11Config
# [bundled] from nct_v2.v11_model import CorticalColumnNetwork, create_v11_model


```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
#

# Data Loading
#


def get_random_batch(vocab_size: int, batch_size: int, seq_len: int, device: torch.device):
    """Generate random batch for smoke testing."""
    input_ids = torch.randint(0, vocab_size, (batch_size, seq_len), device=device)
    targets = torch.randint(0, vocab_size, (batch_size, seq_len), device=device)
    return input_ids, targets


def _get_or_build_nct_tokenizer(corpus_text: str = None):
    """Load NCT tokenizer from disk, or build from corpus if not available."""
    # [bundled] from nct_tokenizer.tokenizer import NCTTokenizer

    vocab_dir = os.path.join(os.path.dirname(__file__), 'nct_tokenizer', 'vocab')
    if os.path.exists(os.path.join(vocab_dir, 'vocab.json')):
        return NCTTokenizer.load(vocab_dir)

    # Build vocab from provided corpus
    if corpus_text is None:
        raise RuntimeError(
            "NCT tokenizer vocab not found. Build it first:\n"
            "  python3 -m nct_tokenizer.build_vocab\n"
            "Or pass corpus_text to build on the fly."
        )

    print("[NCT Tokenizer] Building vocab from training corpus...")
    tok = NCTTokenizer()
    tok.build_vocab(corpus_text, max_roots=8000, max_whole_words=5000,
                    min_word_freq=3, min_morpheme_freq=2)
    tok.save(vocab_dir)
    return tok


def load_wikitext_data(seq_len: int, split: str = 'train', variant: str = 'auto'):
    """Load WikiText dataset, tokenized with NCT morphological tokenizer.

    The brain processes morphemes, not BPE byte-pairs. V11 uses the NCT
    dual-route tokenizer (Coltheart 2001 DRC model) as its ONLY tokenizer.

    Args:
        variant: 'auto' tries WikiText-103 first (100M tokens), falls back to
                 WikiText-2 (5.5M tokens). Use '103' or '2' to force.
    """
    try:
        from datasets import load_dataset
    except ImportError:
        print("WARNING: datasets not available, using random data")
        return None, None

    # Try WikiText-103 first (20x more data than WikiText-2)
    dataset = None
    if variant in ('auto', '103'):
        try:
            print(f"Loading WikiText-103 ({split})...")
            dataset = load_dataset('wikitext', 'wikitext-103-raw-v1', split=split)
            print(f"  WikiText-103 loaded: {len(dataset)} examples")
        except Exception as e:
            if variant == '103':
                raise
            print(f"  WikiText-103 unavailable ({e}), falling back to WikiText-2")

    if dataset is None:
        print(f"Loading WikiText-2 ({split})...")
        dataset = load_dataset('wikitext', 'wikitext-2-raw-v1', split=split)

    # Collect raw text for vocab building (if needed)
    texts = [ex['text'].strip() for ex in dataset if ex['text'].strip()]
    corpus = "\n".join(texts)

    # NCT morphological tokenizer -- the brain's tokenizer
    tokenizer = _get_or_build_nct_tokenizer(corpus_text=corpus)

    all_tokens = []
    for text in texts:
        enc = tokenizer.encode(text)
        all_tokens.extend(enc.ids)

    all_tokens = torch.tensor(all_tokens, dtype=torch.long)
    print(f"  Loaded {len(all_tokens):,} tokens (NCT morphological, V={tokenizer.vocab_size})")

    return all_tokens, tokenizer


class TextEnvironment:
    """
    Autonomous data acquisition for V11.

```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
    Reads text files from a directory, tokenizes them, and serves batches.
    The model can point at a directory and learn forever.

    Supports:
    - Reading all .txt files from a directory
    - Cycling through data infinitely
    - Shuffling files between epochs
    - Reporting what file is being processed (for logging)
    """

    def __init__(self, data_dir: str, tokenizer, seq_len: int = 256,
                 batch_size: int = 2, device: torch.device = torch.device('cpu')):
        import glob as _glob
        self.tokenizer = tokenizer
        self.seq_len = seq_len
        self.batch_size = batch_size
        self.device = device
        self.files = sorted(_glob.glob(os.path.join(data_dir, '**/*.txt'), recursive=True))
        if not self.files:
            # Fallback: try any text-like files
            for ext in ['*.md', '*.py', '*.json', '*.csv', '*.rst']:
                self.files.extend(_glob.glob(os.path.join(data_dir, f'**/{ext}'), recursive=True))
            self.files = sorted(self.files)
        self._token_buffer = torch.tensor([], dtype=torch.long)
        self._file_idx = 0
        self._epoch = 0
        self.current_file = None
        print(f"  TextEnvironment: {len(self.files)} files from {data_dir}")

    def _load_next_file(self):
        """Load and tokenize the next file."""
        if not self.files:
            return
        if self._file_idx >= len(self.files):
            self._file_idx = 0
            self._epoch += 1
            import random
            random.shuffle(self.files)
        path = self.files[self._file_idx]
        self._file_idx += 1
        self.current_file = os.path.basename(path)
        try:
            with open(path, 'r', encoding='utf-8', errors='ignore') as f:
                text = f.read()
            if text.strip():
                enc = self.tokenizer.encode(text)
                new_tokens = torch.tensor(enc.ids, dtype=torch.long)
                self._token_buffer = torch.cat([self._token_buffer, new_tokens])
        except Exception:
            pass  # Skip unreadable files

    def get_batch(self) -> tuple:
        """Get next batch of (input_ids, targets)."""
        needed = (self.seq_len + 1) * self.batch_size
        while len(self._token_buffer) < needed:
            self._load_next_file()
            if not self.files:
                # No files -- return random data as fallback
                rand = torch.randint(0, 1000, (self.batch_size, self.seq_len + 1))
                return rand[:, :-1].to(self.device), rand[:, 1:].to(self.device)

        # Extract batch from buffer
        chunks = []
        for _ in range(self.batch_size):
            chunk = self._token_buffer[:self.seq_len + 1]
            chunks.append(chunk)
            self._token_buffer = self._token_buffer[self.seq_len:]

        batch = torch.stack(chunks)  # [B, seq_len+1]
        return batch[:, :-1].to(self.device), batch[:, 1:].to(self.device)


def get_batch_from_tokens(tokens: torch.Tensor, batch_size: int, seq_len: int,
                          device: torch.device, step: int):
    """Get a batch from pre-tokenized data."""
    n = tokens.shape[0]
    max_start = n - seq_len - 1
    if max_start <= 0:
        max_start = 1

    # Deterministic batch selection based on step
    starts = [(step * batch_size + i) * seq_len % max_start for i in range(batch_size)]
    input_ids = torch.stack([tokens[s:s + seq_len] for s in starts]).to(device)
    targets = torch.stack([tokens[s + 1:s + seq_len + 1] for s in starts]).to(device)
    return input_ids, targets


#

# Local Output Correction (Zero Backprop)
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
#


_loc_diag_count = [0]  # Mutable counter for one-time diagnostic

@torch.no_grad()
def _local_output_correction(model: CorticalColumnNetwork, result: dict, lr: float):
    """
    Analytical local error feedback for the output pathway -- NO autograd.

    The cross-entropy gradient for a linear LM head is:
        dL/dW = (softmax(logits) - one_hot(targets))^T @ hidden / N
    This is the delta rule (Widrow-Hoff 1960), identical to what backprop
    computes for a single linear layer, but calculated directly.

    Biologically: auditory cortex computes speech error (heard vs intended)
    and corrects motor-to-speech weights. Local computation -- each output
    neuron adjusts its own synaptic weights based on its own error signal.
    """
    logits = result.get('logits')
    targets = result.get('targets')
    hidden = result.get('hidden_state')  # [B, T, D] from lm_head_norm output

    if logits is None or targets is None or hidden is None:
        if _loc_diag_count[0] < 3:
            print(f"  [LOC] SKIPPED: logits={logits is not None}, targets={targets is not None}, hidden={hidden is not
                None}")
            _loc_diag_count[0] += 1
        return

    B, T, V = logits.shape
    D = hidden.shape[-1]
    N = B * T

    #        Output error: softmax(logits) - one_hot(targets)
    # This IS the analytical gradient of cross-entropy w.r.t. logits
    probs = F.softmax(logits.float(), dim=-1)  # [B, T, V], float32 for precision
    targets_flat = targets.reshape(-1).clamp(0, V - 1)  # Handle padding
    one_hot = torch.zeros(N, V, device=logits.device, dtype=probs.dtype)
    one_hot.scatter_(1, targets_flat.unsqueeze(1), 1.0)
    error = probs.reshape(N, V) - one_hot  # [N, V]

    # Mask padding tokens (target == -1)
    valid_mask = (targets.reshape(-1) >= 0).float().unsqueeze(1)  # [N, 1]
    error = error * valid_mask
    n_valid = valid_mask.sum().clamp(min=1)

    #        LM head updates: BrainLMHead has delta_A, delta_B, output_bias
    hidden_flat = hidden.reshape(N, D).float()
    lm = model.lm_head

    # Gradient clipping: prevents divergence when output bias concentrates
    # probability mass and error signals grow for rare tokens.
    MAX_GRAD_NORM = 1.0

    # Output bias: dL/d(bias) = mean(error) per vocab entry
    if hasattr(lm, 'output_bias'):
        db = error.sum(dim=0) / n_valid  # [V]
        db_norm = db.norm()
        if db_norm > MAX_GRAD_NORM:
            db = db * (MAX_GRAD_NORM / db_norm)
        lm.output_bias.data -= lr * db.to(lm.output_bias.dtype)

    # Production delta (low-rank: delta_A @ delta_B approximates  W )
    if hasattr(lm, 'delta_A') and hasattr(lm, 'delta_B'):
        h_proj = hidden_flat @ lm.delta_B.t().float()  # [N, r]
        dA = (error.T @ h_proj) / n_valid  # [V, r]
        dA_norm = dA.norm()
        if dA_norm > MAX_GRAD_NORM:
            dA = dA * (MAX_GRAD_NORM / dA_norm)
        lm.delta_A.data -= lr * dA.to(lm.delta_A.dtype)

        error_proj = error @ lm.delta_A.float()  # [N, r]
        dB = (error_proj.T @ hidden_flat) / n_valid  # [r, D]
        dB_norm = dB.norm()
        if dB_norm > MAX_GRAD_NORM:
            dB = dB * (MAX_GRAD_NORM / dB_norm)
        lm.delta_B.data -= lr * dB.to(lm.delta_B.dtype)

    if _loc_diag_count[0] < 3:
        print(f"  [LOC] Applied: lr={lr}, |error|={error.abs().mean():.6f}, "
              f"|dA|={dA.abs().mean() if hasattr(lm, 'delta_A') else 0:.8f}, "
              f"|db|={db.abs().mean() if hasattr(lm, 'output_bias') else 0:.8f}")
        _loc_diag_count[0] += 1


#

# Training Loop
#
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
346
347 def train(args):
348     """Main training function."""
349     device = torch.device('cuda' if torch.cuda.is_available() else
350                           'mps' if torch.backends.mps.is_available() else 'cpu')
351     print(f"Device: {device}")
352
353     #        Config
354     _preset_constructors = {
355         '1b': V11Config.v11_1b,
356         'laptop': V11Config.v11_laptop,
357     }
358     if args.preset in _preset_constructors:
359         cfg = _preset_constructors[args.preset](
360             learning_mode=args.mode,
361             wake_sleep_enabled=args.wake_sleep,
362             sleep_interval=args.sleep_interval,
363             base_lr=args.lr,
364             batch_size=args.batch_size,
365             seq_len=args.seq_len,
366             n_inference_steps=args.inference_steps,
367             pc_max_steps=args.inference_steps,     # Wire --inference_steps to actual PC loop bound
368             pc_min_steps=min(2, args.inference_steps),
369             dropout=0.1 if not args.smoke else 0.0,
370             use_bf16=args.bf16 and device.type == 'cuda',
371         )
372     else:
373         cfg = V11Config(
374             d_model=args.d_model,
375             d_column=args.d_model,  # Same as d_model for simplicity
376             n_columns=args.n_columns,
377             d_ff=args.d_model * 4,
378             vocab_size=args.vocab_size,
379             max_seq_len=args.seq_len * 2,
380             learning_mode=args.mode,
381             wake_sleep_enabled=args.wake_sleep,
382             sleep_interval=args.sleep_interval,
383             base_lr=args.lr,
384             batch_size=args.batch_size,
385             seq_len=args.seq_len,
386             n_inference_steps=args.inference_steps,
387             dropout=0.1 if not args.smoke else 0.0,
388             use_bf16=args.bf16 and device.type == 'cuda',
389         )
390
391     #        Model
392     model = create_v11_model(cfg).to(device)
393     if not args.smoke and device.type == 'cuda' and hasattr(torch, 'compile'):
394         try:
395             model = torch.compile(model, mode='reduce-overhead')
396             print("  torch.compile enabled (reduce-overhead)")
397         except Exception as e:
398             print(f"  torch.compile failed ({e}), continuing without")
399     param_counts = model.count_parameters()
400
401     print(f"\n{'=' * 60}")
402     print(f"NCT V11: Cortical Column Network")
403     print(f"{'=' * 60}")
404     print(f"  Mode:           {args.mode}")
405     print(f"  Columns:        {cfg.n_columns}")
406     print(f"  D_model:        {cfg.d_model}")
407     print(f"  D_ff:           {cfg.d_ff}")
408     print(f"  Assoc heads:    {cfg.n_assoc_heads}")
409     print(f"  Inference steps:{cfg.n_inference_steps}")
410     print(f"  Wake-sleep:     {cfg.wake_sleep_enabled}")
411     print(f"  Vocab:          {cfg.vocab_size}")
412     print(f"  Parameters:")
413     for k, v in param_counts.items():
414         print(f"    {k:20s}: {v:>12,}")
415     print(f"{'=' * 60}\n")
416
417     #        Data (NCT morphological tokenizer -- the brain's tokenizer)
418     train_tokens = None
419     tokenizer = None
420     text_env = None
421     if not args.smoke:
422         if args.data_file is not None and os.path.exists(args.data_file):
423             # Memory-mapped binary token file (from prepare_openwebtext.py)
424             # Zero RAM overhead -- reads directly from disk via mmap
425             import struct as _struct
426             print(f"Loading mmap token file: {args.data_file}")
427             with open(args.data_file, 'rb') as _f:
428                 magic = _f.read(4)
429                 assert magic == b'NCT1', f"Bad magic: {magic} (expected NCT1)"
430                 file_vocab = _struct.unpack('<I', _f.read(4))[0]
431                 _f.read(4)  # reserved
432             file_size = os.path.getsize(args.data_file)
433             n_file_tokens = (file_size - 12) // 4
434             import numpy as np
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
            mmap_tokens = np.memmap(args.data_file, dtype=np.int32, mode='r',
                                    offset=12, shape=(n_file_tokens,))
            train_tokens = torch.from_numpy(mmap_tokens.astype(np.int64))
            cfg.vocab_size = file_vocab
            print(f"  Loaded {n_file_tokens:,} tokens ({n_file_tokens/1e9:.2f}B) via mmap (V={file_vocab})")
            print(f"  Chinchilla ratio: {n_file_tokens/300e6:.1f} tokens/param")
            model = create_v11_model(cfg).to(device)
            # Load tokenizer for generation (lightweight, no corpus needed)
            tokenizer = _get_or_build_nct_tokenizer()
        else:
            train_tokens, tokenizer = load_wikitext_data(args.seq_len, variant=args.wikitext)
            if tokenizer is not None:
                cfg.vocab_size = tokenizer.vocab_size
                # Rebuild model with correct vocab size
                print(f"  NCT tokenizer: V={cfg.vocab_size} (morpheme dual-route)")
                model = create_v11_model(cfg).to(device)

        # Autonomous data environment (reads from directory)
        if args.data_dir is not None and tokenizer is not None:
            text_env = TextEnvironment(
                args.data_dir, tokenizer, seq_len=args.seq_len,
                batch_size=args.batch_size, device=device,
            )

    #        Validation data (for generalization monitoring)
    val_tokens = None
    if not args.smoke and tokenizer is not None:
        val_data = load_wikitext_data(args.seq_len, split='validation',
                                      variant=args.wikitext if hasattr(args, 'wikitext') else 'auto')
        if val_data[0] is not None:
            val_tokens = val_data[0]
            print(f"  Validation tokens: {len(val_tokens):,}")

    #        Optimizer (for backprop mode only -- hybrid uses NTP-output-only)
    optimizer = None
    if args.mode == 'backprop':
        param_groups = model.get_optimizer_groups(lr=args.lr)
        optimizer = torch.optim.AdamW(param_groups, betas=(0.9, 0.95))

    #        AutonomousLoop (for hybrid/PC/hebbian modes)
    loop = None
    if args.mode in ('hybrid', 'predictive_coding', 'hebbian'):
        # [bundled] from nct_v2.v11_autonomous import AutonomousLoop, AutonomousConfig
        auto_cfg = AutonomousConfig(
            sleep_interval=cfg.sleep_interval if cfg.wake_sleep_enabled else 10000,
            eval_interval=50,
            consolidation_interval=200,
        )
        loop = AutonomousLoop(model, auto_cfg)

    #        Auto-resume: brain_best.pt, fallback to brain_latest.pt if corrupted
    if args.resume_latest and args.save_dir and not args.resume:
        best_path = os.path.join(args.save_dir, 'brain_best.pt')
        latest_path = os.path.join(args.save_dir, 'brain_latest.pt')
        if os.path.exists(best_path):
            try:
                torch.load(os.path.join(args.save_dir, 'meta_best.pt'), map_location='cpu', weights_only=False)
                args.resume = best_path
                print(f"  [AUTO-RESUME] Found brain_best.pt")
            except Exception:
                print(f"  [AUTO-RESUME] brain_best.pt corrupted, trying brain_latest.pt")
                if os.path.exists(latest_path):
                    args.resume = latest_path
                    print(f"  [AUTO-RESUME] Falling back to brain_latest.pt")
        elif os.path.exists(latest_path):
            args.resume = latest_path
            print(f"  [AUTO-RESUME] brain_best.pt not found, using brain_latest.pt")
        else:
            import glob as _glob_resume
            brain_files = sorted(
                _glob_resume.glob(os.path.join(args.save_dir, 'brain_step_*.pt')),
                key=lambda p: int(os.path.basename(p).replace('brain_step_', '').replace('.pt', '')),
            )
            if brain_files:
                args.resume = brain_files[-1]
                print(f"  [AUTO-RESUME] Found {len(brain_files)} checkpoints, resuming from: {args.resume}")
            else:
                print(f"  [AUTO-RESUME] No checkpoints in {args.save_dir}, starting fresh.")

    #        Resume from checkpoint

    start_step = 0
    best_loss = float('inf')
    if args.resume:
        print(f"\n  Resuming from: {args.resume}")
        model.load_state(args.resume, strict=True)
        # Resolve companion files (brain_* → autonomous_*, meta_*)
        base = args.resume.replace('brain_best', 'autonomous_best').replace('brain_latest',
             'autonomous_latest').replace('brain_step_', 'autonomous_step_')
        meta_base = args.resume.replace('brain_best', 'meta_best').replace('brain_latest',
             'meta_latest').replace('brain_step_', 'meta_step_')
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        if loop is not None and os.path.exists(base):
            loop.load_autonomous_state(base)
            print(f"  Restored autonomous state: {base}")
        if os.path.exists(meta_base):
            meta = torch.load(meta_base, map_location='cpu', weights_only=False)
            start_step = meta.get('step', 0)
            best_loss = meta.get('best_loss', float('inf'))
            if optimizer is not None and meta.get('optimizer_state_dict') is not None:
                optimizer.load_state_dict(meta['optimizer_state_dict'])
            print(f"  Resumed at step {start_step}, best_loss={best_loss:.4f}")
        print(f"  HPC memories: {model.hippocampus.memory_size}")
        if loop is not None:
            print(f"  Autonomous global_step: {loop.global_step}")

    #        AMP

    use_amp = cfg.use_bf16 and device.type == 'cuda'
    scaler = torch.amp.GradScaler('cuda') if use_amp else None
    autocast_ctx = torch.amp.autocast('cuda', dtype=torch.bfloat16) if use_amp else \
                   torch.amp.autocast('mps', dtype=torch.bfloat16) if (args.bf16 and device.type == 'mps') else \
                   torch.cuda.amp.autocast(enabled=False)

    #        Graceful shutdown handler (Ctrl+C saves brain state)
    _shutdown_requested = [False]  # Mutable container for closure
    def _handle_sigint(signum, frame):
        if _shutdown_requested[0]:
            print("\n  [SIGINT] Force quit.")
            sys.exit(1)
        _shutdown_requested[0] = True
        print("\n  [SIGINT] Graceful shutdown -- saving brain at end of current step...")
    signal.signal(signal.SIGINT, _handle_sigint)

    #        Phase 0: Prenatal Self-Organization
    if args.prenatal > 0 and start_step == 0:
        # [bundled] from nct_v2.v11_autonomous import PrenatalDevelopment
        print(f"\n{'=' * 60}")
        print(f"PHASE 0: PRENATAL SELF-ORGANIZATION")
        print(f"  Steps: {args.prenatal}")
        print(f"  The brain bootstraps itself before seeing any data.")
        print(f"{'=' * 60}\n")

        model.train()
        prenatal = PrenatalDevelopment(model)
        prenatal_start = time.time()

        for ps in range(1, args.prenatal + 1):
            with autocast_ctx:
                p_metrics = prenatal.step(total_prenatal_steps=args.prenatal)

            if ps % max(1, args.prenatal // 20) == 0 or ps == 1:
                phase = p_metrics.get('phase', '?')
                pc_loss = p_metrics.get('pc_loss', 0)
                sigma2 = p_metrics.get('sigma2_mean', 0)
                da = p_metrics.get('da_mean', 0)
                w_mag = p_metrics.get('weight_mag_mean', 0)
                col_diff = p_metrics.get('column_diff', 0)
                gate_evo = p_metrics.get('gate_evolution', 0)
                elapsed = time.time() - prenatal_start
                print(f"  prenatal {ps:>5d}/{args.prenatal} | {phase:17s} | "
                      f"pc {pc_loss:.4f} |  ² {sigma2:.3f} | DA {da:.3f} | "
                      f"w_mag {w_mag:.4f} | col_diff {col_diff:.6f} | gate_evo {gate_evo:.6f} | "
                      f"{elapsed:.0f}s", flush=True)

            # Early convergence check
            if ps >= 100 and prenatal.has_converged():
                print(f"\n  [CONVERGED] Prenatal development converged at step {ps}/{args.prenatal}")
                break

            if _shutdown_requested[0]:
                print(f"\n  [SIGINT] Prenatal interrupted at step {ps}")
                break

        # Print development report
        report = prenatal.get_development_report()
        print(f"\n{report}")

        # Save post-prenatal state
        if args.save_dir:
            os.makedirs(args.save_dir, exist_ok=True)
            prenatal_path = os.path.join(args.save_dir, 'brain_prenatal.pt')
            model.save_state(prenatal_path)
            print(f"  Saved prenatal brain: {prenatal_path}")

        if _shutdown_requested[0]:
            _print_summary(args, model, cfg, loop, prenatal_start, 0, float('inf'), 0)
            return model

        print(f"\nPhase 0 complete. Transitioning to Phase 1: External Data Learning...\n")

    #        Training Loop
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
    print(f"Starting training: {'continuous' if args.continuous else f'{args.steps} steps'}, mode={args.mode}")
    model.train()

    total_tokens = 0
    start_time = time.time()
    step = start_step  # Track for graceful shutdown summary
    _overfit_strikes = 0  # Consecutive overfitting detections
    _overfit_max = 3      # Auto-stop after N consecutive strikes

    # Step iterator: finite or infinite (continuous mode)
    if args.continuous:
        # Unstoppable mode: run forever, self-improving
        step_iter = iter(range(start_step + 1, sys.maxsize))
        print(f"  CONTINUOUS MODE: running indefinitely from step {start_step + 1}")
        print(f"  (Ctrl+C to stop -- brain state auto-saved every {args.save_interval} steps)")
    else:
        step_iter = iter(range(start_step + 1, args.steps + 1))

    for step in step_iter:
        step_start = time.time()

        # Get batch (priority: TextEnvironment > pre-tokenized > random)
        if text_env is not None:
            input_ids, targets = text_env.get_batch()
        elif train_tokens is not None:
            input_ids, targets = get_batch_from_tokens(
                train_tokens, args.batch_size, args.seq_len, device, step
            )
        else:
            input_ids, targets = get_random_batch(
                cfg.vocab_size, args.batch_size, args.seq_len, device
            )

        total_tokens += input_ids.numel()

        #        Training step based on mode

        if args.mode == 'backprop':
            # Standard backprop (baseline comparison -- still a valid mode)
            with autocast_ctx:
                metrics = model.backprop_step(input_ids, targets, optimizer)

        elif args.mode == 'hybrid':
            # AutonomousLoop handles EVERYTHING:
            #   - forward pass
            #   - local learning (PC + Hebbian + neuromod)
            #   - sleep scheduling + dreams + consolidation
            #   - self-play (every N steps)
            #   - EWC estimation (every N steps)
            #   - self-evaluation
            # ALL learning is analytical (.data manipulation) -- same as PC mode.
            # torch.no_grad() saves ~3 GiB by eliminating unused gradient graph.
            with torch.no_grad(), autocast_ctx:
                result = loop.step(input_ids, targets, device=device)

                # NTP-output-only: analytical delta rule on LM head ONLY.
                # Biological: auditory cortex corrects speech production pathway,
                # does NOT propagate errors into cortical association areas.
                # LR scaled 100x: error signal is O(1/V) per token, delta rule is stable.
                # Grad clipping in _local_output_correction prevents divergence.
                if result.get('loss') is not None:
                    _local_output_correction(model, result, lr=args.lr * 100)

            metrics = {
                'loss': result.get('loss', torch.tensor(0.0)).item() if isinstance(result.get('loss'), torch.Tensor) else
                     result.get('loss', 0.0),
                'pc_loss': result.get('pc_loss', torch.tensor(0.0)).item() if isinstance(result.get('pc_loss'),
                     torch.Tensor) else result.get('pc_loss', 0.0),
                'ppl': math.exp(min(result.get('loss', torch.tensor(20.0)).item() if isinstance(result.get('loss'),
                     torch.Tensor) else 20.0, 20)),
                **{k: v for k, v in result.items() if k not in ('loss', 'pc_loss', 'logits', 'metrics')},
                **result.get('metrics', {}),  # Flatten model metrics (sigma2, DA, etc.)
            }

        elif args.mode == 'predictive_coding':
            # Pure local learning via AutonomousLoop -- NO NTP backprop at all.
            # ALL learning is analytical (.data manipulation): PC local_update,
            # Hebbian update, and _local_output_correction (delta rule).
            # torch.no_grad() eliminates the ~200-400 MB gradient graph that was
            # previously built but never used, preventing CUDA OOM on 6 GB GPUs.
            with torch.no_grad(), autocast_ctx:
                result = loop.step(input_ids, targets, device=device)

                # Local output correction: analytical delta rule on LM head ONLY.
                # Biologically: auditory cortex computes heard-vs-intended error
                # and corrects motor→speech weights. Local, no backprop through columns.
                # Without this, PC updates columns but the LM head never learns the
                # representation→vocabulary mapping, causing NTP loss to plateau.
                # LR scaled 100x for output pathway: the error signal is O(1/V) per token
                # so the base lr gives microscopic updates. The delta rule is inherently
                # stable (bounded error), so high LR won't diverge. Matches gate_lr_mult=20.
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
            if result.get('loss') is not None:
                _local_output_correction(model, result, lr=args.lr * 100)

        metrics = {
            'loss': result.get('loss', torch.tensor(0.0)).item() if isinstance(result.get('loss'), torch.Tensor) else result.get('loss', 0.0),
            'pc_loss': result.get('pc_loss', torch.tensor(0.0)).item() if isinstance(result.get('pc_loss'), torch.Tensor) else result.get('pc_loss', 0.0),
            'ppl': math.exp(min(result.get('loss', torch.tensor(20.0)).item() if isinstance(result.get('loss'), torch.Tensor) else 20.0, 20)),
            **{k: v for k, v in result.items() if k not in ('loss', 'pc_loss', 'logits', 'metrics')},
            **result.get('metrics', {}),  # Flatten model metrics (sigma2, DA, etc.)
        }

    elif args.mode == 'hebbian':
        # Pure Hebbian via AutonomousLoop -- no backprop, no PC loss backward
        with torch.no_grad():
            result = loop.step(input_ids, targets, device=device)

        # Local output correction: analytical delta rule on LM head ONLY.
        # Even pure Hebbian mode needs this -- Hebbian updates L2/3 associations
        # but never touches the representation→vocabulary mapping in the LM head.
        # Without this, loss plateaus above random baseline (as seen in prior runs).
        # LR scaled 100x: same rationale as PC mode (error O(1/V), delta rule stable).
        if result.get('loss') is not None:
            _local_output_correction(model, result, lr=args.lr * 100)

        metrics = {
            'loss': result.get('loss', torch.tensor(0.0)).item() if isinstance(result.get('loss'), torch.Tensor) else result.get('loss', 0.0),
            'ppl': math.exp(min(result.get('loss', torch.tensor(20.0)).item() if isinstance(result.get('loss'), torch.Tensor) else 20.0, 20)),
            **{k: v for k, v in result.items() if k not in ('loss', 'pc_loss', 'logits', 'metrics')},
            **result.get('metrics', {}),  # Flatten model metrics (sigma2, DA, etc.)
        }

    #        Wake-Sleep (backprop mode still uses manual sleep)
    if args.mode == 'backprop' and model.should_sleep():
        print(f"\n  [SLEEP] Entering sleep phase (wake steps: {model.learning_manager.wake_sleep.wake_steps})")
        sleep_metrics = model.sleep_phase()
        print(f"  [SLEEP] Replayed {sleep_metrics.get('replayed_memories', 0)} memories")
        print(f"  [SLEEP] Sleep phase #{sleep_metrics.get('sleep_phase', 0)} complete\n")

    #        Logging

    step_time = time.time() - step_start
    loss_val = metrics.get('loss', 0.0)

    # Self-repair: checkpoint best + rollback if diverging
    repair = model.checkpoint_best(loss_val)
    if repair.get('rollback'):
        print(f"\n  [ROLLBACK] Loss diverged: {repair['from_loss']:.4f} → {repair['to_loss']:.4f}")
        print(f"  [ROLLBACK] Count: {repair['rollback_count']}, Sleep: {repair['sleep_interval']}\n")

    if step % args.log_interval == 0 or step == 1:
        elapsed = time.time() - start_time
        tok_per_sec = total_tokens / elapsed if elapsed > 0 else 0

        log_parts = [
            f"step {step:>5d}/{'   ' if args.continuous else args.steps}",
            f"loss {loss_val:.4f}",
            f"ppl {metrics.get('ppl', 0):.1f}",
            f"pc {metrics.get('pc_loss', 0):.4f}",
            f" ² {metrics.get('sigma2_mean', 0):.3f}",
            f"DA {metrics.get('da_mean', 0):.3f}",
            f"tok/s {tok_per_sec:.0f}",
            f"{step_time*1000:.0f}ms",
        ]
        if 'plasticity_mean' in metrics:
            log_parts.append(f"plast {metrics['plasticity_mean']:.3f}")
        if metrics.get('hpc_memory_size', 0) > 0:
            log_parts.append(f"hpc {metrics['hpc_memory_size']}")

        print(" | ".join(log_parts), flush=True)

    #        Generalization Monitor ( ² ratio -- brain's overfitting detector)
    # The model detects its own overfitting through its uncertainty system:
    #    ² ratio (val/train) ~ 1.0 → generalizing well
    #    ² ratio > 2.0 → overconfident on train, surprised by new → overfitting
    # Response: increase sleep frequency (more forgetting = more generalization)
    if val_tokens is not None and step % 500 == 0 and step > 0:
        # Save column outputs -- monitoring forward passes would overwrite them,
        # causing the next training step's Hebbian update to fire on stale data.
        _saved_col_outputs = model._last_column_outputs
        _saved_predictions = model._last_predictions
        _saved_targets = model._last_targets
        _saved_precisions = model._last_precisions
        _saved_neuromod = model._last_neuromod
        _saved_errors = model._last_errors

        model_was_training = model.training
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
            model.eval()  # Prevent HPC encoding during monitoring
            with torch.no_grad(), torch.amp.autocast('cuda', dtype=torch.bfloat16):
                # Train  ²: mean precision on 5 training batches
                train_sigma2_samples = []
                for s in range(5):
                    t_input, _ = get_batch_from_tokens(
                        train_tokens, min(args.batch_size, 8), args.seq_len, device, step + s
                    )
                    model.forward(t_input)
                    if model._last_column_outputs:
                        t_s2 = sum(o.sigma2.mean().item() for o in model._last_column_outputs) /
                             len(model._last_column_outputs)
                        train_sigma2_samples.append(t_s2)

                # Val  ²: mean precision on 5 validation batches
                val_sigma2_samples = []
                for s in range(5):
                    v_input, _ = get_batch_from_tokens(
                        val_tokens, min(args.batch_size, 8), args.seq_len, device, step + s
                    )
                    model.forward(v_input)
                    if model._last_column_outputs:
                        v_s2 = sum(o.sigma2.mean().item() for o in model._last_column_outputs) /
                             len(model._last_column_outputs)
                        val_sigma2_samples.append(v_s2)

            if model_was_training:
                model.train()

            # Restore column outputs so next training step's learning is correct
            model._last_column_outputs = _saved_col_outputs
            model._last_predictions = _saved_predictions
            model._last_targets = _saved_targets
            model._last_precisions = _saved_precisions
            model._last_neuromod = _saved_neuromod
            model._last_errors = _saved_errors

            if train_sigma2_samples and val_sigma2_samples:
                train_s2 = sum(train_sigma2_samples) / len(train_sigma2_samples)
                val_s2 = sum(val_sigma2_samples) / len(val_sigma2_samples)
                s2_ratio = val_s2 / max(train_s2, 0.01)

                status = "HEALTHY"
                if s2_ratio > 2.0:
                    _overfit_strikes += 1
                    status = f"OVERFITTING (strike {_overfit_strikes}/{_overfit_max})"
                    # Brain response: increase sleep frequency (more consolidation)
                    if loop is not None:
                        old_interval = loop.config.sleep_interval
                        loop.config.sleep_interval = max(20, old_interval - 20)
                        status += f" sleep:{old_interval}→{loop.config.sleep_interval}"
                else:
                    _overfit_strikes = max(0, _overfit_strikes - 1)  # Recovery
                    # Restore sleep interval toward default when healthy
                    if loop is not None and _overfit_strikes == 0:
                        default_sleep = cfg.sleep_interval if hasattr(cfg, 'sleep_interval') else 100
                        if loop.config.sleep_interval < default_sleep:
                            loop.config.sleep_interval = min(default_sleep, loop.config.sleep_interval + 10)
                    if s2_ratio < 0.8:
                        status = "UNDERFITTING"

                print(f"  [GENERALIZATION]  ² train={train_s2:.3f} val={val_s2:.3f} "
                      f"ratio={s2_ratio:.2f} → {status}", flush=True)

                # Auto-stop: 3 consecutive overfitting checks = stop training
                if _overfit_strikes >= _overfit_max:
                    print(f"\n  [AUTO-STOP] Sustained overfitting detected "
                          f"({_overfit_max} consecutive checks with  ² ratio > 2.0)")
                    print(f"  Saving final checkpoint and stopping.")
                    if args.save_dir:
                        os.makedirs(args.save_dir, exist_ok=True)
                        model.save_state(os.path.join(args.save_dir, 'brain_latest.pt'))
                        if loop is not None:
                            loop.save_autonomous_state(os.path.join(args.save_dir, 'autonomous_latest.pt'))
                        torch.save({'step': step, 'best_loss': best_loss},
                                   os.path.join(args.save_dir, 'meta_latest.pt'))
                    _shutdown_requested[0] = True

        #        Checkpointing

        if args.save_dir:
            os.makedirs(args.save_dir, exist_ok=True)

            # 1. Best model: save when loss improves (min 100 steps apart to avoid thrashing early on)
            if loss_val < best_loss and (step - getattr(model, '_best_save_step', 0)) >= 100:
                best_loss = loss_val
                model._best_save_step = step
                best_brain = os.path.join(args.save_dir, 'brain_best.pt')
                model.save_state(best_brain)
                if loop is not None:
                    loop.save_autonomous_state(os.path.join(args.save_dir, 'autonomous_best.pt'))
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
                torch.save({'step': step, 'best_loss': best_loss, 'metrics': metrics},
                           os.path.join(args.save_dir, 'meta_best.pt'))
                print(f"  NEW BEST {loss_val:.4f} (PPL {math.exp(min(loss_val, 20)):.1f}) → {best_brain}")
            elif loss_val < best_loss:
                best_loss = loss_val  # Track best loss even if not saving yet

            # 2. Rolling checkpoint: overwrite single file every save_interval
            if step % args.save_interval == 0:
                brain_path = os.path.join(args.save_dir, 'brain_latest.pt')
                model.save_state(brain_path)
                if loop is not None:
                    loop.save_autonomous_state(os.path.join(args.save_dir, 'autonomous_latest.pt'))
                torch.save({
                    'step': step,
                    'optimizer_state_dict': optimizer.state_dict() if optimizer else None,
                    'best_loss': best_loss,
                    'metrics': metrics,
                }, os.path.join(args.save_dir, 'meta_latest.pt'))
                print(f"  Saved checkpoint (step {step}) → {brain_path}")

        #         Graceful shutdown check

        if _shutdown_requested[0]:
            # Save final state to rolling checkpoint before exit
            if args.save_dir:
                os.makedirs(args.save_dir, exist_ok=True)
                brain_path = os.path.join(args.save_dir, 'brain_latest.pt')
                model.save_state(brain_path)
                if loop is not None:
                    loop.save_autonomous_state(os.path.join(args.save_dir, 'autonomous_latest.pt'))
                torch.save({
                    'step': step,
                    'optimizer_state_dict': optimizer.state_dict() if optimizer else None,
                    'best_loss': best_loss,
                    'metrics': metrics,
                }, os.path.join(args.save_dir, 'meta_latest.pt'))
                print(f"  [SHUTDOWN] Brain saved at step {step} → {brain_path}")
            break

    #         Final Summary

    _print_summary(args, model, cfg, loop, start_time, total_tokens, best_loss, step)
    return model


def _print_summary(args, model, cfg, loop, start_time, total_tokens, best_loss, final_step):
    """Print training summary. Factored out so Ctrl+C handler can use it."""
    elapsed = time.time() - start_time
    print(f"\n{'=' * 60}")
    print(f"Training {'interrupted' if args.continuous else 'complete'}!")
    print(f"  Steps:     {final_step}")
    print(f"  Time:      {elapsed:.1f}s ({elapsed/60:.1f}m)")
    print(f"  Best loss: {best_loss:.4f}")
    print(f"  Best PPL:  {math.exp(min(best_loss, 20)):.1f}")
    print(f"  Tokens/s:  {total_tokens / elapsed:.0f}" if elapsed > 0 else "  Tokens/s:  0")
    if loop is not None:
        auto_status = loop.get_status()
        print(f"  Autonomous steps: {auto_status['global_step']}")
        print(f"  Dreams: {auto_status['total_dreams']}")
        print(f"  HPC memories: {model.hippocampus.memory_size if cfg.hpc_enabled else 0}")
    elif cfg.wake_sleep_enabled:
        status = model.learning_manager.get_status()
        print(f"  Sleep phases: {status['sleep_phases']}")
        print(f"  HPC memories: {model.hippocampus.memory_size if cfg.hpc_enabled else 0}")
    print(f"{'=' * 60}")


#
# Smoke Test
#

def smoke_test():
    """Quick validation that the model works."""
    print("=" * 60)
    print("NCT V11 SMOKE TEST")
    print("=" * 60)

    device = torch.device('cuda' if torch.cuda.is_available() else
                          'mps' if torch.backends.mps.is_available() else 'cpu')
    print(f"Device: {device}")

    # Small config for testing
    cfg = V11Config(
        d_model=128,
        d_column=128,
        n_columns=4,
        d_ff=512,
        vocab_size=1000,
        max_seq_len=64,
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
        n_assoc_heads=4,
        d_inhibitory=32,
        nmda_slots=2,
        nmda_d_slot=32,
        silent_d_trace=32,
        phono_capacity=4,
        phono_d_slot=32,
        d_relay=64,
        d_trn=8,
        hpc_d_sparse=256,
        hpc_memory_size=32,
        d_cerebellar=64,
        d_bg=32,
        n_inference_steps=2,
        dropout=0.0,
        learning_mode='hybrid',
        wake_sleep_enabled=True,
        sleep_interval=5,
        sleep_replay_steps=3,
    )

    model = create_v11_model(cfg).to(device)
    params = model.count_parameters()
    print(f"\nModel parameters: {params['total']:,} ({params['trainable']:,} trainable)")

    # Test 1: Forward pass
    print("\n[TEST 1] Forward pass...")
    B, T = 2, 32
    input_ids = torch.randint(0, cfg.vocab_size, (B, T), device=device)
    targets = torch.randint(0, cfg.vocab_size, (B, T), device=device)

    result = model.forward(input_ids, targets)
    assert result['logits'].shape == (B, T, cfg.vocab_size), f"Bad logits shape: {result['logits'].shape}"
    assert 'loss' in result, "No loss in result"
    assert 'pc_loss' in result, "No pc_loss in result"
    assert torch.isfinite(result['loss']), f"Non-finite loss: {result['loss']}"
    print(f"  logits: {result['logits'].shape}")
    print(f"  loss: {result['loss'].item():.4f}")
    print(f"  pc_loss: {result['pc_loss'].item():.4f}")
    print(f"  cb_loss: {result['cb_loss'].item():.4f}")
    print(f"  PASS")

    # Test 2: Local learning step
    print("\n[TEST 2] Local learning step...")
    learn_metrics = model.local_learning_step()
    print(f"  metrics: {learn_metrics}")
    print(f"  PASS")

    # Test 3: Backprop step
    print("\n[TEST 3] Backprop step...")
    optimizer = torch.optim.AdamW(model.get_optimizer_groups(lr=1e-3))
    bp_metrics = model.backprop_step(input_ids, targets, optimizer)
    assert 'loss' in bp_metrics, "No loss in backprop metrics"
    print(f"  loss: {bp_metrics['loss']:.4f}")
    print(f"  ppl: {bp_metrics['ppl']:.1f}")
    print(f"  PASS")

    # Test 4: Wake-sleep cycle
    print("\n[TEST 4] Wake-sleep cycle...")
    sleep_triggered = False
    # Run enough steps to trigger sleep (dev program may increase interval)
    effective_interval = model.learning_manager.development.get_sleep_interval()
    for i in range(effective_interval + 5):
        result = model.forward(input_ids, targets)
        model.local_learning_step()
        if model.should_sleep():
            sleep_metrics = model.sleep_phase()
            print(f"  Sleep triggered at step {i+1} (interval={effective_interval})")
            print(f"  Replayed: {sleep_metrics.get('replayed_memories', 0)} memories")
            sleep_triggered = True
            break
    assert sleep_triggered, f"Sleep never triggered in {effective_interval+5} steps!"
    print(f"  PASS")

    # Test 5: Generation
    print("\n[TEST 5] Generation...")
    prompt = torch.randint(0, cfg.vocab_size, (1, 8), device=device)
    generated = model.generate(prompt, max_new_tokens=16)
    assert generated.shape == (1, 24), f"Bad generated shape: {generated.shape}"
    print(f"  Generated: {generated.shape}")
    print(f"  PASS")

    # Test 6: No NaN/Inf
    print("\n[TEST 6] Numerical stability...")
    for name, param in model.named_parameters():
        assert torch.isfinite(param).all(), f"Non-finite param: {name}"
    print(f"  All parameters finite")
    print(f"  PASS")

    # Test 7: Gate values
    print("\n[TEST 7] Gate values (identity at init check)...")
```

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

```
    gates = {}
    gates.update(model.columns[0].get_gate_values())
    for name, val in gates.items():
        print(f"  {name}: {val:.4f}")
        # All gates should be near sigmoid(-2) ~ 0.1192 at init
        # After a few steps, they may have moved
    print(f"  PASS")

    # Test 8: Autonomous loop
    print("\n[TEST 8] Autonomous loop...")
    # [bundled] from nct_v2.v11_autonomous import AutonomousLoop, AutonomousConfig
    auto_cfg = AutonomousConfig(sleep_interval=5, eval_interval=3, consolidation_interval=100)
    loop = AutonomousLoop(model, auto_cfg)
    for i in range(7):
        input_batch = torch.randint(0, cfg.vocab_size, (2, 32), device=device)
        target_batch = torch.randint(0, cfg.vocab_size, (2, 32), device=device)
        result = loop.step(input_batch, target_batch, device=device)
    status = loop.get_status()
    assert status['global_step'] == 7, f"Expected 7 steps, got {status['global_step']}"
    assert status['total_experiences'] >= 7, f"Expected >=7 experiences, got {status['total_experiences']}"
    print(f"  Steps: {status['global_step']}, Experiences: {status['total_experiences']}")
    print(f"  Dreams: {status['total_dreams']}, Sleep pressure: {status['sleep_pressure']:.3f}")
    print(f"  State: {status['state']}")
    print(f"  PASS")

    # Test 9: learn_from_experience (always-on)
    print("\n[TEST 9] Learn from experience (always-on)...")
    result = model.learn_from_experience(input_ids, targets)
    assert 'logits' in result, "No logits"
    assert 'learning_metrics' in result, "No learning metrics"
    print(f"  Learning metrics: {list(result['learning_metrics'].keys())}")
    print(f"  PASS")
```

Listing 16: Training Script — complete source code (train_v11.py, 1,078 lines)

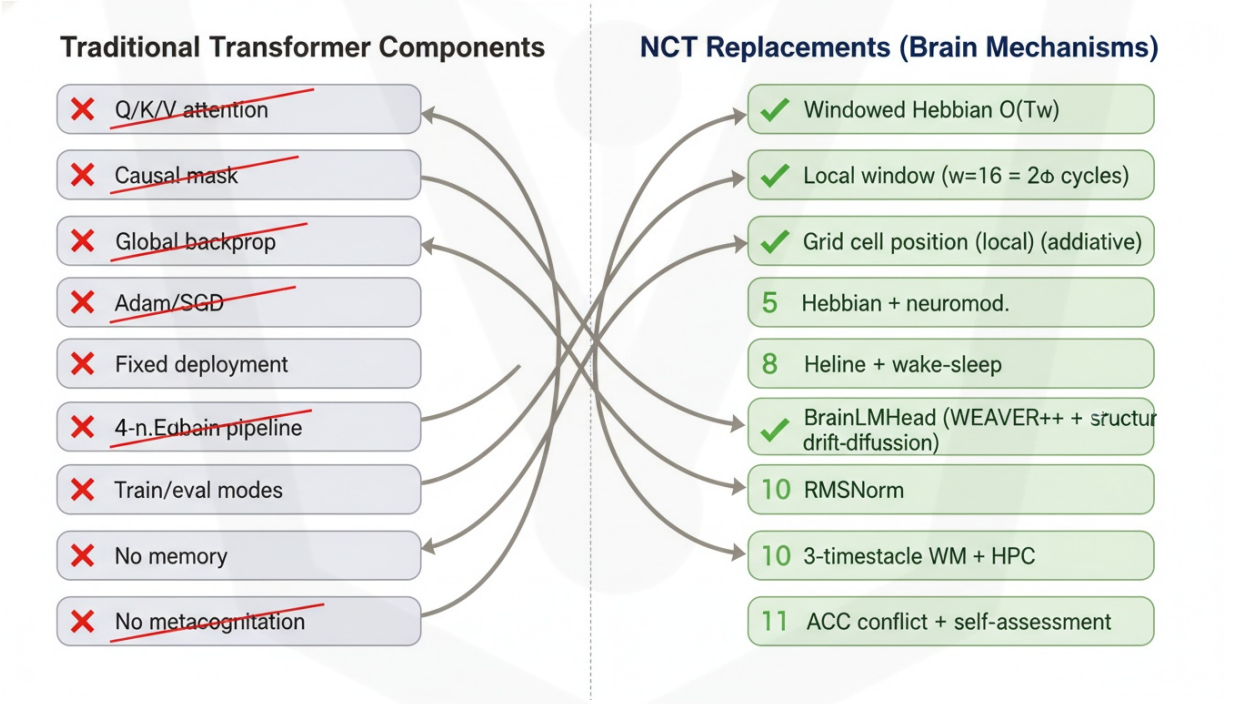


Figure 89: (Illustrated) NCT Cortical Column Network vs. Standard Transformer: a side-by-side comparison of architectural properties, learning mechanisms, and deployment characteristics.

Part III
Theorem Index

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

18 Complete Theorem Mapping

NCT is grounded in 91 theorems + 22 propositions documented in `docs/nct-training-mathematics.md`. The following table maps every theorem referenced in the V11 codebase to its location in this document.

Table 14: Complete theorem-to-section mapping. All 30 theorems referenced in the V11 standalone code.

Thm	Name	V11 Module	Paper Section
Core Theorems (A–E)			
A	Predictive Coding Replaces Backprop	PredictiveCodingOptimizer	section 12
B	Hebbian + Anti-Hebbian = PCA	HebbianOptimizer, L23	section 10, section 12
C	Neuromodulatory RL Without Backprop	Brainstem, NeuromodPlasticity	section 11, section 12
D	Wake-Sleep Self-Improvement	WakeSleepScheduler	section 12
E	Structural Plasticity	CorticalColumn	section 10
Column Internals			
7	BCM Homeostatic Plasticity	BCMHomeostasis	section 10 (BCM subsection)
31	NMDA Attractor Slots	ColumnWorkingMemory	section 10 (WM subsection)
35	Activity-Silent Trace (Tsodyks-Markram)	ColumnWorkingMemory	section 10 (WM subsection)
36	Phonological Loop	ColumnWorkingMemory	section 10 (WM subsection)
38	VIP Disinhibition	ColumnInterneurons	section 10 (Interneurons)
41	PV/Chandelier Output Veto	ColumnInterneurons	section 10 (Interneurons)
42	Metaplasticity	Metaplasticity	section 10 (Metaplasticity)
46	SST Dendritic Inhibition	ColumnInterneurons	section 10 (Interneurons)
53	Oscillatory PAC	ColumnOscillations	section 10 (Oscillations)
Brain Embeddings			
56	Hub-and-Spoke Semantic Decomposition	HubAndSpokeProjection	section 6
57	Sparse Distributed Representations	SparseDistributedProjection	section 6
58	Dual-Route Morphological Composition	MorphemeCompositionGate	section 6
Position Encoding			
59	Grid Cell Learnable Frequencies	GridCellPositionEncoding	section 7
61	Grid Cell Boundary Remapping	GridCellPositionEncoding	section 7
62	Per-Head Period Scales	GridCellPositionEncoding	section 7
LM Head & Decoding			
63	Brain Lexical Production (WEAVER++)	BrainLMHead	section 8
64	Inhibition of Return	InhibitionOfReturn	section 9
65	Confidence-Gated Deliberation	BrainDecoder	section 9
Subcortical			
10	Prospective HPC Head	BrainDecoder (speculative)	section 9
28	DDM Evidence Quality	DriftDiffusionSampler	section 9
39	ACC Conflict Deliberation	SelfAssessment	section 11
47	DA-Modulated Decision Threshold	DriftDiffusionSampler	section 9
48	Model-Derived Noise	DriftDiffusionSampler	section 9
Agentic			
71	BG D1/D2 Tool Selection	V11ToolAdapter	section 16
72	σ^2-Optimal Retrieval	V11RetrievalAdapter	section 16
73	Agent Orchestration	V11ToolAdapter	section 16
V10.2 Infrastructure (in standalone as dependencies, not core V11)			
74	GQA (Grouped Query Attention)	NCTv2Config (legacy)	Not in V11 columns
75	MLA (Multi-Head Latent Attention)	NCTv2Config (legacy)	Not in V11 columns
76	Sliding Window Attention	NCTv2Config (legacy)	Not in V11 columns
77	MoE Expert Routing	NCTv2Config (legacy)	Not in V11 columns

Note: Theorems 74–77 (GQA, MLA, sliding window, MoE) are V10.2 transformer infrastructure. They appear in the standalone file because the legacy `NCTv2Config` class (included as a dependency for brain modules) references them. V11's cortical columns do not use transformer attention — they replace it entirely with Windowed Hebbian Association (Theorem B).

Theorems not in V11: The full math specification (`docs/nct-training-mathematics.md`) contains 91 theorems + 22 propositions + 3 SSM theorems. Of these, theorems 80–91 cover V10.2-

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0

specific features (distillation, PEFT/LoRA, quantization, pruning) that are infrastructure modules rather than core brain mechanisms. Theorems 1–6, 8–9, 11–27, 29–30, 32–34, 37, 40, 43–45, 49–52, 54–55, 60, 66–70, 78–79 cover V10.2 mechanisms that were either incorporated into V11 under different names or superseded by the cortical column architecture.

Part IV
Results and Analysis



19 Training Results (5,000 Steps, 300M Parameters)

Table 15: Main results on WikiText-2 validation.

Metric	Random	NCT (PC)
Val Loss	9.53	**3.99**
Val PPL ↓	13,817	**54.0**
PC Loss	–	6.22
σ^2	1.0 (fixed)	1.88 (adapted)
Backprop through columns	–	**None**
Dream phases	–	150

20 Parameter Distribution

Component	Params	%
Column-local	253.4M	84.3%
Global coordination	9.7M	3.2%
Fixed (embed+head)	25.9M	8.6%
Subcortical	11.6M	3.9%
Total	**300.1M**	**100%**

References

© 2026 Verace Pvt. Ltd. PolyForm Noncommercial License 1.0.0



Verace

NeuroCortex Transformer

Complete Technical Reference

Krrish Choudhary
Founder, Verace
krrish@verace.in

© 2026 Verace Pvt. Ltd. (https://verace.in)

Published as open research under the PolyForm Noncommercial License 1.0.0