# Basic Model

## Contents

**BasicModel** **6**
Basic Model of Cognition . . . 6
Documentation . . . 7
Overview . . . 7
Files . . . 7
Quick Start . . . 7
XML Configuration . . . 8
Output . . . 8

**Installation and Usage** **8**
Prerequisites . . . 8
Setup . . . 9
Makefile Targets . . . 9
Training . . . 9
Models . . . 9
Testing and Benchmarks . . . 9
Documentation . . . 9
Utilities . . . 10
Makefile Variables . . . 10
train.py Options . . . 10
General . . . 10
Remote Execution (SSH) . . . 10
Environment Variables . . . 11
Remote Training (ArborMini) . . . 11
Workflow . . . 11
SSH Key Setup . . . 11
Quick Start . . . 11

**Architecture** **12**
Overview . . . 12
Spaces . . . 12
Reconstruction Symbols . . . 13
Single Optimizer with Overlapping Weight Spaces . . . 14
Training Loop . . . 14
Modes of operation . . . 15
Pipeline as a unit, two-tier reset . . . 15
Three-File Architecture . . . 16
Language System . . . 17
Sigma and Pi Layers . . . 17
Dimensionality Constraints . . . 18
Pi Layer Derivation (Historical) . . . 18
Invertible Linear Layer (LDU) . . . 18

InvertibleLinearLayer – LDU Factorisation . . . 18
Ergodic Noise Injection (Factor-Level) . . . 19
stable=True . . . 19
noise_d . . . 19
naive Flag . . . 19
Noise Lifecycle in Ergodic Mode . . . 20
Class Renames . . . 20
Ergodic Exploration . . . 20

**Basic Model** **20**

**Spaces** **27**
Overview . . . 27
Base Class: Space . . . 28
Normalization and Ranges . . . 29
Codebook Similarity Metric . . . 29
Per-space metric . . . 30
Why Euclidean for Perceptual and Symbolic . . . 30
Why dot product (not Euclidean, not cosine) for Conceptual . . . 30
Configuring the metric . . . 31
Codebook Uniqueness Contract . . . 31
Lexicon (Projective Unit Ball) . . . 32
Distance . . . 32
Matmul-form lookup . . . 33
SBOW training: pode (attractor) and antipode (repulsion target) . . . 33
Why this geometry, not the torus . . . 34
InputSpace . . . 34
PerceptualSpace . . . 35
ConceptualSpace . . . 36
SymbolicSpace . . . 38
Monotonicity of the lift / lower chain . . . 39
SyntacticSpace . . . 40
OutputSpace . . . 41

**Language** **41**
Overview . . . 41
Tiers (P / C / S / L) . . . 42
Grammar . . . 42
Accessors . . . 42
Grammar configuration . . . 43
`data/grammar.cfg` dispatch . . . 43
Syntactic sugar consumption . . . 44
Shamatha speech mode . . . 44
Operator semantics . . . 44
Notation . . . 45
Ternary commitment ops (`true` / `false` / `non`) . . . 45
Negation (`not`) — Sym . . . 45
Conjunction / disjunction — S-tier (post-codebook scalar) . . . 45
Intersection / union — L-tier (lattice on bivectors) . . . 45
Part / equals — Def (currently tensor scoring) . . . 46
Slot selectors (`what` / `where` / `when`) — RETIRED . . . 46
Absorb — base-class marker . . . 46
Query — Mereological . . . 46
Swap (Sinkhorn soft permutation) — Sym . . . 46

Lift / lower — SigmaPi (mode-dispatch ops) . . . 46
Chunking — Percepts . . . 47
Mereological suite — see Mereology.md . . . 47
GrammarLayer Implementations . . . 47
Base contract . . . 47
`NotLayer` — `S = not(S)` . . . 48
`NonLayer` — `S = non(S)` . . . 48
`IntersectionLayer` — L-tier lattice min on bivector activation . . . 48
`UnionLayer` — L-tier lattice max on bivector activation . . . 49
`ConjunctionLayer` / `DisjunctionLayer` — S-tier monotonic on scalar activation . . . 49
`LiftLayer` / `LowerLayer` — S-tier gated SVD slice on muxed event . . . 50
`EqualsLayer` / `PartLayer` — mereology . . . 50
`TrueLayer` / `FalseLayer` — pole projection . . . 51
`SwapLayer` — `S = swap(S, S)` . . . 51
`QueryLayer` — `S = query(S, S)` . . . 52
Retired: `WhatLayer` / `WhereLayer` / `WhenLayer` / `AbsorbLayer` . . . 53
`PiLayer` — parametrized AND fold . . . 53
`SigmaLayer` — parametrized OR fold . . . 54
Class registry . . . 55
Inverse contract by operator . . . 55
Rule prediction . . . 56
Composition . . . 57
Phase 1: deterministic `not` (legacy 2D path) . . . 57
Phase 2: chart inside pass . . . 57
Tensor word buffer . . . 57
Chart-derived SVO . . . 58
POS side-channel . . . 58
Storage tiers . . . 58
Lexical bootstrap . . . 58
Mechanism 1 — RHS POS compatibility mask . . . 58
Mechanism 3 — Rule-prediction conditioning . . . 58
Why this trains the network to assign POS to every word . . . 59
`<writeSyntax>true</writeSyntax>` — syntax-tree dump . . . 59
Per-space dispatch (`SyntacticLayer`) . . . 60
`WordSpace.forwardSymbols` / `reverseSymbols` . . . 60
Downward head emission (`S -> C`) . . . 60
Projection-coefficient scoring . . . 60
`WordSpace.reconstruct(state, codebook_space)` . . . 60
Word encoding . . . 61
Decomposition . . . 61
How syntax is trained . . . 61
Two-Layer logic . . . 61
SymbolicSpace integration . . . 62
Forward path . . . 62
Reverse path . . . 62
Key properties . . . 62
AssociationLayer (EQUALS implementation) . . . 62
Future: RuleNode on the ReconstructionStack . . . 62
Design history: learned SVO plan . . . 63
Historical starting point (pre-chart) . . . 63
Phases summary . . . 63
Stable hooks already wired . . . 64

**Mereology** **64**

Parthood as Clipped Cosine Projection . . . 64
The Full Suite . . . 64
Region Partition . . . 65
Mereological Fusion . . . 65
ImpenetrableLayer: Five-Relations Regularizer . . . 66
Penalty . . . 66
Trust . . . 66
Diagnostics . . . 66
Configuration . . . 66
Why This Design . . . 67

**Logic** **67**
Overview . . . 67
1. Subsymbolic Layer . . . 67
Operators . . . 67
2. Symbolization . . . 68
3. Symbolic Layer (Scalars in [-1,1]) . . . 68
Negation (affirming) . . . 68
Non (non-affirming) . . . 68
Union (monotonic) . . . 68
Intersection (monotonic) . . . 68
Parthood as Projection . . . 69
4. Key Insight . . . 69
5. Rationality . . . 70
Truth Statements . . . 70
TruthLayer (WordSpace.truth_layer) . . . 70
Truth Field . . . 70
Consonance and Dissonance . . . 71
Propositional Structure . . . 71
Truth Accumulation Pipeline . . . 71
6. Consistency and Verification . . . 71
Internal Consistency (within a TruthSet) . . . 71
Verification (incoming statement against stored truth) . . . 72
Logical Entailment (augmenting geometry with symbolic closure) . . . 72
7. . . . 73

**Radial Operators for Hypersphere Ternary Logic** **73**
Overview . . . 73
Truth Values . . . 73
Magnitude Ordering . . . 73
Operators . . . 74
RadMin (Radial Minimum – Conjunction / AND) . . . 74
RadMax (Radial Maximum – Disjunction / OR) . . . 74
NOT (Sign-Flipping Negation) . . . 75
NON (Non-Affirming Negation) . . . 75
Luminosity . . . 76
Fusion . . . 76
Supporting measures . . . 77
Semantic Summary . . . 77
Open Questions . . . 77
8. Cross-references . . . 78

**Reasoning System** **78**
Partitioned Symbolic Space . . . 78

Reasoning Methods . . . 78
`isConsistent() -> dict` . . . 78
`ground(activation, threshold=0.6) -> dict` . . . 78
`isTrue(activation) -> float` . . . 79
`extrapolate(grammar, seed_indices, max_new, attenuation) -> dict` . . . 79
TruthLoss . . . 79
Bidirectional Reasoning Loop . . . 79
Grammar Learning . . . 79
Architecture Modes: useButterflies × useGrammar . . . 80
Flat (both false) . . . 80
Butterfly (useButterflies=true, useGrammar="none") . . . 80
Grammar-directed (useButterflies=false, useGrammar="all") . . . 81
Configuration . . . 81
Contemplative Awareness Methods . . . 81
Testing . . . 82

**Training** **82**
Overview . . . 82
Phase 1: Embedding Pretraining (`make train` / `embed.py train`) . . . 82
Pipeline . . . 82
SBOW (Sentence Bag of Words) . . . 82
CBOW (Continuous Bag of Words) . . . 83
Two-Pass Architecture . . . 83
Configuration (Makefile variables) . . . 83
Memory Considerations . . . 84
Phase 2: Network Training (`BasicModel.py`) . . . 84
Masked Prediction Modes . . . 84
`<trainEmbedding>`: Embedding Update Mode . . . 84
Gradient Flow Through Codebook . . . 85
`JOINT`: Single Combined Loss . . . 85
Why MSE over Embeddings (not Cross-Entropy over Vocabulary) . . . 86
Training Loop . . . 86
SBOW vs CBOW . . . 87
Three-File Architecture . . . 87
Embedding Space Geometry . . . 87
Profiling . . . 88

**Ergodic Exploration via Adaptive Bias-Variance Control** **88**
1. The Ergodic Weights . . . 89
2. Why Standard Adam Fails on $\alpha$ . . . 89
3. The Gradient Energy Sensor . . . 90
3.1 Modified Second-Moment Estimator . . . 90
3.2 Bias Correction . . . 90
3.3 What $\hat{v}_t$ Measures . . . 90
3.4 The Alpha Update Rule . . . 90
3.5 Layer-Local Certainty (Per-Neuron Sigma) . . . 91
4. Comparison: Adam vs. Gradient Energy Sensor . . . 91
5. Derivation: Why Zero-Mean Implies Drop $m_t$ . . . 91
6. Dropout via Bias . . . 92
7. The `global_temp` Sensitivity Knob . . . 92
8. Initialization and Bootstrap . . . 92
9. Layer Architecture . . . 93
ErgodicLayer Base Class . . . 93
InvertibleLinearLayer (LDU factorisation) . . . 93

10. Training Loop Integration . . . 94
11. Summary of the Full Algorithm . . . 94
Factor-Level Noise Injection . . . 94
Previous Approach: Matrix-Level Blending . . . 94
New Approach: Factor-Level Injection . . . 95
stable=True and the d Backbone . . . 95
SigmaLayer and PiLayer Integration . . . 95
Abstract . . . 95
Background . . . 96
Part I: What Machine Minds Are – Weight Ergodicity . . . 96
Weights as Ergodic Samples . . . 96
Temperature and Annealing . . . 97
Advantages of Ergodic Weights . . . 97
Part II: What Machine Minds Feel – Network Invertibility . . . 97
Bidirectional Perception . . . 97
Symbols, Worldlines, and Karmic Inertia . . . 97
Implementation . . . 98
Part III: What Machine Minds Know – Output Certainty . . . 98
Certainty vs. Probability . . . 98
Certainty-Weighted Loss . . . 98
Knowing What You Do Not Know . . . 99
Methods . . . 99
Results . . . 100
Discussion . . . 101
Conclusion . . . 101
References . . . 101

**XML Configuration Parameters** **101**
`<architecture>` . . . 102
`<architecture><data>` . . . 102
`<architecture><training>` . . . 102
BasicModel Configuration . . . 103
`<InputSpace>` . . . 104
`<PerceptualSpace>` . . . 104
`<ConceptualSpace>` . . . 105
`<SymbolicSpace>` . . . 105
`<SyntacticSpace>` . . . 105
`<OutputSpace>` . . . 106
`InvertibleLinearLayer` . . . 106
`<architecture><server>` . . . 106
Example: XOR Configuration . . . 107
Example: Embedding / Chatbot Configuration . . . 108

# BasicModel

## Basic Model of Cognition

*The basic model of cognition relies on conceptual hyperplanes and perceptual prototypes to synthesize and analyze the input space. It uses a high-dimensional embedding to characterize mental space and integrates symbolic computation. Its three major operations are intersection (which forms percepts from concepts), union (a bidirectional mapping between concepts and percepts), and equality (where symbols are elements that map across perceptual and conceptual domains).*

## Documentation

| Document | Description |
|---|---|
| Architecture | Pipeline design, layer types, invertible LDU factorisation |
| Spaces | All six spaces: Input → Perceptual → Conceptual → Symbolic → Syntactic → Output |
| Ergodic | Gradient energy sensor, adaptive exploration, factor-level noise injection |
| Training | Two-phase training, SBOW embeddings, masked prediction modes |
| Params | Complete XML configuration reference |
| BasicModel | Cognitive science foundations |
| Language | Grammar, word encoding, symbolic and syntactic spaces |
| Logic | Subsymbolic and symbolic logic operations |
| Reasoning | Truth methods, bidirectional reasoning, contemplative awareness stubs |
| MachineMinds | What machine minds are, feel, and know |
| Installation | Setup, Makefile targets, environment variables |

## Overview

BasicModel is a parameterized neural architecture that answers the question "what is the truest thing we can say of this experience?".

Model configurations are specified in XML. See doc/Architecture.md for the full mathematical treatment.

## Files

| File | Description |
|---|---|
| bin/BasicModel.py | Main entry point: model factory, training loop, comparison plots, HTML report |
| bin/Model.py | Layer library: SigmaLayer, PiLayer, ErgodicLayer, LinearLayer, TruthLayer |
| bin/Space.py | Space classes: InputSpace, PerceptualSpace, ConceptualSpace, SymbolicSpace, OutputSpace, Gram |
| bin/embed.py | Word vector training: CBOW/SBOW with negative sampling, `WordVectors` (gensim-compatible `.k` |
| bin/SigmaPi.py | Standalone demo of the SigmaPi network solving XOR |
| data/ | XML model configurations |
| doc/Architecture.md | Algorithm details: Sigma/Pi layers, ergodic exploration, gradient energy sensor |
| doc/Params.md | Full XML parameter reference |
| doc/Training.md | Embedding pretraining, CBOW/SBOW, masked prediction, `<trainEmbedding>` modes |
| doc/Installation.md | Setup, Makefile targets, train.py options, remote training |
| test/ | Unit tests |

## Quick Start

```
# Set up virtual environment
make install

# Run a single model
make simple          # data/simple.xml
make ergodic         # data/ergodic.xml

# Compare two models side-by-side
make compare         # defaults: data/simple.xml vs data/ergodic-only.xml
make compare XML1=data/simple.xml XML2=data/ergodic.xml

# Run tests
make test
```

```
# Generate PDF documentation
make doc_pdf
```

### XML Configuration

Models are configured via XML files in `data/`. Training and data parameters live in nested sub-elements:

```
<model>
  <architecture>
    <modelType>simple</modelType>   <!-- simple | lm | embedding -->
    <ergodic>false</ergodic>
    <certainty>false</certainty>
    <reconstruct>NONE</reconstruct>  <!-- NONE | symbols | output | both -->
    <maskedPrediction>NONE</maskedPrediction>

    <data>
      <dataset>xor</dataset>        <!-- xor | mnist | text | ... -->
    </data>

    <training>
      <numTrials>1</numTrials>
      <numEpochs>20</numEpochs>
      <batchSize>10</batchSize>
      <learningRate>0.001</learningRate>
      <weightsPath>output/BasicModel.ckpt</weightsPath>
      <autoload>true</autoload>
      <autosave>false</autosave>
    </training>
  </architecture>

  <InputSpace> ... </InputSpace>
  <PerceptualSpace> ... </PerceptualSpace>
  <ConceptualSpace> ... </ConceptualSpace>
  <SymbolicSpace> ... </SymbolicSpace>
  <OutputSpace> ... </OutputSpace>
</model>
```

See doc/Params.md for the full parameter reference.

### Output

Each run produces an HTML report (timestamped in `output/`) containing:

- **Error per Epoch** – training and test loss curves
- **Accuracy per Digit** – per-class accuracy breakdown

In compare mode, additional overlay plots show combined loss and accuracy across models with color-coded legends.

## Installation and Usage

### Prerequisites

- **Python 3.12** with a virtual environment at `.venv/`
- **PyTorch** with MPS support (Apple Silicon) or CUDA

- **pandoc** (optional, for PDF generation via `make doc_pdf`)

## Setup

```
make install   # creates .venv with dependencies
```

This installs all Python dependencies (including PyTorch) into a project-local virtual environment. All Makefile targets invoke Python through `.venv/bin/python`, so there is no need to activate the environment manually.

---

## Makefile Targets

### Training

| Target | Description |
|---|---|
| `make train` | Full training (Phase 1 embeddings + Phase 2 model), logged to `output/logs/` |
| `make train_micro` | Small run: 500 docs, 1 random shard, logged |
| `make train_remote` | Full training on `arbormini.local` via SSH |
| `make train_micro_remote` | Micro training on `arbormini.local` via SSH |

### Models

| Target | Description |
|---|---|
| `make run` | Run `BasicModel.py` with the config specified by `XML1` |
| `make xor` | Run with `data/XOR_exact.xml` |
| `make simple` | Run with `data/simple.xml` |
| `make ergodic` | Run with `data/ergodic.xml` |
| `make tomatoes` | Run with `data/tomatoes.xml` |
| `make mnist` | Run with `data/mnist.xml` |
| `make compare` | Compare two models side-by-side using `XML1` and `XML2` |
| `make SigmaPi` | Run `SigmaPi.py` |
| `make SymPercept` | Run `SymPercept.py` |
| `make SPNN` | Run `SPNN.py` |

### Testing and Benchmarks

| Target | Description |
|---|---|
| `make test` | Run unit tests (forces `BASICMODEL_DEVICE=cpu`) |
| `make bench` | Run training benchmarks (baseline, no env tweaks) |

### Documentation

| Target | Description |
|---|---|
| `make doc_pdf` | Generate `BasicModel.pdf` from doc chapters via pandoc |

The PDF is assembled from the following chapters in order: `README.md`, `doc/Architecture.md`, `doc/BasicModel.md`, `doc/Spaces.md`, `doc/Language.md`, `doc/Logic.md`, `doc/Ergodic.md`, `doc/Training.md`, `doc/MachineMinds.md`, `doc/Params.md`, `doc/Installation.md`.

### Utilities

| Target | Description |
|---|---|
| `make clean` | Remove generated files (`BasicModel.pdf`) |
| `make all` | Default target; alias for `make xor` |

---

## Makefile Variables

All variables can be overridden on the command line, e.g. `make run XML1=data/ergodic.xml`.

| Variable | Default | Description |
|---|---|---|
| `MODEL` | `data/BasicModel.xml` | XML config file for training targets |
| `XML1` | `data/simple.xml` | Primary config for `make run` and `make compare` |
| `XML2` | `data/ergodic-only.xml` | Secondary config for `make compare` |
| `TRAIN_HOST` | *(empty)* | Remote training host; set automatically by `train_remote` targets |
| `TRAIN_USER` | `arogers` | SSH user for remote training |
| `TRAIN_KEY` | `~/.ssh/id_ed25519_arbormini` | SSH private key file |
| `TRAIN_DIR` | `~/WikiOracle/basicmodel` | Working directory on the remote host |

---

## train.py Options

`train.py` orchestrates the full training pipeline: Phase 1 (embeddings) followed by Phase 2 (model training).

### General

| Flag | Default | Description |
|---|---|---|
| `--model`, `-m` | `data/BasicModel.xml` | XML config file |
| `--max-docs` | *(from XML config)* | Override `maxDocs` – limits the number of Wikipedia documents processed |
| `--num-shards` | *(from XML config)* | Override `numShards` – number of embedding shards to train |
| `--random-shards` | off | Pick random shard indices instead of sequential, for variety across runs |
| `--force-embeddings` | off | Retrain embeddings even if the output file already exists |
| `--log [FILE]` | off | Capture stdout/stderr to a `.log` file in `output/logs/`. Omit the filename |
| `--profile` | off | Run Phase 2 under `cProfile`; writes a `.prof` file to `output/profiles/` a |

### Remote Execution (SSH)

| Flag | Default | Description |
|---|---|---|
| `--host` | *(none)* | Remote host (e.g. `arbormini.local`). When set, training runs remotel |
| `--user` | `arogers` | SSH user |
| `--key-file` | `~/.ssh/id_ed25519_arbormini` | SSH private key |
| `--remote-dir` | `~/WikiOracle/basicmodel` | Working directory on the remote machine |

---

## Environment Variables

| Variable | Description |
|---|---|
| `BASIC_MAX_DOCS` | Override `maxDocs` in `BasicModel.py` (set automatically by `train.py` when `--max-` |
| `BASIC_NUM_SHARDS` | Override `numShards` in `BasicModel.py` (set automatically by `train.py` when `--nu` |
| `BASICMODEL_DEVICE` | Force a specific device, e.g. `cpu`. Used by `make test` to avoid GPU-dependent fail |
| `PYTORCH_MPS_HIGH_WATERMARK_RATIO` | MPS memory allocation limit. Set to `0.0` by `train.py` to disable the high-waterm |
| `PYTHONUNBUFFERED` | Disable Python output buffering. Set to `1` by `train.py` so log output streams in re |

---

## Remote Training (ArborMini)

The `train_remote` and `train_micro_remote` targets automate the full remote-training workflow. Under the hood they set `TRAIN_HOST=arbormini.local` and delegate to `train.py --host`.

### Workflow

1. **Rsync** syncs the local project tree to `arbormini.local`:

   ```
   rsync -av --progress \
     -e "ssh -i ~/.ssh/id_ed25519_arbormini" \
     --exclude .venv \
     --exclude __pycache__ \
     --exclude .DS_Store \
     --exclude output/ \
     ./  arogers@arbormini.local:~/WikiOracle/basicmodel/
   ```

   The exclusions keep the transfer fast: `.venv/` (the remote has its own), `__pycache__/` (bytecode), `.DS_Store` (macOS metadata), and `output/` (embeddings, logs, profiles).

2. **SSH** connects to ArborMini and runs `train.py` with the same flags that were passed locally (minus `--host`):

   ```
   ssh -i ~/.ssh/id_ed25519_arbormini -t arogers@arbormini.local \
     "cd ~/WikiOracle/basicmodel && PYTHONUNBUFFERED=1 PYTHONPATH=bin .venv/bin/python bin/train.py --
   ```

3. **Output stays remote.** Profile data (`.prof` files) and training logs remain on ArborMini under `~/WikiOracle/basicmodel/output/`.

### SSH Key Setup

Remote targets require passwordless SSH access. Generate and copy a key if you have not already:

```
ssh-keygen -t ed25519 -f ~/.ssh/id_ed25519_arbormini
ssh-copy-id -i ~/.ssh/id_ed25519_arbormini arogers@arbormini.local
```

### Quick Start

```
# Fast sanity check: 500 docs, 1 shard, on ArborMini
make train_micro_remote

# Full training run on ArborMini
make train_remote
```

# Architecture

**Stage 2 in progress.** A wide ConceptualSpace codebook (nVectors ≫ nOutput), a shared `SparsityRegLayer` attached to Perceptual / Conceptual / Symbolic spaces, a three-way PerceptualSpace chunking switch (`raw | bpe | lexicon`), and a tri-state `useGrammar` config (`all | thoughtFree | none`) are landing progressively. Design: `specs/2026-04-16-stage2-wide-conceptual-codebook-design.md`. Plan: `plans/2026-04-16-stage2-wide-conce` Phases 1-5.1/5.2 are in; Task 5.3 (merged mode-blind outer forward loop) is deferred – WordSpace's actual shape is a per-tier dispatcher rather than a single `forward(ss)`, so the outer-loop refactor needs a design addendum before implementation.

## Overview

BasicModel is a bidirectional neural architecture organized as a pipeline of six **spaces**, each implementing a distinct representational transformation:

```
Forward:  InputSpace -> PerceptualSpace -> ConceptualSpace -> SymbolicSpace -> SyntacticSpace -> OutputS
Reverse:  OutputSpace -> SyntacticSpace -> SymbolicSpace -> ConceptualSpace -> PerceptualSpace -> InputS
```

The forward pass transforms raw input into predictions. The reverse pass reconstructs the original input from the symbolic representation. Both directions are trained simultaneously with a single optimizer minimizing a combined loss:

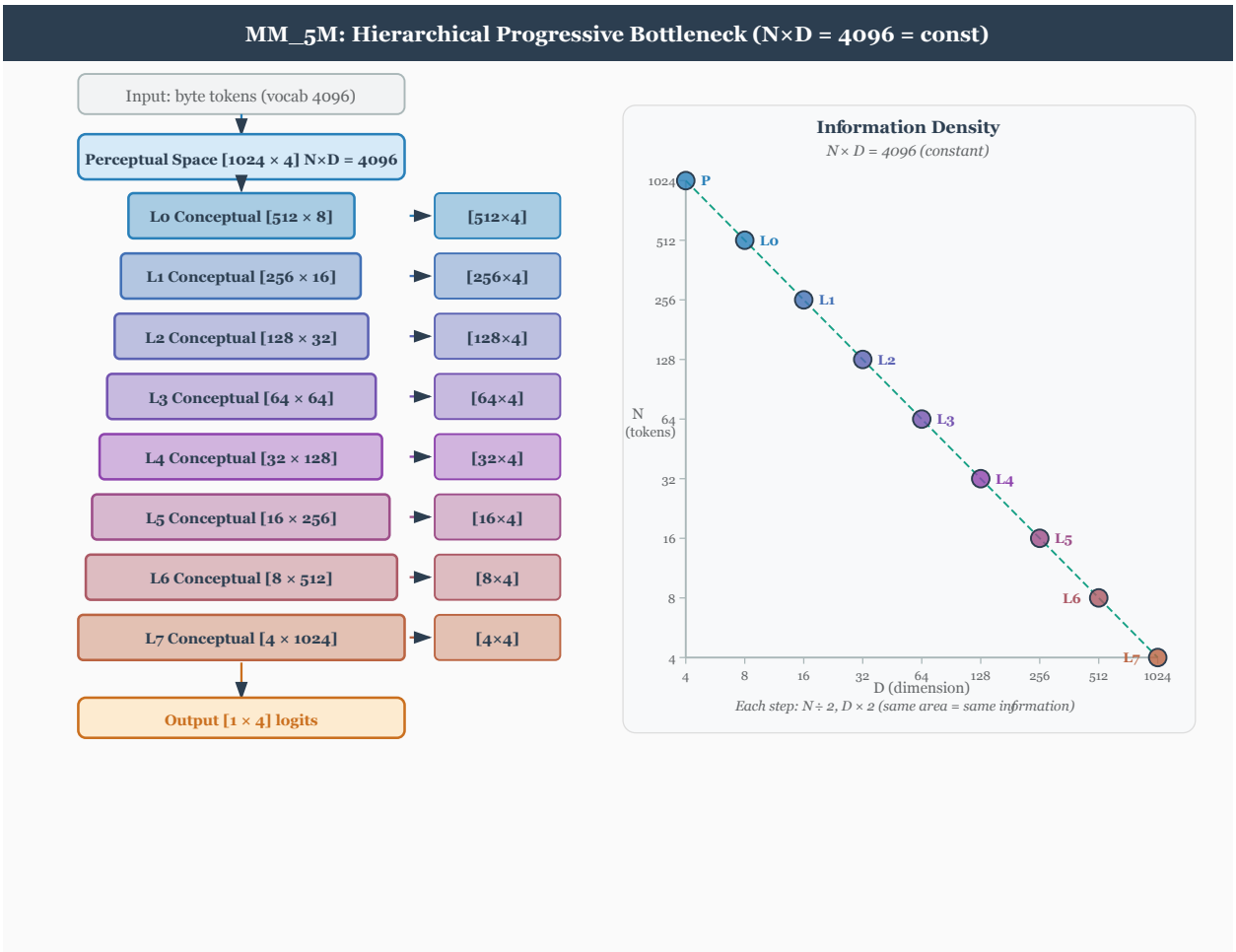
```
totalLoss = (1 - reconRatio) * outputLoss + reconRatio * reconstructionLoss
```

### Spaces

| Space | Role | Layer Type |
|---|---|---|
| **InputSpace** | Lifts raw data into working dimensionality | LiftingLayer |
| **PerceptualSpace** | Per-percept aggregation | SigmaLayer (`self.sigma`, P -> sub-percept; d |
| **ConceptualSpace** | Multiplicative abstraction (P -> C) | PiLayer (`self.pi`, P -> C, with `forwardPi` / |
| **SymbolicSpace** | Discrete activation bottleneck (C -> S) + Codebook | SigmaLayer (`self.sigma`, C -> S, with `forwa` |
| **SyntacticSpace** | Binary derivation tree (CNF grammar) | Grammar + WordEncoding |
| **OutputSpace** | Final prediction | LinearLayer |

Each level-crossing space owns one bidirectional layer that handles both forward and reverse via its own self-inverse: `ConceptualSpace.pi` does P->C forward and C->P reverse; `SymbolicSpace.sigma` does C->S forward and S->C reverse. Pi and Sigma have independent weights – there is no shared layer across the boundary. See doc/Logic.md §8 and doc/Spaces.md for the full framing.

Dimensions (`nDim`) are not passed to Space constructors; each subclass reads its content dimensionality from `TheObjectEncoding` (e.g., `TheObjectEncoding.perceptDim` for PerceptualSpace). Codebook sizes (`nVectors`) are likewise stored on `TheObjectEncoding` and may differ from the active count (`nActive`); the factory validates `nVectors >= nActive` for every space.




Layer selection depends on `<reconstruct>` and `invertible`:

1. **`reconstruct=NONE`**: Use non-invertible layers (`PiLayer`, `SigmaLayer`) for the forward pass only. No reverse pipeline is created.
2. **`reconstruct=<any> + invertible`**: A single invertible layer (`PiLayer(invertible=True)`, `SigmaLayer(invertible=True)`) serves both directions, sharing weights.
3. **`reconstruct=<any>` + not `invertible`**: Two layers with separate weights – call `forward()` on one and `reverse()` on the other. This avoids the expressivity limitation where a non-invertible layer's forward pass cannot represent the inverse of another (e.g., PiLayer's product structure is not closed under inversion). **Note:** The reverse path uses a matrix pseudoinverse (`pinv`) which may be numerically unstable due to SVD convergence issues. Setting `<invertible>true</invertible>` avoids this by using shared-weight inversion instead.

### Reconstruction Symbols

The symbolic bottleneck can lose information needed for reconstruction. For example, XOR maps 2 inputs to 1 output, but `XOR(0,0)=0` and `XOR(1,1)=0` are distinct inputs producing the same output. A single output symbol cannot reconstruct which input was presented.

To solve this, the `nSymbols` produced by SymbolicSpace are split:

- **`nOutputSymbols = OutputSpace.nActive`** – fed to OutputSpace for prediction
- **`nReconSymbols = nSymbols - nOutputSymbols`** – carried in `end_state` for reconstruction

This is essentially a skip connection through the symbolic bottleneck: an extra channel that preserves

information lost in the output projection. Reconstruction symbols receive gradient only from reconstruction loss, never from output loss.

For XOR with `nSymbols=3`, `nOutput=1`: symbol 0 predicts the output; symbols 1-2 carry enough information to distinguish all 4 input patterns, enabling perfect reconstruction.

### Single Optimizer with Overlapping Weight Spaces

A critical design choice: the forward and reverse passes share a **single Adam optimizer** that minimizes a combined loss:

```
totalLoss = (1 - reconRatio) * outputLoss + reconRatio * reconstructionLoss
```

This works because the forward and reverse weight spaces **partially overlap**. They are neither disjoint (which would allow independent optimizers) nor identical (which would create destructive interference). Instead, certain layers share weights between directions (e.g., shared embeddings, the symbolic bottleneck itself) while others are direction-specific (e.g., `pi1`/`pi2`, `sigma1`/`sigma2`, `linear1`/`linear2`).

The partial overlap means:

- **Shared weights** receive gradient from *both* output and reconstruction loss, learning representations useful in both directions simultaneously.
- **Direction-specific weights** receive gradient from only their respective loss, specializing freely without interference.
- **No ping-pong**: separate optimizers on overlapping parameters would pull weights in alternating, conflicting directions each step. A single optimizer sees the combined gradient and finds a consistent descent direction.

This resolves the fundamental tension between forward prediction and backward reconstruction in bidirectional architectures – a problem first identified in A.M. Rogers, T.T. Shannon, and G.G. Lendaris, "A comparison of DHP based antecedent parameter tuning strategies for fuzzy control," *Proceedings Joint 9th IFSA World Congress and 20th NAFIPS International Conference*, Vancouver, BC, Canada, 2001, pp. 580-585, doi: 10.1109/NAFIPS.2001.944317. See also `doc/research/` for a local copy.

When `invertible=true`, the overlap is total: a single invertible layer serves both directions, and its weights receive the full combined gradient. The single-optimizer approach handles this case naturally.

### Training Loop

Training uses the single Adam optimizer with persistent state (momentum/variance accumulate across epochs):

1. Forward pass: input → prediction + `end_state`
2. Compute `outputLoss` from prediction vs. target
3. Reverse pass: `end_state` → reconstructed input
4. Compute `reconstructionLoss` from reconstruction vs. original input
5. Backpropagate `totalLoss = (1 - reconRatio) * outputLoss + reconRatio * reconstructionLoss`
6. If ergodic: run `paramUpdate()` (gradient energy sensor updates alpha)
7. Optimizer step (embedding params excluded when `trainEmbedding` is `NONE`, `CBOW`, or `SBOW`)
8. If `trainEmbedding` is `CBOW`, `SBOW`, or `BOTH`: run embedding update step on same sentence

Alpha annealing (ergodic mode): the code-level exploration parameter starts at `1.0` (full exploration) and decays to `0.0` (full exploitation) within the first 5% of epochs via `alpha = max(0, 1 - epoch / warmup)` where `warmup = numEpochs // 20`. Note: the code convention (`alpha=1` means explore) is the inverse of the ergodic math convention (`alpha=0` means explore); layers translate between them internally.

When `<useButterflies>true</useButterflies>` (and `conceptualOrder > 1`), the Sigma-Pi loop switches from a flat concatenation-based cycle to a pairwise butterfly architecture with per-level butterfly-mode Pi/Sigma layers and a geometrically partitioned symbol dimension. `<useGrammar>true</useGrammar>` (legacy spelling for `useGrammar="all"` in `<WordSpace>`) selects a grammar-directed progressive-bottleneck

variant instead. Full grammar mode and butterflies are mutually exclusive – butterfly permutations fight constituency structure. The planned `shamathaSpeech` mode is a narrow DNF-object grammar with contiguity checks, not the full constituency path. See Reasoning.md Section Architecture Modes for the full comparison.

See Params.md for all XML configuration parameters. See Training.md for embedding pretraining, SBOW, masked prediction modes, and the `<trainEmbedding>` control.

### Modes of operation

The architecture has three orthogonal mode dimensions plus a subsymbolic-enable flag, set independently in the model XML.

**Butterfly mode** (`<useButterflies>true</useButterflies>`):

Pairwise sigma/pi with N-halving across `<conceptualOrder>` stages. Each per-stage `ConceptualSpace` / `SymbolicSpace` instance is built with a butterfly-mode SigmaLayer / PiLayer that packs `[B, N, D]` to `[B, N/2, 2*D]` before the inner LDU and unpacks after. Slot count halves per stage; per-slot dim doubles; total per-stage volume preserved. Validated by `nPercepts $\times$ state_dim == nSymbols $\times$ symbol_width` at `ModelFactory.validate_config`. Used by MM_xor, MM_5M, MM_400M. Without butterflies, every stage shares the same shape and no halving happens (the "plain" / `useGrammar=all` paths). Reference: `bin/Models.py _build_staged_pipeline` butterfly branch; `bin/Layers.py ButterflyLayer._butterfly_pack` / `_butterfly_unpack` / `_butterfly_merge` / `_butterfly_unmerge`.

**Serial mode** (`BASICMODEL_DEVICE` / runtime flag `serial_mode`):

A runtime fast path for streaming / autoregressive contexts. `PerceptualSpace` / `ConceptualSpace` may use the slide-and-recompute path where the previous step's per-cell warm cache (kept on `subspace.serial_cache`) is reused for cells that haven't shifted. Independent of butterfly / parallel mode – gated only by the runtime flag. The cache is keyed on the owner-Space id, cleared on hard `Reset`, rebuilt cheaply on the next forward. Reference: `Space.serial_mode` flag (set by `BaseModel.create_from_config`), `SubSpace.serial_cache` dict, and the `test_serial_mode_*.py` tests.

**Conceptual input sourcing** (per-order switch; the legacy loopback concat was retired 2026-05-09):

Each stage's `ConceptualSpace.forward` reads exactly one input source per stage. Stage 0 reads `PerceptualSpace.event` directly (full content width). Stages $t \geq 1$ read the active sibling's previous event – `SymbolicSpace.event` under the default `grammar` mode, `SubsymbolicSpace.event` under `<architecture><mode>parallel</mode>` – lifted from the bivector handoff $[B, V_S, 2]$ back to concept content $[B, V, \text{concept_dim}]$ via the C-tier codebook's SVD pseudo-inverse (`Codebook.project\_reverse`). The shift between symbolic and subsymbolic input at higher orders is the architectural locus of attention; both arrive at concept content width matching the perceptual content width that order-0 expects, so the dim chain is uniform without the legacy concat. References: `Spaces.ConceptualSpace._sourced_input` and `Spaces.ConceptualSpace._get_active_input_sibling`. The legacy `_build_combined_input` is dead code awaiting removal.

### Pipeline as a unit, two-tier reset

> *Post the rolling-cursor handoff (`plans/2026-04-26-rolling-cursor-doc-streaming-handoff.md`) and the brick-vectorization handoff (`plans/2026-04-27-brick-vectorization-and-legacy-removal-handoff.md`).*

`runBatch` is a pure compute brick: forward → loss → backward → optimizer.step. It does **not** decide when to reset per-row state, does **not** consume `_end_of_stream` for control flow, and (after the §6 vectorization landed) does **not** issue any GPU→host sync inside the brick body.

Reset lives in the outer doc-streaming loop in `runEpoch`. The same loop drives both the byte cursor (AR text byte) and the trial cursor (non-AR / numeric / non-byte) – there is no longer a separate DataLoader-iteration branch; `next_tick` is the universal dispatch (§8e of the brick-vectorization handoff):

```
while not ds.all_done():
    inp, out, hard_eos = ds.next_tick()              # 3-tuple, host-side
    runBatch(inp, out)                                # the compute brick
    flush_word_buffers()                              # §6c materialize subspace.word
    dispatch_per_row_reset(hard_eos)                  # hard resets
    dispatch_soft_reset()                             # grammar <start> reductions
    post_tick_compact()                               # truth_layer.compact
```

For AR text byte, `inp` is a byte slab and `hard_eos[b]` flips True when row b's cursor exhausts a doc. For non-AR / numeric data the cursor aligns with the trial: each tick yields one batch of trials with `hard_eos = [True] * B` (every row ends its trial each tick).

**Hard reset.** `TheData` walks each row's document one slab of up to 1024 bytes at a time. A row's `hard_eos` flips True the tick its cursor exhausts the current document. The full row-state cascade fires for that row only. Other rows continue mid-document with state preserved.

**Soft reset.** `Chart.compose` detects when a row's parse derivation reduces to `<start>` (a new top-level grammar element naming the start symbol; see Language.md). The signal accumulates on `wordSpace._sentence_completed: list[bool]` and is drained per-tick. A soft reset re-arms `_stm_fired[b]` and clears `_last_svo[b*K..]` and the parse-stack rows for `b`, but **preserves discourse history** — discourse accumulates across sentences within a document and clears only on hard reset.

**No truncation.** A document longer than `slab_bytes` (= 1024 by default) spans multiple ticks of its row. Concatenating the per-tick slabs for any row reproduces the original document byte-exact. The `valid_mask: [B, K]` contract handles partial-fill tails (last slab of a doc shorter than `slab_bytes`) via the same NULL-padding semantics already in place.

**Compute-brick contract.** No `.item()`, no `.tolist()`, no Python conditional on a tensor value, no GPU→host copy inside `runBatch`. The brick-vectorization handoff (§6) made this true:

- §6a removed `stm_residual_microbatch`'s `.item()` early-out (always call `disc.predict()`; gate the bias on already-fired rows via multiplication).
- §6b dropped the per-cell `should_store` gate from the truth layer; `record_batch` stages every cell with a trust score, and post-tick `compact()` filters in one host sync outside the brick.
- §6c adds tensor `word_records` / `word_count` buffers to `SubSpace` so the chart compose can scatter entries inside the brick; `flush_word_buffer` materializes them onto `subspace.word` once per tick from the outer loop. The chart compose dispatches one vector `add_word` call per depth step (replacing the per-row scalar loop).

CUDA-graph capture of the brick (§7 in the same handoff) is the remaining piece. Two residual `.tolist()` calls in `Chart._chart_inside` (`best_pair`, `best_rule_local`) plus a few `if compat.sum() == 0: break`-style data-dependent control flow points produce graph breaks; the plan defers handling those to the GB10-side capture wiring.

### Three-File Architecture

Model behaviour is partitioned across three independent artifacts:

| File | Contents | Managed by |
|---|---|---|
| **XML config** (e.g. `BasicModel.xml`) | Architecture, hyperparameters, paths | Hand-edited |
| **Embedding artifact** (e.g. `BasicModel.kv`) | Word vectors (codebook) | Phase 1: `embed.py` |
| **Weights checkpoint** (e.g. `BasicModel.ckpt`) | Model layer parameters (excludes embeddings) | Phase 2: `BasicModel.py` |

The `save_weights()` method explicitly filters out embedding parameters (`_emb.weight`) from the checkpoint. Embedding vectors live in the `.kv` artifact and are loaded separately. This separation allows:

- Retraining embeddings without touching model weights

- Retraining model layers with frozen embeddings
- Swapping codebooks between models that share the same architecture

---

## Language System

The symbolic and syntactic spaces implement a binary deep-structure grammar in Chomsky Normal Form. SymbolicSpace maps continuous concept activations to a discrete one-hot encoding via an invertible layer and codebook. SyntacticSpace generates a derivation tree from the active symbols, stored as word tuples `(batch, vector, rule)`.

**Unified S-tier grammar (2026-04-19 rewrite).** The previous C/P/S three-tier partition was collapsed: all compositional operations now live on the single S tier over a bivector-shaped SymbolicSubSpace. The 17 S-tier productions include the ternary operators (`true(S)`, `false(S)`, `non(S)`), the what/where/when column selectors, and the mereological and logical operators (`not(S)`, `part(S, S)`, `intersection(S, S)`, `union(S, S)`, `equals(S, S)`, `conjunction`, `disjunction`, `swap`, `query`, `lift`, `lower`).

Parthood (`part`) is the **fundamental** mereological operation, realized as clipped cosine projection on the bivector symbol subspace. The full mereological suite (`whole`, `equal`, `overlap`, `underlap`, `boundary`) composes through `part` on `Basis`. `equals(S, S)` is propositional identity on S and delegates to `Basis.equal` (mutual parthood).

See Logic.md for the parthood formula and suite, Mereology.md for the five-relations reference and the `ImpenetrableLayer` regularizer, and Language.md for the full grammar, word encoding, and open implementation questions about differentiable tree structure and rule operations.

**Shamatha Speech target.** A separate narrow grammar mode is planned for one-pointed object speech: form a complete DNF over the active percepts, but permit each `conjunction` / `disjunction` only when the operands' `where()` supports are connected and their `when()` supports are continuous. This is not serial cursor mode; it may compose over all active percepts, then render the resulting object as strict DNF English. See Language.md and plans/2026-04-28-shamatha-speech-contiguity-handoff.md.

---

## Sigma and Pi Layers

Given a weight matrix $W \in \mathbb{R}^{m \times n}$ and input vector $x \in \mathbb{R}^n$, the output vector $y \in \mathbb{R}^m$ is computed as:

For the Sigma layer:

$$y_j = Wx + b$$

$$y_j = b_j + \sum_{i=1}^{n} W_{ji} x_i$$

For the Pi layer (code implementation – log-space linear):

$$s_i = \log \frac{1 + x_i}{1 - x_i} = 2 \operatorname{atanh}(x_i)$$

$$z_j = \sum_i W_{ji}\, s_i + b_j$$

$$y_j = \frac{e^{z_j} - 1}{e^{z_j} + 1} = \tanh(z_j/2)$$

The forward path maps $[-1, 1] \to (0, \infty)$ via `_to_mult`, takes the log, applies a linear transform (InvertibleLinearLayer), exponentiates, and maps back via `_from_mult`. Domain and range are both $[-1, 1]$. The reverse path inverts each step exactly: `_to_mult(y)`, log, $W^{-1}(z - b)$, exp, `_from_mult`.

**Conceptual motivation.** The classical product form $y_j = b_j \prod_i (1 + W_{ji} x_i)$ becomes, after taking logs, a sum $\log y_j = \log b_j + \sum_i \log(1 + W_{ji} x_i)$. The code generalises this idea: it moves into a log-multiplicative domain via atanh, performs a standard linear operation there, and returns via tanh. The atanh transform stretches values near $\pm 1$ toward infinity, making the layer sensitive to strong activations – the multiplicative structure emerges from the nonlinear domain transform rather than from literal products.

**Monotonicity of the bivector chain.** When `<bivectorOutput>true</bivectorOutput>` is configured on PerceptualSpace, ConceptualSpace, and SymbolicSpace, every activation in the P $\to$ C $\to$ S chain lives on the non-negative paired-index cone `[0, 1]^{2K}`. The Pi / Sigma layers select `NonNegativeInvertibleLinearLayer` (or `NonNegativeLinearLayer` in the non-invertible case) under `monotonic=True`, giving entry-wise $W \geq 0$. A positive matrix is the canonical monotone operator on the positive cone: $a \leq b$ componentwise $\Rightarrow Wa \leq Wb$ componentwise. The componentwise order on the cone *is* the parthood partial order, so every lift / lower in the chain is order-preserving pole-by-pole. PerceptualSpace joins the chain via the Q2 promotion `(aP, aN) = (\max(0, x), \max(0, -x))` at its activation boundary (spec §Q2 / §B3). The bivector layout keeps the contradiction corner `[1, 1]` distinct from the ignorance corner `[0, 0]` under positive matmul – a single bitonic axis would let $aP - aN$ cancel under summation, collapsing both to zero. See Spaces.md "Monotonicity of the lift / lower chain".

---

## Dimensionality Constraints

- The output dimensionality of the input layer must be equal to the output dimensionality of the perceptual layer, since the conceptual layer operates on both.
- The input dimensionality of the symbolic layer must be equal to the input dimensionality of the perceptual layer, since they both operate on the output of the conceptual layer.
- The input dimensionality of the output layer must be equal to the sum of the output dimensionalities of the symbolic layers.

---

## Pi Layer Derivation (Historical)

The original motivation for the multiplicative layer was to factorise an exponential sum into a product of linear terms:

$$e^{y_j} = e^{b_j} + \sum_{i=1}^{n} e^{1 + W_{ji} x_i}$$

$$y_j = b_j \cdot \prod_{i=1}^{n} (1 + W_{ji} x_i)$$

The current implementation replaces this with the log-space linear form described above, which achieves the same multiplicative semantics through atanh/tanh domain transforms while guaranteeing exact invertibility via the LDU-factored InvertibleLinearLayer.

---

## Invertible Linear Layer (LDU)

### InvertibleLinearLayer – LDU Factorisation

The core invertible linear primitive factors the weight matrix W as:

```
W = L @ D_embed @ U
```

where:

- $L \in \mathbb{R}^{nIn \times nIn}$: unit lower-triangular matrix (diagonal fixed at 1). Stored as `raw_L`; the strict lower triangle is extracted and the diagonal is forced to 1 at each forward call.
- **D**: diagonal vector of length `rank = min(nIn, nOut)`, embedded into a rectangular `[nIn, nOut]` matrix `D_embed` by zero-padding.
- $U \in \mathbb{R}^{nOut \times nOut}$: unit upper-triangular matrix (diagonal fixed at 1). Stored as `raw_U`; the strict upper triangle is extracted symmetrically.

**Exact inverse via triangular solves.**

```
W^{-1} = U^{-1} @ D^{-1} @ L^{-1}
```

Each factor is inverted by a triangular solve (`torch.linalg.solve_triangular`). No SVD is required, and the inverse is exact as long as all diagonal entries of D are nonzero.

**Parameter count.** $nIn^2 + \text{rank} + nOut^2$ total parameters. Initialized at L = I, d = 1, U = I so that the initial map is the identity.

### Ergodic Noise Injection (Factor-Level)

Rather than the matrix-level blend $W_{\text{eff}} = b \cdot W + t \cdot N$ (which destroys the LDU structure and makes the inverse approximate), noise is injected directly into the raw parameters of each factor before the triangular structure is extracted:

```
L_eff = I + strict_lower(raw_L + t * noise_raw_L)
U_eff = I + strict_upper(raw_U + t * noise_raw_U)
d_eff = b * d_effective + t * noise_d
W_eff = L_eff @ D_eff_embed @ U_eff
```

Because `W_eff` is still in LDU form, its exact inverse is always available by the same triangular solves – no approximation, regardless of the noise level.

### stable=True

When `stable=True`, `_d_effective()` clamps each diagonal entry `d_i` to magnitude `[eps, 1]` with sign preserved before the ergodic blend:

```
d_clamped_i = sign(d_i) * clamp(|d_i|, eps, 1)
d_eff = b * d_clamped + t * noise_d
```

This keeps `d_eff` bounded away from zero, preventing `W_eff` from becoming singular. `stable=True` is the only place the stability constraint is enforced; no additional clamp on `d_eff` is needed.

### noise_d

Noise for the diagonal factor is sampled with $|d_i| \in [\text{eps}, 1]$ and random sign, so the noise factor is itself always invertible. This ensures that even at full temperature (`b=0, t=1`) the effective matrix is still well-conditioned.

### naive Flag

| `naive` | Forward path | Reverse path |
|---|---|---|
| `False` (default) | Apply L, D, U sequentially to x; backprop through each factor separately | Triangular solves: $U^{-1}$, $D^{-1}$, |
| `True` | Materialise `W_eff` as a dense matrix; apply `W_eff @ x` | Materialise the dense LDU in |

The `naive=False` path never materialises W_eff as a full matrix, saving memory and allowing each factor's gradients to flow independently.

### Noise Lifecycle in Ergodic Mode

Noise buffers (`noise_raw_L`, `noise_raw_U`, `noise_d`) are:

1. **Resampled at the start of every ergodic `forward()`** – new noise is drawn before constructing `W_eff`.
2. **Resampled again at the end of every ergodic `reverse()`** – fresh noise is drawn after the reverse computation completes.

The window between the two calls is exactly when `reverse()` needs to reconstruct the same `L_eff`, `U_eff`, `d_eff` that `forward()` used. Because the buffers are not resampled during that window, `reverse()` reads the stored buffers directly – no explicit caching of `W_eff` is required.

### Class Renames

The LDU layer supersedes the previous SVD-based implementation:

| Old name | New name | Status |
|---|---|---|
| `LULayer` | `InvertibleLinearLayer` | Renamed |
| `InvertibleLinearLayer` (SVD) | – | Removed |
| `InvertibleSigmaLayer` | `SigmaLayer(invertible=True)` | Merged |
| `InvertiblePiLayer` | `PiLayer(invertible=True)` | Merged |

---

## Ergodic Exploration

The ergodic bias-variance control system is documented in Ergodic.md.

# Basic Model

***The Basic Model of Cognition is a neural network that consists of four spaces: real space, perceptual space, conceptual space, and symbolic space.***

**Model Entities:**

| Entity | Meaning |
|---|---|
| $x$, $\hat{x}$, $X$ | Actual and predicted input (experiences), input space |
| $y$, $\hat{y}$, $Y$ | Actual and predicted output (actions), output space |
| $p$, $P$ | Percepts, perceptual space |
| $c$, $C$ | Concepts, conceptual space |
| $s$, $S$ | Symbols, symbolic space |
| $W_e$ | Experiencing, experiential weights |
| $W_a$ | Acting, action weights |
| $W_k$ | Knowing, knowledge weights |
| $W_t$ | Thinking, thought weights |

### *Background*

Much of this work is based on the Basic Model of Cognition described briefly at cognitivesciencesociety.org/framework-cognitive
More details are provided in the book The Whole Part.

### *Inputs and Outputs*

Inputs and outputs derive from a real space that is ineffable. This essay merely characterizes them relative to other things.

The adaptable parameters of a mind are initially zero, a state known as *tabula rosa*. Learning is initiated by random perturbations of a map relative to its territory, and the degree of that exploration is referred to as *temperature*.

Inputs are scaled to `[-1, 1]` via the global data min/max. The signed range allows the network to represent both presence and absence symmetrically, while the philosophical point remains: to paraphrase Saint Augustine, *negative is the privation of the positive*.

Outputs and Inputs are dehumanizing terms: the world is a living world, so it helps to call outputs "actions" and inputs "experiences". When minds perceive and act within the world, their mistakes are called expectation errors and performance errors.

Action is a function of perceptual space and our expectation, where that space is often filtered by conceptual projections.

Performance errors are defined relative to action, desired effect, and exhaustion.

Expectation errors are defined relative to perception and perception-as-reconstructed.

To accommodate an input context window which changes, the input is augmented with relative spatiotemporal encoding by extending its dimensionality. While much of the network can treat these as simply additional features of the input, the attention mechanism must deal with them explicitly since attention is directed within space and time (i.e. it treats the what and where pathways separately).

***Perceptual Space***

Perceptual spaces are composed of perceptual prototype vectors, or ***percepts***.[1]

Perceptual spaces map onto reality in a straightforward way, at least in so far as we perceive correctly They are bidirectional, although not necessarily symmetric. Technically, they are real similarity spaces with arbitrarily high dimensionality, where logical negation is not defined.

Percepts that do not correspond to objects are *hallucinations*.

Percepts generalize a perceptual space by forming a Voronoi tessellation using prototype and/or exemplar vectors. They are related to one another in virtue of a connection which define a Self Organizing Map. Percepts operate in parallel. Insofar as categories are concerned, they form an embodiment of prototype theory.

The perceptual error function is defined by the mismatch between perception and existing percepts, where that mismatch may exert an influence on the world.

Perceptual accuracy is a measure of the distance between inputs and prototype vectors. That distance is similar to cosine similarity, although to the extent we don't know a perceptual well, it is a simple distance measure in the range (0-1).

Attraction and aversion within a perceptual space manifest as weighted prototype vectors: attraction is a pole at the percepts's location, and aversion is a zero at that location. As a result, they warp perceptual space, making us more or less likely to perceive things that we love or hate.

Perfectly recognized percepts lie on the unit hypersphere within conceptual space.

Two percepts can be joined into one concept without loss of characterization.

***Conceptual Space***

***Concepts*** create boundaries in conceptual space, a space which is grounded in the support vectors of perception. By creating boundaries, they support logical negation. Due to this relativity, they may refer to negative entities which may or may not exist on perceptual or objective levels.

[1] Perceptual space is known as "embedding space" in the current literature on transformer neural net architectures.

Conceptual spaces are capable of representing negative entities.

Concepts are references that do not exist in the space in which the percepts are defined, but in the space formed by the collections of perceptual activations. Thus, the space of concepts can not be directly overlaid on the space of percepts, because it occurs at a different epistemic level, although conceptual space can be projected onto perceptual space.

Concepts that do not correspond to percepts are *false*.

Concepts collect percepts into fuzzy sets. Percepts are weighted by a degree of membership, which allow concepts to perform several logical operations such as and, or, and not. They can be seen either as linear separating hyperplanes which partition conceptual space into two parts, or as sets which are defined as a linear combination of the percept activations.

Concepts are relative: they are meaningful only insofar as they create a partition of a meaningful space.

As the concepts of a mind approach certainty, the slope of the decision boundary becomes infinite.

There is a single state of unknowing which is common to all knowledge vectors, which does not categorize space because the slope of the activation function is zero.

It is also the case that knowledge vectors of length zero are undirected, which seems like a state of unknowing which is slightly different than slightly different the measure of certainty. Informally, the concept must be learned before its certainty can be assessed.

Thus, conceptual spaces are similarity spaces that are partitioned by decision boundaries that impose positivity and negativity on a hypersphere of knowing.

Percepts are incorporated into meronomies through the use of concepts, which create transitive part hierarchies.

When symbols are so incorporated, the referential nature of symbols creates intransitive part hierarchies.

Within conceptual space, attraction and aversion to concepts can be expressed as a hyperplane bias (i.e. the tendency to partition reality in a biased way).

Within conceptual space, certainty is expressed as the slope of the activation function. Is it tuned after the other parameters have ceased to change as a result of correct categorization.

Certainty (or clarity) can also be grounded in symbolic space, where symbolic support vectors are taken as *a priori* truth (or at least learned with some certainty reflected in their nonzero magnitude).

Concepts can be split into two precepts without loss of characterization.

***Symbolic Space***

*Symbolic Spaces* are composed of percepts that are references to concepts. They are therefore a subtype of perceptual spaces that map onto conceptual spaces.

Because symbols do not have any spatial context, they exist as zero-dimensional points within perceptual space. If we dont have feeling receptors in the brain, there is no positional encoding for this layer.

In virtue of that representation, symbols can be conceptualized and collected into sets. The creation of sets of symbols, since they have no inherent ordering, creates equivalence classes.

In humans, perceptual space is projected onto a 3D subspace within the cerebellum before symbolic encoding.

Symbols are independent, permanent, and integral (at least when interpreted *as* references, even though they ultimately exist within a dependent, temporary, and continuous space).

The top-down, serial activation of a symbolic space differs from the bottom-up activation of perceptual spaces because it casts shadows.

The formation of symbols increases the dimensionality of conceptual space (as would an outer product).

The formation of concepts from percepts decreases the dimensionality of perceptual space (as would an inner product).

Symbols are zero-dimensional entities, or epistemological points. As references, symbols form a discrete perceptual space and can be represented as a binary array.

The participation of symbols in multiple sets creates structural relations which takes the place of positional information.[2]

Symbols are erroneous if their associated concept is not true.

***Reasoning***

Reasoning within symbolic spaces involves computing with parts and wholes. This may be seen as using Venn diagrams to perform computations on a space, which results in ability to perform Bayesian-type statistics dynamically.

Those probabilities, once learned, can be stored as unidirectional connection weights between concepts, perhaps in conjunction with a part-whole structure imposed on those concepts by their corresponding symbols.

***Symmetric Perception: the Single Layer Linear Case***

- A symmetric perception network predicts both input and output. By taking advantage of functions that may be surjective in two directions, it is able to compute a bijective mapping.
- The term "*symmetric perception*" refers to the fact that perception is bidirectional: it is an interaction in which we see the world and in which the world sees us. The later fact is often forgotten from an egocentric point of view; culturally, as Edward has observed, perception is seen as a passive process. In the context of neural networks, which often serve as models of the mind, this egocentric view continues: functions are computed that map inputs onto outputs, and the influence of the outputs on the inputs is forgotten. This essay explores the technical aspects of symmetric perception as well as several guiding principles for humanistic mental models.

Symmetric Perception, in the general case, finds a function f(x) which minimizes the MSE with respect to the following:

$$\begin{pmatrix} in \\ out \end{pmatrix} \cdot \begin{pmatrix} f(x) & 0 \\ 0 & f^{-1}(x) \end{pmatrix} = \begin{pmatrix} out \\ in \end{pmatrix}$$

In the linear case, $f(x)$ becomes a single invertible matrix A. In general, the mapping from input to output may not be linear, and the inverse may not exist, in which case the previous set of input/output equations is written as:

$$\begin{pmatrix} in \\ out \end{pmatrix} \cdot \begin{pmatrix} f(x) & 0 \\ 0 & g(x) \end{pmatrix} = \begin{pmatrix} out \\ in \end{pmatrix}$$

This problem bears some resemblance to the standard regression problem $Ax = b$, except that x is known and we wish to find the set of points $A$ (or weights) that map one linear space to another. So the familiar Linear Least-Squares solution to the equation $x = \left(A^\top A\right)^{-1} A^\top b$ becomes an attempt to find a matrix $A$ such that:

$Ax = b$ and $A^{-1}b = x$

This solution is seen as the "best" solution by neural networks that wish to determine a weight matrix A that maps from x onto b. That much is shared between the two problems, although there are nonlinearities

[2]For example, {0,1} and {0,1}, if those sets are orthogonal to one another, pose no constraint on the location of the 1 within the 2x2 grid that they partition. However, {0,1,1} and {0,1} partition a 3x2 space and simultaneously impose the constraint that two 1's must lie on the same row. This is a fact which is not present in either the row set or the column set.

in the general neural network case which enable a much better fit. This solution is optimal in the sense that it minimizes the Sum of Squared Errors, where the errors are defined as the residuals when b is projected into the column space of A.

However, the principle of retrocausality suggests an additional objective; namely, we wish to create a matrix such that it is optimal both for the the transformation of the input to the output and the output to the input.

In general, we want a matrix $A$ such that minimizes reconstruction loss over both $Ax = b$ and $A^{-1}b = x$. Since we are simultaneously minimizing the error vectors of each, and since the column space of a matrix is identical to the column space of its inverse, we will end up with a matrix $A$ that contains the vectors $x$ and $b$ within both its column space and row space (which is necessary because we are seeking a bijective function).

As a first approximation, we might write $A$ as a product of two matrices, each of which performs a perfect surjection: $A \cdot A^{-1} = I$

$$(\frac{b}{2} * x^{-1}) \cdot (\frac{x}{2} * b^{-1}) = I$$

However, this will still be only an approximate solution, since it is unlikely that a somewhat arbitrary combination of both matrices will result in the correct orthogonal matrix. To find the best matrix $A$ involves either an iterative solution as would be performed by NN learning, ping-pong style, or (theoretically?) via multiple round trips consisting of Gram-Schmidt orthogonalization with respect to the matrices $A$ and $A^{-1}$.

***Symmetric Perception: the Single Layer Nonlinear Case***

If $f(x)$ and $g(x)$ share a common weight matrix $W$, we may write the equations for symmetric perception as:

Forward computation and gradient:

$$\hat{y} = f(g^{-1}(x)W)$$

$$e_y = \frac{1}{2}(\hat{y} - y)^2$$

$$\frac{\partial e_y}{\partial \hat{y}} = \hat{y} - y$$

$$\frac{\partial \hat{y}}{\partial W} = \delta_f(W^\top \cdot g^{-1}(x))$$

$$\frac{\partial W}{\partial x} = \delta_g(x)$$

$$\Delta W_y = \eta \frac{\partial e_y}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial W} g^{-1}(x)$$

$$\Delta W_y = \eta(\hat{y} - y)\delta_f(W^\top \cdot g^{-1}(x))g^{-1}(x)$$

Reverse computation and gradient:

$$\hat{x} = g(W^\top f^{-1}(y))$$

$$e_x = \frac{1}{2}(\hat{x} - x)^2$$

$$\frac{\partial e_x}{\partial \hat{x}} = \hat{x} - x$$

$$\frac{\partial \hat{x}}{\partial W} = \delta_g(W \cdot f^{-1}(y))$$

$$\frac{\partial W}{\partial y} = \delta_f(y)$$

$$\Delta W_x = \eta \frac{\partial e_x}{\partial \hat{x}} \frac{\partial \hat{x}}{\partial W} f^{-1}(y)$$

$$\Delta W_x = \eta(\hat{x} - x)\delta_g(W \cdot f^{-1}(y))f^{-1}(y)$$

Since the single matrix $W$ is shared, we might combine the partial derivatives of the input and output errors via sum and set the gradient to zero to explore the solution space, which seems to minimize the cross-covariance between the input and output spaces:

$$\frac{\partial e_y}{\partial W} + \frac{\partial e_x}{\partial W} = \eta(\hat{y} - y)\delta_f(W^\top \cdot g^{-1}(x))g^{-1}(x) + \eta(\hat{x} - x)\delta_g(W \cdot f^{-1}(y))f^{-1}(y)$$

If we assume that the gradients never go to zero, then we have the sum of the partial derivatives with respect to the error going to zero when both $x = i$ and $y = o$. However, it may be the case that $xy - iy = ox - yx$, which means that the simultaneous gradient descent is ping-ponging between the two cost functions. So a better cost function would be a multiplication of the two error partials, which is an intersection of the space that satisfies both equations. In that case, we can multiply the terms to get a better understanding of the error surface that we are minimizing with respect to the input and the output:

$$\frac{\partial e_y}{\partial W} \frac{\partial e_x}{\partial W} = \eta(\hat{y} - y)\delta_f(W^\top \cdot g^{-1}(x))g^{-1}(x)(\hat{x} - x)\delta_g(W \cdot f^{-1}(y))f^{-1}(y)$$

***Distance Metrics***

$$d(x_1, x_2) = \frac{\|x_1\|^n \cdot \|x_2\|^n}{\|x_1 - x_2\|^{2n}}$$

Which can be written in the Z-plane as:

$$d(x_1, x_2) = \|x_1 - x_2\| \cdot \frac{(x - z_1)}{(x - z_2)}$$

Where $z_1$ is a zero and $z_2$ is a pole.

***Exploration/Exploitation as a function of temperature:***

$$y = x \cdot \frac{(W + rt)}{\|W\|}$$

Where:

- $\boldsymbol{x}$, $\boldsymbol{y}$ are the input and output vectors
- $\boldsymbol{W}$ is the weight matrix
- $\boldsymbol{r}$ is a random vector
- $\boldsymbol{t}$ represents temperature

***Transfer Functions***

By using rotation matrices with a tunable parameter $\Theta$ before and after the projection from one space to the other, and by restricting the projection matrix to be diagonal, it is possible to restrict the solution space to an eigendecomposition of the orthonormal basis that maps input to output (i.e. it calculates the SVD of the matrix). For the 2D case, this can be written:

$$y = \begin{bmatrix} \cos\theta_1 & -\sin\theta_1 \\ \sin\theta_1 & \cos\theta_1 \end{bmatrix} \begin{bmatrix} \Sigma_1 & 0 \\ 0 & \Sigma_2 \end{bmatrix} \begin{bmatrix} \cos\theta_2 & -\sin\theta_2 \\ \sin\theta_2 & \cos\theta_2 \end{bmatrix} x$$

It may also be possible to use a nontunable scalar function as the transfer function and not restricting the matrix to be diagonal, which is approximately how neural networks apply nonlinearities to the sum of products by weight matrices.

$$f(x) = \sin(x)$$

$$g(x) = \cos(x)$$

$$f^{-1}(x) = \arcsin(x)$$

$$g^{-1}(x) = \arccos(x)$$

$$\partial_f(x) = \cos(x)$$

$$\partial_g(x) = \sin(x)$$

$$\partial_{f^{-1}}(x) = \frac{1}{\cos(\arcsin(x))}$$

$$\partial_{g^{-1}}(x) = \frac{1}{\sin(\arccos(x))}$$

If we write the forward equations and gradients of $f()$ and $g()$ using these functions, we end up with the following formulation:

$$\hat{y} = \sin(\cos^{-1}(x)W)$$

$$\hat{x} = \cos(W^\top \sin^{-1}(y))$$

$$\Delta W_y = \eta(\hat{y} - y)\cos(W^\top \cdot \cos^{-1}(x))\cos^{-1}(x)$$

$$\Delta W_x = \eta(\hat{x} - x)\sin(W \cdot \sin^{-1}(y))\sin^{-1}(y)$$

The important thing to notice is that the reconstruction of the reverse perception and the calculation of the gradient of forward perception involve common terms, and may be simplified:

$$\hat{y} = f(g^{-1}(x)W)$$

$$\hat{x} = g(W^\top f^{-1}(y))$$

$$\Delta W_y = \eta(\hat{y} - y)\delta_f(W^\top \cdot g^{-1}(x))g^{-1}(x)$$

$$\Delta W_x = \eta(\hat{x} - x)\delta_g(W \cdot f^{-1}(y))f^{-1}(y)$$

$$\Delta W_y = \eta(f(g^{-1}(x)W) - y)\delta_f(W^\top \cdot g^{-1}(x))g^{-1}(x)$$

$$\Delta W_x = \eta(g(W^\top f^{-1}(y)) - x)\delta_g(W \cdot f^{-1}(y))f^{-1}(y)$$

$$\Delta W_y \Delta W_x = \eta(f(g^{-1}(x)W) - y)\delta_f(W^\top \cdot g^{-1}(x))g^{-1}(x) \cdot (g(W^\top f^{-1}(y)) - x)\delta_g(W \cdot f^{-1}(y))f^{-1}(y)$$

If we take the product of both:

$$\Delta W_y \Delta W_x = \eta(\hat{y} - y)\delta_f(W^\top \cdot g^{-1}(x))g^{-1}(x) \cdot (\hat{x} - x)\delta_g(W \cdot f^{-1}(y))f^{-1}(y)$$

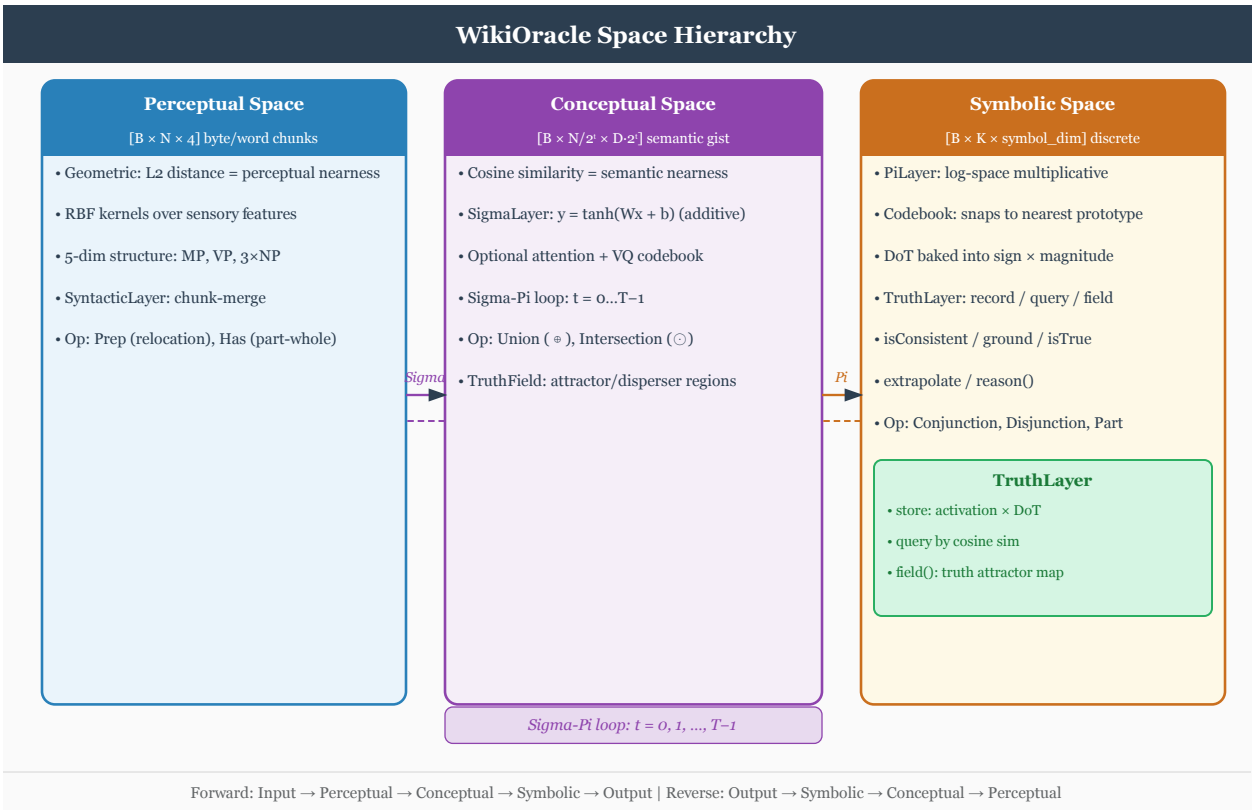
$$\Delta W_y \Delta W_x = \eta(\hat{x} - x)(\hat{y} - y)\delta_f(W^\top \cdot g^{-1}(x))\delta_g(W \cdot f^{-1}(y))f^{-1}(y)g^{-1}(x)$$

# Spaces

## Overview

BasicModel is organized as a pipeline of six **spaces**, each performing a distinct representational transformation. Data flows forward from raw input to task output; the reverse pass reconstructs the original input from the symbolic representation.

```
Forward:  InputSpace -> PerceptualSpace -> ConceptualSpace -> SymbolicSpace -> SyntacticSpace -> OutputS
Reverse:  OutputSpace -> SyntacticSpace -> SymbolicSpace -> ConceptualSpace -> PerceptualSpace -> InputS
```




---

## Base Class: Space

All spaces inherit from a common `Space` base that manages the shared infrastructure:

**Shape management.** Each space tracks `inputShape` and `outputShape` as `[nObjects, nDim]` tuples, where `nObjects` is the active count and `nDim` is the per-object dimensionality. Dimensions are not passed as constructor arguments; each subclass reads them from `TheObjectEncoding` (e.g. `TheObjectEncoding.perceptDim` for PerceptualSpace).

**Codebook / VQ quantization.** When `nVectors > nActive`, the space maintains a codebook of `nVectors` candidate vectors and selects the `nActive` most activated ones via top-k selection. This is the vector-quantization bottleneck.

**Reshape flag.** When the input and output object counts differ, the reshape flag flattens the `[B, nObj, nDim]` tensor to `[B, nObj * nDim]` before passing it to the next space, then restores the object structure on the way back.

**Attention.** Each space may optionally include an attention mechanism (`hasAttention=true`) that reweights objects before the main transformation.

**set_sigma propagation.** When ergodic mode is active, `set_sigma()` is forwarded from the top-level model down through every space and into every child layer, so the per-neuron exploration level is kept consistent across the entire pipeline.

**Reset cascade — hard vs. soft.** Every space exposes `Reset(batch=None, hard=True)`. The `(batch, hard)` signature is now **required** — the legacy zero-arg fallback in `_reset_call` was removed by the brick-vectorization handoff (§8d).

| Call form | Scope | Use |
|---|---|---|
| `space.Reset()` | All rows | Whole-state wipe (e.g. epoch boundary, manua |
| `space.Reset(batch=b, hard=True)` | Row `b` only | Document boundary, full per-row state wipe |
| `wordSpace.soft_reset(batch=b)` | Row `b`, sentence-scoped state only | Grammar `<start>` reduction completes |

Hard-reset clears: parse stack, `_last_svo`, `_stm_fired`, codebook commit accumulator, discourse history, `serial_cache`, `_ar_embedded`, `_end_of_stream` for the affected row(s).

Soft-reset clears the per-sentence working buffers: the parse stack rows owned by the source row, `_last_svo[b]`, the category and reconstruction stacks for those rows, and re-arms `_stm_fired[b]` so the next sentence's discourse-prediction bias fires once. **Does not** touch discourse history (`InterSentenceLayer` ring buffer) or codebook EMA — those are document-scoped and form the inter-sentence prior.

Reset is dispatched from the outer doc-streaming loop in `runEpoch`, never from inside `runBatch` (which is a pure compute brick — see Architecture.md "Pipeline as a unit, two-tier reset").

---

## Normalization and Ranges

Every space enforces a canonical range on its vectors and activations. Each space always normalizes its output to the correct range.

| Space | Data Contract |
|---|---|
| InputSpace | Data scaled -1..1 for scalars or vector norms |
| PerceptualSpace | Modal/demuxed (what/where/when encoding). Signed unit-magnitude scalars or vectors centered at the |
| ConceptualSpace | Combined/muxed (event encoding). Signed unit-magnitude scalars or vectors (tanh-bounded). Event no |
| SymbolicSpace | Symbols are percepts. Concept-to-symbol mapping. One symbol encoded at a time (most highly active) |
| SyntacticSpace | Words are concepts encoding grammatical rules. Stored as word tuples with production rules |
| OutputSpace | Rescaled from activation range to original data range |

**Data scaling.** `Data` computes global scalar `input_min`/`input_max` and `output_min`/`output_max` from the training data at load time. InputSpace uses `Data.normalize(x, "input")` to scale non-embedding inputs to `[-1, 1]`, and `Data.denormalize(x, "input")` in reverse to restore the original range. OutputSpace uses `Data.denormalize(x, "output")` to map from `[-1, 1]` (symbolic activation range) to the original output range, and `Data.normalize(x, "output")` in reverse.

**Symbols as percepts.** Symbols represent the presence or absence of a named entity and live in `[0, 1]`. Since conceptual activations range `[-1, 1]`, the mapping is `symbol = (activation + 1) / 2`. The `SubSpace.get_symbols()` and `set_symbols()` methods perform this conversion, and SymbolicSpace / SyntacticSpace use them exclusively instead of `get_activation` / `set_activation`.

**Demuxed mode.** When `InputSpace.demuxed=true`, what/where/when components are stored independently in the SubSpace rather than concatenated into a single event tensor. This keeps the codebook pure (positional data never contaminates content vectors). ModalSpace routes each component through independent PerceptualSpaces. Downstream spaces see an identical muxed tensor via `materialize()`.

---

## Codebook Similarity Metric

`Codebook` (in `bin/Spaces.py`) wraps `VectorQuantize` (in `bin/Layers.py`, moved out of Spaces.py during the April 2026 perf pass). The choice of similarity metric for nearest-code retrieval is not one-size-fits-all across spaces — it follows the geometry of what each codebook stores.

**Per-space metric**

| Space | Codebook geometry | What is stored |
|---|---|---|
| PerceptualSpace | $[-1, +1]^d$ hypercube | Feature *patterns* — magnitud |
| SymbolicSpace | $[-1, +1]^d$ hypercube (paired bivector slots for tetralemma corners) | Symbol *patterns* — same as p |
| ConceptualSpace | Unit L2-norm directions (concepts are *named directions* in belief space) | Concept *directions*; the *input* |

### Why Euclidean for Perceptual and Symbolic

These codebooks store *what something looks like* (a pattern of feature intensities). A vector `0.5$\cdot$v` carries half as much "of feature v" as `1.0$\cdot$v`, and the right notion of "different code" is *coordinate-wise distance*, not direction. Cosine would conflate `0.5$\cdot$v` with `1.0$\cdot$v` because they share a direction — that loses real information. There is no codebook-level negation operator on these spaces (negation in SymbolicSpace is encoded structurally in paired-index bivector slots, not by negating the vector), so the natural metric is Euclidean L2.

The retrieval is implemented as a matmul + cached-norm subtract, not `torch.cdist`. Expanding the squared distance:

$$\|x - c_i\|^2 = \|x\|^2 + \|c_i\|^2 - 2\,(x \cdot c_i)$$

$\|x\|^2$ is a positive constant across $i$ and drops from the argmin. So:

$$\arg\min_i \|x - c_i\|^2 = \arg\max_i (x \cdot c_i - \tfrac{1}{2}\,\|c_i\|^2)$$

`VectorQuantize` keeps $\|c_i\|^2$ in a `[V]` buffer (`_b_norms_sq`, refreshed in the codebook setter and at the end of each EMA update), and each forward does:

```
indices = (flat @ codebook.T - 0.5 * b_norms_sq).argmax(dim=-1)
```

That's one matmul (`[N, D] $\cdot$ [D, V]`) plus one broadcast subtract plus one argmax. Same FLOPs as `torch.cdist`'s internal mm-trick path, but skips the `sqrt`, the per-row `$\Vert$x$\Vert$$^2$` add, and the cdist autograd plumbing — and gives Inductor a smaller graph to compile.

The naive expansion `((codebook - x)**2).sum(-1)` would be **slower** because it broadcasts to a full `[N, V, D]` intermediate before reducing (GBs of memory at PerceptualSpace's `V = 8192`). Both `cdist`'s mm-trick and the cached-norm matmul above avoid that.

### Why dot product (not Euclidean, not cosine) for Conceptual

ConceptualSpace concepts are *named directions* in belief space. The codebook entry $c_i$ is a unit vector pointing toward concept $i$. An input $x$ projected onto $c_i$ via the dot product gives the *signed strength of belief that $x$ affirms concept $i$*:

- $x \cdot c_i = +1$ — input fully affirms concept $i$
- $x \cdot c_i = 0$ — input is orthogonal (no information about $i$)
- $x \cdot c_i = -1$ — input fully denies concept $i$ (strong negative)
- intermediate values — partial belief, with sign preserved

Nearest concept (most-affirmed) is $\arg\max_i (x \cdot c_i)$. Two consequences:

1. **The codebook must be unit L2-norm.** The `EMA` path in `VectorQuantize` (the `use_cosine_sim=True` branch) renormalizes the running codebook to unit norm after each update, so this invariant holds end-to-end. Init-time normalization happens in `Codebook.addVectors`.

2. **The input must NOT be normalized.** The input magnitude *is the certainty signal* and must survive into the dot product so downstream layers see the right strength. Cosine similarity would divide it out — wrong for this space.

For *ranking* the concepts (which is what `argmax` cares about), $x \cdot c_i$ and $\cos(x, c_i) = (x \cdot c_i)/\|x\|$ are monotone-equivalent because $\|x\| \geq 0$ is a positive constant across $i$ and cancels out. So omitting the input-side normalization preserves the certainty signal *and* costs less — $O(N \cdot D + V \cdot D)$ for the matmul (no per-input normalize sweep).

This is why a single matmul suffices for ConceptualSpace retrieval:

```
# codebook is unit L2-norm (maintained by EMA)
indices = (flat @ codebook.T).argmax(dim=-1)
```

### Configuring the metric

The metric is a class attribute on `Codebook` (`use_dot_product`, default `False`). Each `Space` exposes a parallel class attribute of the same name; `Space._build_object_basis` (and the other Codebook-construction sites) mirrors `self.use_dot_product` onto the basis instance before calling `basis.create()`. To opt a Space into dot-product retrieval, set `use_dot_product = True` as a class attribute on the Space subclass — `ConceptualSpace` does this; no XML knob is required. `Codebook.addVectors` reads `self.use_dot_product` and propagates it as `VectorQuantize(use_cosine_sim=…)`.

The flag name `use_cosine_sim` on the underlying `VectorQuantize` is historical (it originally meant "normalize both sides and use cosine"); after the April 2026 perf pass the input-side normalization is gone for exactly the reason above, so the operator is now a pure dot product and the flag's effective meaning is "codebook-is-unit- norm; rank by dot product".

The default codebook initialization (`Codebook.addVectors`) likewise branches: dot-product mode L2-normalizes the random init so the EMA path starts from a valid unit-norm codebook; pattern mode rescales to the `[-1, +1]` hypercube.

---

## Codebook Uniqueness Contract

Every codebook entry — across InputSpace's vocabulary, PerceptualSpace's Lexicon, and the ConceptualSpace / SymbolicSpace prototype codebooks — must be **unique under both `WhereEncoding` and `WhatEncoding`**. The two axes carry independent uniqueness invariants:

**`.where` — globally unique positional key.** Enforced by the class-level **codebook offset registry** on `WhereEncoding` (`Spaces.py:223-276`). Each codebook calls `WhereEncoding.allocate_codebook_slice(n_vectors)` at construction and receives a contiguous integer offset; all codebooks share the same sinusoidal `div_term = 2$\pi$ / total_allocated`, so the `(sin, cos)` decode is unambiguous across the union of slices. A given codebook entry's `.where` is therefore a globally-unique key — InputSpace vocab item $k$, PerceptualSpace prototype $i$, ConceptualSpace prototype $j$, and SymbolicSpace prototype $m$ all live in disjoint `.where` slices.

**`.what` — distinct prototype content within and across codebooks.** Two entries with identical `.what` content collapse to the same parthood identity (`equal(A, A) = 1`) and become interchangeable under any positive Pi / Sigma map — they form a redundant pair the network cannot distinguish. The contract is therefore that no two entries (in any codebook, or between codebooks) should have identical `.what` prototype rows under the WhatEncoding mapping.

Current enforcement:

| Source | Mechanism |
|---|---|
| SymbolicSpace codebook | `ImpenetrableLayer` overlap penalty + variance floor (`Layers.py:4402`); five-relations classifi |
| ConceptualSpace codebook | Same `ImpenetrableLayer` available; not yet wired by default |

| Source | Mechanism |
|---|---|
| PerceptualSpace Lexicon | Cosine-margin pode/antipode SBOW training (Lexicon's "projective unit ball" — `x $\sim$ $` |
| InputSpace vocabulary | Shares PerceptualSpace's Lexicon (text mode); raw-data mode requires the data loader to ded |

The two axes are dual:

- `.where` uniqueness is **structural** — enforced at construction via the registry, never violated at runtime.
- `.what` uniqueness is **learned** — encouraged by the `ImpenetrableLayer` regularizer and the Lexicon's antipodal quotient, but a freshly-initialized codebook can have near-duplicate rows before training begins.

Together they guarantee that the parthood lattice the bivector activation chain rides on (see Monotonicity of the lift / lower chain) is well-formed: every slot has a distinct identity under both the positional key (`.where`) and the content prototype (`.what`).

---

## Lexicon (Projective Unit Ball)

The **Lexicon** (`bin/Layers.py`) is the vocabulary store that backs PerceptualSpace word embeddings and SymbolicSpace symbol prototypes. Each row is a vector $w_i$ in the **projective unit ball** — the closed ball $B^D = \{x : \|x\|_2 \leq 1\}$ with the **negation identification** $w \sim -w$, which realizes real projective space $\mathbb{RP}^D$. There is no edge to the embedding space; the boundary sphere is glued to itself by the $\pm$-quotient, and the pode / wrapped-pode pair structure SBOW training depends on holds at every $\|w\|$.

**Terminology pin** — the project distinguishes three notions that are sometimes conflated:

- **Pode** of a pair $(a, b)$: midpoint $(a + b)/2$, the natural attractor for SBOW positive-pair updates.
- **Wrapped pode**: alternative midpoint reached through the $\pm$-quotient, $(a - b)/2$ — the midpoint via the *negation* of $b$. Used to compute the shorter arc and to pick which representative of $b$ is closer to $a$.
- **Antipode** of a single point $p$: the furthest point from $p$ in the manifold's topology. SBOW uses the antipode as a *balancing repulsion target* (not as an equivalence partner). On the flat torus the antipode is unique ($\text{wrap}(p + 1)$ per axis); on $\mathbb{RP}^D$ it is **not** unique — the maximum-distance set is the orthogonal hyperplane $\{w : \langle p, w\rangle = 0\}$, a $(D{-}1)$-sphere — and an SBOW negative sample on the projective ball must pick a representative of that set.

So $-w$ is the **negation** of $w$ (the $\pm$-quotient partner), *not* the antipode of $w$.

### Distance

For $a, b \in B^D$ the projective squared distance is

$$d^2_{\mathbb{RP}}(a, b) \;=\; \min(\|a - b\|_2^2,\ \|a + b\|_2^2) \;=\; \|a\|_2^2 + \|b\|_2^2 - 2\,|\langle a, b\rangle|.$$

Equivalently: take the *pode* (midpoint) and the *wrapped pode* (midpoint through the negation of $b$),

$$\text{pode}(a, b) = \tfrac{1}{2}(a + b), \qquad \text{wpode}(a, b) = \tfrac{1}{2}(a - b),$$

and observe that $d_{\mathbb{RP}}(a, b) = 2{\cdot}\min(\|a{-}\text{pode}\,\|,\ \|a{-}\text{wpode}\,\|)$. The lookup just picks whichever representative of $b$ — namely $b$ or $-b$, the two negation-quotient reps of the same projective point — is closer to $a$. (Note: this *negation* representative is not the antipode of $b$; on $\mathbb{RP}^D$ the antipode is the orthogonal hyperplane.)

### Matmul-form lookup

For a fixed query $x$, sorting by smallest $d^2_{\mathbb{RP}}$ is exactly sorting by largest

$$\text{score}(x, w_i) \;=\; |\langle x, w_i\rangle| - \tfrac{1}{2}\|w_i\|_2^2.$$

Implementation: cache `W_norm2 = W.square().sum(-1)` once per optimizer step, then top-k is `(x @ W.T).abs() - 0.5 * W_norm2` followed by `torch.topk` — a dense matmul, an elementwise abs, and a broadcast subtract. No $V \cdot D$ outer-product intermediate.

The `Lexicon` API:

```
lexicon = Lexicon(V, D)              # default: projective unit ball
lexicon.project_unit_ball_()         # call after optimizer.step()
W_index, W_norm2 = lexicon.lookup_index()

# Projective (RP^D) - antipode-aware, default.
idx, dist_sq, scores = Lexicon.topk_rp(x, W_index, W_norm2, k=32)

# Plain L2 (each row independent) - for sites where w and -w are distinct.
idx, dist_sq, scores = Lexicon.topk_l2(x, W_index, W_norm2, k=32)

# Pairwise primitives (SBOW / chart parser):
Lexicon.rp_distance_sq(a, b)
Lexicon.rp_similarity(a, b)
Lexicon.rp_pode(a, b)
Lexicon.rp_wrapped_pode(a, b)
Lexicon.rp_closer_rep(a, b)          # sign(<a, b>) * b
```

For $V \gtrsim 10^5$, use `topk_rp_chunked` (or `topk_l2_chunked`) to bound the peak score-tensor size by `chunk_size` rows of $W$ instead of the full $(B, V)$ matrix.

### SBOW training: pode (attractor) and antipode (repulsion target)

SBOW uses two different notions of "balancing point" for a pair $(a, b)$:

- **Pode (attractor).** Positive-pair updates pull $a$ and $b$ toward their midpoint $\text{pode}(a, b) = (a + b)/2$. On the projective ball the gradient picks the closer of $b$ and $-b$ (via wpode) so the attraction is along the shorter arc.
- **Antipode (balancing repulsion target).** Negative-pair updates push the row away from the *furthest* point in the manifold — the antipode. This is what keeps the embedding from collapsing to a single cluster.

On the **flat torus** the antipode of $p$ is unique ($\text{wrap}(p+1)$ per axis). On the **projective unit ball** it is the $(D{-}1)$-dimensional orthogonal hyperplane $\{w : \langle p, w\rangle = 0\}$ — not a unique point — so SBOW must sample a representative orthogonal direction when it wants a single repulsion target.

The negative-sampling gradient under projective distance has two regimes by $\text{sign}\langle a, b\rangle$. When the inner product is positive, the gradient is the standard contrastive repulsion along $(a - b)$; when it is negative, the gradient pushes $a$ away from $-b$ along $(a + b)$ — i.e. it operates on the *negation* representative of $b$ rather than $b$ itself. This is the projective replacement for the torus's coordinate-wise wrap and preserves the pair structure SBOW depends on without requiring modular arithmetic.

After every optimizer step the trainer calls `lexicon.normalize()`, which clips row norms so $\|w_i\| \le 1$ stays invariant. `W_norm2` should be refreshed any time the weight matrix changes.

**Why this geometry, not the torus**

The earlier Lexicon used the flat torus $T^D = [-1, 1)^D$ with wrapped MSE distance $d_T^2(x, w) = D^{-1}\|\text{wrap}(w - x)\|_2^2$. That is homogeneous and edgeless — geometrically attractive — but exact lookup needs coordinatewise wrap and a $V \cdot D$ broadcast subtract that becomes the dominant cost at $V = 10^6$ (see the IR-mode profile and `test/bench_codebook_lookup.py`). The projective unit ball keeps the antipodal pair structure SBOW wants *and* lets the lookup compile to a single matmul.

The torus primitives (`Lexicon.wrap`, `Lexicon.delta`, `Lexicon.distance_sq`, `Lexicon.similarity`, `Lexicon.inner_distance`, `Lexicon.outer_distance`, `Lexicon.antipode`, `Lexicon.random`) and the `torus=True` constructor flag remain on the class as **legacy** static methods so call sites still operating in torus coordinates keep working during the transition. They are clearly marked `LEGACY (torus)` in their docstrings; new code must use the `rp_*` primitives.

---

## InputSpace

**Role.** Receives the raw source buffer from `Data()` and lifts it into the model's internal working dimensionality.

**Text mode – forward.** Delegates tokenization to `Lex`, which produces a span table of `(start, end, type)` entries – one span per token. Each span is converted to a vector with two components:

- `nWhat` dimensions – token content, encoded via `Basis` / `Codebook` (the word embedding lookup).
- `nWhere` dimensions – positional information derived from the character offset.

The result is a `[nActive, nWhat + nWhere]` tensor representing the tokenized sentence.

**Text mode – reverse.** Reconstructs the source string from the latent state by inverting the span encoding: each vector is mapped back to its nearest codebook entry (word embedding), then spans are reassembled into characters using the stored offset table.

**Numeric mode.** Tensor data is passed through unchanged; the LiftingLayer projects the native input dimension (e.g. 784 for MNIST) to the model's working dimensionality `nDim`. Non-embedding inputs are then scaled to `[-1, 1]` using the global data min/max, and the reverse path restores the original range.

**Key parameters.**

| Parameter | Description |
|---|---|
| `nActive` | Sequence length: maximum tokens per input |
| `nDim` | Output dimensionality per vector (set on TheObjectEncoding) |
| `nWhere` | Positional dimensions appended to each token vector |
| `nWhen` | Temporal dimensions appended to each token vector |
| `lexer` | Tokenization mode: `"word"` or `"sentence"` |
| `codebook` | Whether input values are discrete |
| `demuxed` | Store what/where/when independently (default: `false`) |

**Layer.** `LiftingLayer` – bridges native input dimension to `nDim`.

**Invertibility.** InputSpace is always non-invertible in the strict sense; the reverse path is a separate reconstruction procedure using the span table, not a matrix inverse.

**Document streaming and `valid_mask`.** Documents longer than `nOutput` bytes are not truncated. `TheData` maintains a per-row cursor `(doc_idx[b], offset[b])` and `next_tick()` returns `(input, output, hard_eos)` where `input` is a `[B, nOutput]` slab containing the next `nOutput` or fewer bytes from each row's current document and `hard_eos` is a host-side `[B] list[bool]` flag set on ticks where a row's cursor exhausts the current document. A short fill at document end NULL-pads the slab tail; the

stem's `valid_mask: [B, K] bool` flips False for the padded positions, and downstream state-mutation propagation (codebook EMA, parse-stack push, truth-layer record) skips them. Concatenating per-tick slabs across a row's document run reproduces the original bytes byte-exact. The `_lex_batch` truncation that silently dropped overlong inputs was replaced with an `assert n_tokens < nObj` in §8g of the brick-vectorization handoff; the cursor is responsible for sizing the slab to fit the lex's `nObj - 1` content slots.

**Cursor universal — trial mode for non-AR data.** The brick-vectorization handoff §8e unified the data path: `next_tick()` is the single dispatch interface for both AR text byte (the rolling-cursor case) and non-AR data (numeric, non-byte text). In trial mode (`slab_bytes` not set), each tick yields one batch of trials with `hard_eos = [True] * B` – the data cursor aligns with the trial, mirroring the legacy DataLoader contract. The runEpoch outer loop drives `ds.next_tick()` directly for both modes; the surrounding DataLoader exists only so existing tests can grab `loader.dataset`.

`_end_of_stream` is a host-side `list[bool]` diagnostic only — never consulted for control flow — under the brick-vectorization handoff §8c. The canonical hard-reset signal is the cursor's `hard_eos` list from `next_tick()`. The scalar/tensor variants of `_end_of_stream` were removed; the diagnostic list is sized lazily by the AR forward path and cleared per-row by `Reset(batch=b, hard=True)`. The legacy `InputSpace.batch_advances_sentence` stub property was deleted in §8a.

---

## PerceptualSpace

**Role.** Transforms raw input vectors into perceptual representations via multiplicative interactions. Models prototype-based feature detection: each percept is a product of local input features.

**Forward operation (log-space linear).**

```
s_i = log((1 + x_i) / (1 - x_i))          -- atanh domain transform
z_j = W @ s + b                             -- linear in log-multiplicative space
y_j = (exp(z_j) - 1) / (exp(z_j) + 1)     -- tanh back to [-1, 1]
```

The atanh/tanh domain transform gives multiplicative semantics: addition in log-space corresponds to multiplication of the original features.

**Reverse operation (invertible=True).** A single `PiLayer(invertible=True)` is shared for both directions. The reverse path inverts each step: `_to_mult(y)`, log, `W^{-1}(z - b)`, exp, `_from_mult`. The matrix inverse uses InvertibleLinearLayer (LDU factorisation) for exact inversion.

**Reverse operation (invertible=False, reversible=True).** Two separate `PiLayer` instances – `pi1` for forward, `pi2` for reverse – each with independent weights.

**Key parameters.**

| Parameter | Description |
|---|---|
| `nActive` | Number of active perceptual vectors |
| `nVectors` | Codebook size; enables VQ when > nActive |
| `invertible` | True: shared invertible layer; False: separate pi1/pi2 |
| `codebook` | False -> `.what` is a passthrough `Tensor`; True -> `Codebook` |
| `hasAttention` | Enable attention reweighting |
| `bivectorOutput` | When `true`, applies the Q2 promotion $(a_P, a_N) = (\max(0, x), \max(0, -x))$ to the per-slot percept event |

**Layer.** `PiLayer` (one or two instances depending on `invertible`).

**Range.** Vectors live in the signed hypercube `[-1, 1]^d` (tanh-bounded). The space is centered at the origin for geometric symmetry, though it has no negation operator – percepts represent feature magnitudes with sign indicating direction. Activation is `[-1, 1]`. Tanh is applied to enforce the range.

Under `bivectorOutput=true` the *per-slot scalar activation* is replaced by a non-negative paired-index pair $[aP, aN] \in [0,1]^2$. The signed-sum reduction `x = sum_d event[..., d]` is split via `aP = max(0, x); aN = max(0, -x)` – the canonical bitonic-to-bivector map (spec §Q2). Reverse recovers the per-slot scalar `aP - aN` and broadcasts uniformly across the input dim; per-feature detail within a slot is intentionally lossy (the prototype-content channel is the C tier).

**Invertibility.** `invertible=True`: shared layer, exact inverse. `invertible=False`: separate layers, approximate pseudoinverse in reverse.

---

## ConceptualSpace

**Role.** Transforms perceptual vectors into abstract concepts via additive linear operations. Models conceptual hyperplanes that partition perceptual space.

**Geometry contrast with the Lexicon.** ConceptualSpace is geometrically distinct from the projective-unit-ball Lexicon used by PerceptualSpace and SymbolicSpace. Its codebook stores **named directions** (unit-norm vectors $c_i \in S^{D-1}$); the *input* magnitude in $[-1, +1]$ encodes *belief certainty* with sign, and that magnitude must survive into the similarity score. So:

- **Lexicon** ($\mathbb{RP}^D$). Each row is a vocabulary point in $B^D/(x \sim -x)$. Score is $|\langle x, w_i\rangle| - \frac{1}{2}\|w_i\|^2$. Antipodal identification is the *defining* invariant — `word` and its antipode collapse to the same lookup result.
- **ConceptualSpace.** Each codebook row is a unit-norm direction in $S^{D-1}$ (no antipodal identification). Score is $\langle x, c_i\rangle$ — a single matmul with no abs and no norm correction because the codebook is constant-norm. Sign matters: $x \cdot c_i = +1$ affirms concept $i$, $-1$ denies it, 0 is orthogonal. Cosine similarity would divide out the certainty signal and is therefore *not* used here.

The PiLayer below operates on the input side of this boundary; the codebook geometry above governs how concepts are looked up *as* named directions.

**Owned layer.**

| Layer | Direction | Math | Notes |
|---|---|---|---|
| `self.pi` (`PiLayer`) | P ↔ C | log-domain multiplicative, monotonic | `forwardPi` (P → C) and `reversePi` (C → P) poi |

ConceptualSpace owns one PiLayer that handles both directions of the P↔C boundary via its own self-inverse. When the space is reversible without an invertible layer, two PiLayers (`pi1`, `pi2`) are constructed and `forwardPi` / `reversePi` route to them; otherwise a single `pi` serves both.

**Binary forward (Step 5).** `self.pi.forward` accepts a `binary=True` flag that hard-selects the top-2 input operands by $|x_i|$ (via `Ops.top2_select_ste`) and zeros the rest before the log-domain fold. Zero is the multiplicative identity for Pi's AND so unselected operands drop cleanly out of the pool. Backward is straight-through: every input dim retains a learning signal so the layer learns to make all candidates sensible, even when they aren't currently in the top-2. Used by grammar dispatch where a binary rule combines the two most-active constituents.

**Forward operation (P → C).** PiLayer's log-domain product fold:

```
m   = (1 + x) / (1 - x)              # (-1,1) -> (0, inf)
y   = tanh(W * log(m) / 2 + b)       # back to (-1, 1)
```

The `monotonic` flag constrains $W \geq 0$ so the AND fold preserves ordering; `nonlinear=True` keeps the output in `[-1, 1]`.

**Reverse operation (invertible=True).** Exact log-domain inverse via `InvertibleLinearLayer` (or `NonNegativeInvertibleLinearLayer` when `monotonic=True`):

```
l   = log((1 + y) / (1 - y))
x   = tanh(W^{-1} * (l - b) / 2)
```

**Reverse operation (invertible=False, reversible=True).** Two separate `PiLayer` instances – `pi1` for forward, `pi2` for reverse.

**Key parameters.**

| Parameter | Description |
|---|---|
| `nActive` | Number of active concept vectors |
| `nVectors` | Codebook size |
| `invertible` | True: shared invertible layer; False: separate sigma1/sigma2 |
| `hasAttention` | Enable attention |
| `hasNorm` | Enable layer normalization |

**Layer.** `PiLayer` (one or two instances depending on `invertible`).

**Range.** Vectors are tanh-bounded: each element is in `[-1, 1]` (applied by SigmaLayer). The boundary between PerceptualSpace and ConceptualSpace uses `atanh` (forward) / `tanh` (reverse) as exact inverses. Tanh is applied to enforce the element-wise range.

**Activation carrier.** `ActiveEncoding.nDim = 2`: activation is a 2-dim bivector `[aP, aN]` per position, encoding the four corners of the tetralemma (*catuskoti*):

| State | `[aP, aN]` |
|---|---|
| TRUE (*asti*) | `[1, 0]` |
| FALSE (*nasti*) | `[0, 1]` |
| BOTH (*ubhaya*) | `[1, 1]` |
| NEITHER (*anubhaya*) | `[0, 0]` |

BOTH encodes first-class inconsistency (same position affirmed and negated by independent sources/frames); NEITHER encodes unknown (neither affirmed nor negated). Operations obey De Morgan under pole-swap negation $\neg[aP, aN] = [aN, aP]$:

- Conjunction: `[min(aP, bP), max(aN, bN)]`
- Disjunction: `[max(aP, bP), min(aN, bN)]`

See BuddhistParallels.md.

**Invertibility.** `invertible=True`: exact inverse via atanh + `W^{-1}`. `invertible=False`: separate layers with independent weights.

**MASK on `SubSpace._active`.** `SubSpace` tracks two orthogonal per-position tensors: `activation` (the 4-valued truth bivector above) and `_active` shaped `[B, N, M]` where $M$ is the number of modality presence flags (what / where / when). Grammar-rule masking is shape- disambiguated by `_apply_mask`:

| Mask shape | Effect |
|---|---|
| Aligns with `out.shape[-1]` (feature axis) | Element-wise multiply on the output tensor. |
| Aligns with `out.shape[-2]` (position axis) | Zero the masked rows of `subspace._active`; `materialize()` then gates those |

This makes MASK a first-class filter on the presence flags rather than an arithmetic multiplication, so masked positions propagate their "absent" status through the pipeline's active-materialization path.

## SymbolicSpace

**Role.** Converts continuous concept activations into a discrete set of active symbols. This is the information bottleneck of the pipeline: rich perceptual-conceptual representations are compressed into a sparse presence pattern over a codebook of symbol prototypes.

**Owned layer.**

| Layer | Direction | Math | Notes |
|---|---|---|---|
| `self.sigma` (`SigmaLayer`) | C ↔ S | atanh-domain additive, monotonic | `forwardSigma` (C → S) and `reverseSigma` (S |

SymbolicSpace owns one SigmaLayer that handles both directions of the C↔S boundary via its own self-inverse, isomorphic to `ConceptualSpace.pi`. When the space is reversible without an invertible layer, two SigmaLayers (`sigma1`, `sigma2`) are constructed and `forwardSigma` / `reverseSigma` route to them; otherwise a single `sigma` serves both. The legacy `self.layer` PiLayer attribute is gone – consumers use `model.symbolicSpace.sigma` directly.

`self.sigma.forward` accepts the Step-5 `binary=True` flag (see ConceptualSpace section).

**Symbols are percepts.** Each symbol represents the presence (`1`) or absence (`0`) of a named entity. Since conceptual activations range `[-1, 1]`, the mapping between the two domains is `symbol = (activation + 1) / 2`. `SubSpace.get_symbols()` and `set_symbols()` perform this conversion; SymbolicSpace uses them exclusively instead of raw `get_activation` / `set_activation`.

See Language.md for the full language system design.

**Forward operation (codebook=True).**

1. Extract concept activation `[B, nConcepts]` from the input subspace.
2. Map through `SigmaLayer(nConcepts, nSymbols, invertible=True, monotonic=True)` to `[B, nSymbols]`.
3. Store as symbolic presence `[0, 1]` via `set_symbols()`.
4. Reshape to `[B, nSymbols, 1]` (each symbol is 1-dim) and pass through the codebook, which quantizes and produces a one-hot activation over codebook entries weighted by similarity.

The output is a one-hot encoding over the codebook. The codebook provides dense vectors for downstream spaces that require `[B, N, D]` tensors.

**Forward operation (codebook=False).** The SigmaLayer maps the activation to symbolic presence via `set_symbols()`; vectors pass through from the input subspace unchanged.

**Reverse operation.** Reads symbolic presence via `get_symbols()`, then the PiLayer's exact inverse maps `[B, nSymbols]` back to `[B, nConcepts]`, recovering the concept activation.

**Key parameters.**

| Parameter | Description |
|---|---|
| `nActive` | Total number of symbols (output + reconstruction symbols) |
| `nVectors` | Codebook size (= nSymbols when codebook=true) |
| `codebook` | Enable codebook quantization (required for one-hot output) |
| `bivectorOutput` | Forward returns the per-prototype catuskoti bivector $[B, V_S, 2]$ via `Codebook.forward(..., project=T` |

**Range.** Symbols are percepts: each symbol represents the presence (`1`) or absence (`0`) of a named entity and lives in `[0, 1]`. One symbol is encoded at a time (the most highly active; negative products never activate). Each symbol receives where/when encoding from PerceptualSpace, making symbols uniform with percepts. Internally stored as activation in `[-1, 1]` via the `(x+1)/2` mapping.

**Codebook shape (post-2026-05-07 rollback).** The symbol codebook has one row per symbol at the natural `nDim` width: `SymbolicSpace.subspace.what.getW().shape == (nVectors, nDim)`, with `nWhat == nDim` (no leading [pos, neg] bivector pinned in the codebook). Each row is a free coefficient vector over the conceptual axes – "how much of concept_i is this symbol?" – so under V_S = V_C alignment one row corresponds to one concept. The codebook is *not* unit-norm: symbol prototypes are free patterns, retrieved via Euclidean L2 (`use_dot_product = False`).

The per-prototype catuskoti bivector `[B, V_S, 2]` (TRUE=[1,0], FALSE=[0,1], BOTH=[1,1], NEITHER=[0,0]) lives on `subspace.activation`, NOT in the codebook. Populated by `Codebook.forward(input, project=True)` – the **intrinsic snap**:

```
pos[b, n] = sum_v relu(input[b, v] $\cdot$ W[n])
neg[b, n] = sum_v relu(-input[b, v] $\cdot$ W[n])
```

The matching decode `Codebook.reverse(bivec, project=True)` is the cached SVD pseudo-inverse: `C $\to$ S $\to$ C` round-trip projects the input onto span(W) and is a fixed point thereafter (verified by `test/test_idempotent_loop.py`). Where/when ride alongside the encoding on the per-batch muxed event tensor; they don't live inside the codebook itself.

The C ↔ S boundary as **categorization**: calling `SymbolicSpace.forward` IS the act of *naming* — projecting the incoming concept activation onto the named codebook lattice. The snap loss IS the categorization: a clean prototype match yields a TRUE-corner activation; a noisy near-prototype match degrades the TRUE pole; an off-lattice input collapses to NEITHER. The decode is not lossless reconstruction — it's projection onto the codebook subspace, which is what categorization means architecturally. Grammar operators (NOT, AND, OR, lift, lower, …) operate on the bivector activation between snap calls; the dialectic loop is snap (synthesis) → grammar (logic) → decode (analysis) → next pass.

See BuddhistParallels.md for the catuskoti mapping, Logic.md for the bivector operator algebra, and Mereology.md for how the conceptual measures (Area, Luminosity, Contiguous, Continuous, Peaceful) read concept activations directly rather than the symbol codebook.

**Layer.** `PiLayer(nConcepts, nSymbols, invertible=True, monotonic=True)` – maps between activation spaces via monotonic multiplicative transform. The `monotonic=True` constraint ensures weights $W \geq 0$, preserving ordering. Exact inverse via the internal `InvertibleLinearLayer` (LDU factorisation).

**Hierarchical mode.** When `<useButterflies>true</useButterflies>` or `<useGrammar>all</useGrammar>`, BasicModel stores per-stage ConceptualSpace/SymbolicSpace instances in `self.conceptualSpaces` and `self.symbolicSpaces`. Butterfly mode passes butterfly-aware Pi/Sigma layers into those spaces. The symbol dimension is geometrically partitioned so each order writes only to its designated slice. The planned `shamathaSpeech` mode is a narrow DNF-object policy rather than the full grammar hierarchy. See Reasoning.md Section Architecture Modes.

**Invertibility.** Exactly invertible via the PiLayer's reverse path.

---

## Monotonicity of the lift / lower chain

Under the bivector regime (`<bivectorOutput>true</bivectorOutput>` on all three spaces), the percept → concept → symbol chain is an **order-preserving map on a positive cone** – positive linear maps on non-negative paired-index activations preserve the parthood partial order all the way through.

The triple that makes this work:

1. **Activations live on the positive cone** `[0, 1]^{2K}` (paired-index bivector). PerceptualSpace's Q2 promotion `(aP, aN) = (max(0, x), max(0, -x))` (spec §Q2) is the bitonic-to-bivector entry point; from there every space writes only non-negative `[aP, aN]` pairs. The componentwise partial order $\leq$ on this cone *is* the parthood order: $s_1 \leq s_2$ componentwise ⇔ "every pole of evidence in `s1` is dominated by the same pole in `s2`" ⇔ `s1` is part of `s2`.

2. **The Pi / Sigma maps are restricted to** $W \geq 0$ entry-wise (`monotonic=True` selects `NonNegativeInvertibleLinearLayer` under invertibility, `NonNegativeLinearLayer` otherwise). Positive matrices are precisely the monotone operators on the positive cone:

$$a \leq b \text{ componentwise} \Longrightarrow Wa \leq Wb \text{ componentwise}$$

3. **Therefore Pi / Sigma preserve parthood pole-by-pole.** Lifting a set of percepts into a concept (positive linear combination via ConceptualSpace's PiLayer) can only *increase* the bivector pole-by-pole, matching the parthood semantics: a whole contains its parts. Lowering does the same in the other direction.

The bivector layout is what keeps the contradiction corner `[1, 1]` distinct from the ignorance corner `[0, 0]` under positive matmul – a bitonic `[-1, 1]` scalar would let `aP - aN` cancel under summation, collapsing both to zero (the §B2 forbidden collapse). Splitting the two poles onto independent non-negative axes is what lets the positive matmul be both order-preserving *and* contradiction-preserving.

The same-order regularization on each codebook (`ImpenetrableLayer` at S; planned at C / P) maintains an antichain of same-rank prototypes, so the chain's parthood lattice carries cross-order dominance only – siblings at a given `<conceptualOrder>` step remain mutually incomparable. The antichain property is one half of the broader Codebook Uniqueness Contract: the `.what` axis must distinguish every prototype within and across codebooks for the parthood lattice to be well-formed, complementing the structural `.where`-uniqueness enforced by the codebook offset registry.

See Logic.md §Parthood as Projection for the parthood formula and BuddhistParallels.md for the catuskoti / tetralemma mapping the bivector encodes.

---

## SyntacticSpace

**Role.** Generates a binary derivation tree (deep structure) from the set of active symbols produced by SymbolicSpace. Words are concepts encoding grammatical rules. The derivation is a Chomsky Normal Form (CNF) grammar stored as word tuples `(batch, vector, rule)` on the output subspace.

See Language.md for the grammar, word encoding, and open questions about differentiable tree structure.

**Forward operation.**

1. Read symbolic presence via `get_symbols()` (nonzero entries are present symbols).
2. For N present symbols per batch, generate a CNF derivation: N-1 binary rules (randomly selected) + 1 terminal (S → W).
3. Store the derivation as word tuples on the output subspace. Vectors and symbolic presence pass through unchanged via `set_symbols()`.

**Reverse operation.**

1. Walk the word list: every `(batch, vector, rule)` entry marks that position as present.
2. Reconstruct the symbolic presence vector deterministically from the derivation via `set_symbols()`.

The round-trip `forward -> (delete symbols) -> reverse` recovers the original present positions exactly.

**Key parameters.**

| Parameter | Description |
|---|---|
| `nActive` | Number of active word vectors |
| `nDim` | Word vector dimensionality |
| `nVectors` | Codebook size (defaults to nActive) |

**Layer.** No trainable parameters (rule selection is currently random).

**Invertibility.** Reverse deterministically recovers activation from the derivation tree.

---

## OutputSpace

**Role.** Maps symbolic (or syntactic) vectors to task targets via a linear projection. This is the final prediction stage.

**Forward operation.** Linear projection from symbol dimensionality to output dimensionality:

```
y = W_out * x + b_out
```

Always uses `reshape=True` so the `[B, nSymbols, symbolDim]` tensor is flattened before projection.

**Reverse operation.** Projects output predictions back to symbol space via the pseudoinverse of `W_out`. In text mode, snaps each output vector to the nearest entry in the codebook (nearest-neighbour lookup) to recover a discrete token, then assembles tokens into a string.

**getEmbeddedIO() override.** OutputSpace overrides `getEmbeddedIO()` to return the raw target dimensions rather than the encoded dimensions. This ensures the loss is computed in the output vocabulary space, not the embedding space, regardless of any encoding overhead.

**Text mode generation.** In autoregressive language model mode, OutputSpace iterates token-by-token: at each step the predicted token vector is snapped to its nearest codebook entry, that entry is fed back as input, and generation continues until `maxResponseLength` is reached or an end-of-sequence token is produced.

**Key parameters.**

| Parameter | Description |
|---|---|
| `nActive` | Number of output values (e.g. 1 for XOR, vocab size for LM) |
| `nDim` | Output vector dimensionality |
| `nVectors` | Codebook size (defaults to nActive) |

**Layer.** `LinearLayer` with `(bias, temp)` support for ergodic mode. Always `reshape=True`.

**Range.** The forward pass rescales output from `[-1, 1]` (symbolic activation range) to the original data range via `Data.denormalize()`. The reverse pass applies `Data.normalize(x, "output")` to map back to `[-1, 1]` before the reverse linear projection.

**Invertibility.** Reverse uses pseudoinverse; not exactly invertible in general.

# Language

## Overview

The language system is a soft-superposition CKY chart parser that runs over symbol activations. Three pieces cooperate:

- `Grammar` (in `bin/Language.py`) — singleton rule table loaded from XML or `data/grammar.cfg`.
- `Chart` (`Language.py`) — owns the chart parameters, runs the inside / outside passes, and dispatches each per-cell rule application through `wordSpace.host_layer(tier, rule_name)`.
- `GrammarLayer` subclasses (in `bin/Layers.py`) — one class per grammar operator (`not`, `non`, `intersection`, `union`, `conjunction`, `disjunction`, `lift`, `lower`, `equals`, `part`, `true`, `false`, `swap`, `query`). Each class exposes `forward()` / `reverse()` (the unary `Layer` interface) and, for binary ops, `compose()` / `generate()` (the chart's binary tensor interface).

```
InputSpace  ->  PerceptualSpace  ->  ConceptualSpace  ->  SymbolicSpace  ->  OutputSpace
                                                          chart runs here
```

### Tiers (P / C / S / L)

Each rule and each layer carries a tier tag. The four tiers map to distinct activation domains:

| Tier | Owner | Activation domain |
|---|---|---|
| `P` | PerceptualSpace | bivector `[B, V, 2]` per percept |
| `C` | ConceptualSpace | bivector `[B, V, 2]` pre-codebook |
| `S` | SymbolicSpace | scalar `[B, V]` post-codebook (monotonic) |
| `L` | (logical, none) | bivector `[..., 2]` – pure lattice primitive |

S-tier ops compose **monotonic** functions over the post-codebook scalar activation (`effective_activation()`: bivector poles reduced via `max` and gated by modal presence). L-tier ops (`intersection`, `union`) are pure lattice min/max on the bivector activation; they aren't owned by any space. The chart binds an L-tier op at whichever space's tier the operands live in (`intersection(C, C)` binds at C; the L tag is layer-side classification, not a routing tier).

History: the 2026-04-19 merge collapsed the C-tier into S and removed the P-tier syntactic layer (chunking became a non-syntactic pre-pass). The 2026-05-01 refactor moved per-rule math out of `SyntacticLayer` onto `GRAMMAR_LAYER_CLASSES`. The 2026-05-04 cleanup retired the `Fusion` / `Contiguous` / `what` / `where` / `when` / `absorb` operators (Fusion duplicated DisjunctionLayer; the slot selectors were dispatcher concerns; absorb became a base-class sentence marker). The 2026-05-05 directive split lattice-min/max into the new `L` tier and pinned S-tier conjunction/disjunction to the post-codebook scalar activation domain.

Per-space dispatchers are `SyntacticLayer` instances built by `build_space_syntactic_layer`; each holds a tier-specific `host_layers` dict keyed by `rule_name`. The chart authority role (rule-firing probability gating via `GrammarLayer.gated_run`) lives on `Chart` itself — `WordSpace.__init__` installs the chart as `GrammarLayer._chart_authority` and hands it the `Grammar` reference; the legacy module-global `SyntacticLayer` that previously held this responsibility was retired 2026-05-08. The dispatcher's `_read_subspace` / `_write_subspace` honor each layer's `reads_activation` flag: `True` reads/writes the activation field (bivector at C-tier, scalar at S-tier), `False` reads/writes the muxed event tensor.

---

## Grammar

`TheGrammar` is a singleton `Grammar` instance. Production configs load `data/grammar.cfg` through `<WordSpace><language><grammarCfg>...`, which is the authoritative rule table. An XML `<grammar>` block inside `<WordSpace><language>` is the legacy alternative; both paths populate the same `Grammar.rules` list.

A `RuleDef` namedtuple carries `(tier, canonical, arity, method_name, lhs, rhs_symbols)`. `method_name` is the op name from the explicit-op / function-call RHS (`'lift'`, `'intersection'`, `'not'`, …), or `'merge'` for bare-symbol-sequence rules, or `'emit_head'` for the downward `S -> C` projection.

### Accessors

- `Grammar.symbolic()` — indices of all S-tier rules
- `Grammar.symbolic_transition()` — `None` (no cross-tier transitions)
- `Grammar.binary_rules()` — indices of all arity-2 rules
- `Grammar.rule_by_id(i)` — canonical production string
- `Grammar.method_name(i)` — dispatch method name (e.g. `'swap'`)
- `Grammar.arity(i)` — 1 or 2
- `Grammar.tier(i)` — `'P'` / `'C'` / `'S'` / `'L'`

### Grammar configuration

`Grammar.configure()` loads an XML `<grammar>` section. The block splits into `<upward>` (parsing / chart compose), `<downward>` (generation from a deep symbolic state), and `<start>` (the start symbol).

The `<start>S</start>` element is the canonical sentence boundary marker. When the chart reduces a row's parse to a single node of the start category, the layer emits a host-side soft-reset signal for that row. The outer doc-streaming loop drains the signal each tick and dispatches `wordSpace.soft_reset(batch=b)`, which clears every per-sentence working buffer for row `b` (parse stack, category stack, reconstruction stack, `_last_svo[b]`, `_stm_fired[b]`). Discourse history (`InterSentenceLayer` ring buffer) and codebook EMA are *not* cleared — they are document-scoped. Hard reset (document boundary) wipes both.

The loader accepts three RHS forms; the **explicit-op** form is the canonical target:

```
<grammar>
  <upward>
    <!-- (1) Explicit-op form. RHS is the dispatched op call;       -->
    <!--     LHS may be a comma-separated tuple for multi-output    -->
    <!--     reverses.                                               -->
    <rule>S       = lift(NP, VP)</rule>
    <rule>NP      = intersection(AP, NP)</rule>
    <rule>S       = not(S)</rule>
    <rule>S,S     = intersection_inv(VO)</rule>     <!-- multi-output reverse -->

    <!-- (2) Function-call form. Tag = LHS, body = op(args).         -->
    <S>not(S)</S>
    <S>swap(S, S)</S>
    <S>absorb(S, S)</S>

    <!-- (3) Bare-symbol form (transitional). Dispatches via         -->
    <!--     method_name='merge'.                                    -->
    <S>S VP</S>
    <NP>AP NP</NP>
  </upward>
  <downward>
    <!-- One-shot head emission via method_name='emit_head'.         -->
    <S>C</S>
  </downward>
</grammar>
```

The op names in the explicit-op RHS are dispatch keys into `GRAMMAR_LAYER_CLASSES` (Layers.py); `lift`, `lower`, `intersection`, `union`, `not`, `non`, `conjunction`, `disjunction`, `equals`, `part`, `true`, `false`, `swap`, `query`. Reverse productions are *derived mechanically* at grammar-load time — for each forward rule `LHS = op(arg1, arg2)` the loader synthesizes `arg1, arg2 = opReverse(LHS)` (multi-return).

Retired ops (do not use in new grammars):

- `Fusion` / `Contiguous` — duplicate of `disjunction` at S-tier; migrate to `disjunction(S, S)`.
- `what` / `where` / `when` — slot partitioning is the dispatcher's responsibility, not a grammar rule.
- `absorb` — sentence-marker behavior moved to `GrammarLayer.absorb(left, right) -> left` on the base class.

### `data/grammar.cfg` dispatch

The text-based rule table is line-oriented:

```
# comments start with '#'
[upward]
```

```
S = NP                        # PROJECT (single-category RHS)
S = lift(NP, VP)
NP = intersection(AP, NP)
S = not(S)
S = swap(S, S)
S = absorb(S, S)
...

[downward]
C = emit_head(S)
```

Sections are bracketed (`[upward]` / `[downward]`). Each `LHS = body` rule's body is either a function call `op(arg1, arg2)`, a single category (PROJECT), or `epsilon`. Loading is gated via `<WordSpace><language><grammarCfg>` in the XML config — when set, `Grammar._ensure_configured()` calls `Grammar.load_from_cfg(...)` and skips the legacy `<grammar>` block.

#### Syntactic sugar consumption

Under the byte-level lex routed through BPE chunking, raw text positions include syntactic sugar — runs of whitespace, single punctuation marks, newlines. After chunking these become their own word slots that the grammar must reduce away. The post-rolling-cursor grammar adds one explicit absorption rule: `absorb(A, B)` returns `A` and discards `B`. Initial logits should bias against `absorb` to avoid early-training collapse where the parser learns to discard everything past the first token.

#### Shamatha speech mode

Shamatha speech is the one-pointed speech mode, configurable via `useGrammar="shamathaSpeech"` (with `thoughtFree` kept as a compatibility alias). It is *not* serial mode; serial mode constrains the runtime cursor to a moving prefix / next-token task. Shamatha speech may conjoin and disjoin over all active percepts in the current object field at once — its constraint is that the logical object remains single-pointed.

The narrow grammar is the DNF object grammar:

```
literal  := polarity? concept
polarity := "" | "non-" | "not"
term     := literal ("and" literal)*
object   := term ("or" term)*
sentence := object
```

For this grammar to express a single object rather than a scattered aggregate, every logical merge must preserve spatiotemporal contiguity: `where(A)` and `where(B)` must overlap, touch, or be connected by an allowed adjacency edge before any merge may form a larger object; `when(A)` and `when(B)` must overlap or be temporally adjacent. See plans/2026-04-28-shamatha-speech-contiguity-handoff.md.

The polarity surface (`""`, `non-`, `not`) maps to the three channels of `NegationLayer(ternary=True)`: affirmation `x`, non-affirming negation `non(x)`, and affirming negation `-x`.

---

## Operator semantics

Every grammar op has one explicit computation anchor: **Sym** (call a GrammarLayer's tensor kernel directly), **SigmaPi** (level-crossing fold via `SS.forward(CS.forward(...))` and its inverse), **Def** (definitional WordSpace update over the symbolic codebook / mereological graph), or **Percepts** (perceptual pre-processing — chunking, before the grammar sees symbol slots).

### Notation

Let $x, \ell, r \in \mathbb{R}^{B\times N}$ (activation mode) or $\mathbb{R}^{B\times N\times D}$ (vector mode). Let $\mathcal{B}$ be the subspace basis (a bitonic `Basis` or codebook), supplying pointwise primitives `pos`, `negation`, `conjunction`, `disjunction`, `equal`, `part`. When $\mathcal{B}$ is absent the kernels fall back to torch elementwise primitives. An optional concept-axis `mask` gates the output by zeroing non-selected dimensions.

### Ternary commitment ops (`true` / `false` / `non`)

For $x \in [-1, 1]$ (clamped on entry):

$$\text{true}(x) = \max(0,\ x), \qquad \text{false}(x) = \max(0,\ -x), \qquad \text{non}(x) = 1 - |x|$$

These three partition unity: true + false + non = 1 pointwise. The ReLU shape of `true` / `false` makes them the "committed yes" and "committed no" halves of a bitonic axis; `non` is the triangular indeterminate residual peaked at $x = 0$. All three are lossy.

### Negation (`not`) — Sym

`not` is the propositional bivector swap on the leading two dims of a symbol vector: $[x_0, x_1, x_{2..}] \to [x_1, x_0, x_{2..}]$. Pure antipodal flip on the bitonic axis; self-inverse.

### Conjunction / disjunction — S-tier (post-codebook scalar)

`S = conjunction(S, S)` and `S = disjunction(S, S)` operate on the **post-codebook scalar activation**: a `[B, V]` tensor where `V` indexes prototypes in the symbolic codebook and each entry is the non-negative strength of that prototype after the codebook snap. Concretely, `materialize(mode='activation')` returns `effective_activation()` – the bivector poles `[pos, neg]` reduced via `max(pos, neg)` and gated by modal presence.

Because the activation is non-negative scalar, both ops are **strictly monotonic** lattice primitives:

$$\text{conjunction}(\ell, r) = \min(\ell, r), \qquad \text{disjunction}(\ell, r) = \max(\ell, r)$$

(routed through `Ops.intersection(..., monotonic=True)` and `Ops.union(..., monotonic=True)`, which collapse to `torch.min` / `torch.max` via `_lower_kernel(kind='strict')` / `_lift_kernel(kind='strict')`). Both are lossy with `(parent, parent)` pseudo-inverse on `reverse`.

### Intersection / union — L-tier (lattice on bivectors)

`L = intersection(L, L)` and `L = union(L, L)` are pure logical primitives: lattice min/max on a **bivector activation** `[..., 2]`. The L tier means the layer isn't owned by any single space – the chart binds it to whichever space's activation tier the operands live in. In practice the upstream tier is C (where activation is a `[B, V, 2]` bivector pre-codebook); the same kernel works at S when the chart sources S-tier bivector activation, but S-tier production grammars typically use `conjunction` / `disjunction` on the post-codebook scalar instead.

The kernels accept a `monotonic` kwarg:

| `monotonic` | kernel |
|---|---|
| `False` | RadMin / RadMax — same-sign min / max magnitude with zero passthrough |
| `True` | strict lattice min / max (per channel) |

L-tier ops are lossy: `decompose` returns `(parent, parent)` as the pseudo-inverse. When the upstream Basis carries codebook weights, callers can substitute `Ops.lowerReverseAll(Y, W, monotonic)` / `Ops.liftReverseAll(Y, W, monotonic)` for a codebook-search recovery instead of the identity pseudo-inverse.

### Part / equals — Def (currently tensor scoring)

Parthood weights the right operand by its containment in the left: $\text{part}(\ell, r) = \frac{\max(0,\ \ell \cdot r)}{|\ell|\,|r|} \cdot r$. The scalar score is broadcast to $r$'s rank. Empty-operand contract: if $|\ell|$ or $|r|$ is near zero, the score is 1 (the empty set is part of everything).

`equals(\ell, r)` is mutual parthood: $\text{part}(\ell, r) \cdot \text{part}(r, \ell)$, scalar-reduced and broadcast to $r$. Under a `mask`, equals degenerates to an L1-distance agreement score on the selected dims. Both are lossy. The target architecture moves both from tensor scoring to graph updates in WordSpace; see Mereology.md.

### Slot selectors (`what` / `where` / `when`) — RETIRED

Retired 2026-05-04: subspace partitioning is the dispatcher's responsibility (the modality tensors carry the slot structure intrinsically), not a grammar rule.

### Absorb — base-class marker

`absorb(\ell, r) = \ell`. Binary left-pass; the right operand is sugar that the rule predictor opted to discard. No longer a dedicated layer class – the marker lives on `GrammarLayer.absorb(left, right) -> left` so any binary subclass can flag its right operand as syntactic sugar without adding a distinct dispatch entry.

### Query — Mereological

`query(\ell, r) = \ell` (accumulator preservation). The query marker is pushed onto WordSubSpace at the chart's accumulation point when a norm-drop fires (see Composition below).

### Swap (Sinkhorn soft permutation) — Sym

A learned $3 \times 3$ logit matrix $M$ is normalized via $k = 5$ Sinkhorn iterations to produce a doubly-stochastic soft permutation $P$. Given operands $\ell, r$ and a learned broadcast marker $m$, the output is the first row of $P \cdot [\ell, r, m]^\top$. Lossy.

### Lift / lower — SigmaPi (mode-dispatch ops)

`lift` and `lower` are unified mode-dispatchers. Default modes are `lift` → `'OR'` (synthesis, $\vee$) and `lower` → `'AND'` (analysis, $\wedge$). The forward bodies follow the table above under conjunction / disjunction; region-form inputs (tuples $(\ell, u)$) emit the obvious bound intervals.

Cross-mode delegation: `lift(mode='AND')` forwards to `lower(mode='AND')`; `lower(mode='OR')` forwards to `lift(mode='OR')`. `mode='NOT'` is `Ops.negation(X1)` on either dispatcher.

`Ops.lowerReverseAll(Y, W, monotonic)` and `Ops.liftReverseAll(Y, W, monotonic)` return the multi-operand pair `(recovered_left, recovered_right)`. The single-operand analytic forms $\text{Ops.liftReverse}(\text{result}, \text{right}) = \text{result}/(\text{right} + \varepsilon)$ and $\text{Ops.lowerReverse}(\text{result}, \text{right}) = 2 \cdot \text{result} - \text{right}$ are preserved as legacy callers' inverse of the old $\text{Ops.lift} = \odot$ and $\text{Ops.lower} = \text{mean}$ bodies.

The cfg-driven grammar dispatch invokes `Ops.lift` / `Ops.lower` directly. The learnable equivalents live on `SymbolicSpace.sigma.forward` / `ConceptualSpace.pi.forward`, which own their own log-domain (Pi) and atanh-domain (Sigma) matmul bodies. The bridge is the `binary=True` flag on `PiLayer.forward` / `SigmaLayer.forward`: it pre-applies `Ops.top2_select_ste` to hard-select the top-2 input operands before the layer body runs.

### Chunking — Percepts

Chunking is not a grammar rule. It is a perceptual pre-pass implemented by `ChunkLayer` / `PerceptualSpace.chunk_static`, before the grammar sees symbol slots. No `chunk` entry exists in `GRAMMAR_LAYER_CLASSES`.

### Mereological suite — see Mereology.md

The mereology suite (`part`, `whole`, `equal`, `overlap`, `underlap`, `boundary`, `copart`) lives in `Ops.*`; the canonical specification of its semantics is Mereology.md. Each member has a vector form (default) and a scalar form (`scalar=True`). `part` and `equals` are surfaced through grammar dispatch as `PartLayer` / `EqualsLayer`; the rest are utility-level.

Empty-set conventions (parthood scalar form): $\text{part}(\emptyset, y) = 1$, $\text{part}(x, \emptyset) = 0$, $\text{part}(\emptyset, \emptyset) = 1$.

---

## GrammarLayer Implementations

`GrammarLayer` (in `bin/Layers.py`) is the base class for layers that implement grammar rule operators. Each subclass declares `rule_name`, `arity`, `invertible`, and `lossy` as class attributes, then overrides `forward()` (and `reverse()` if `invertible`). Binary subclasses additionally override `compose(left, right)` and `generate(parent)` for the chart's binary tensor interface; the defaults route arity-1 layers through `forward` / `reverse` and raise on arity-2 unless overridden.

The hardcoded module-level `GRAMMAR_LAYER_CLASSES` dict at the bottom of the section maps `rule_name` to subclass. Per-space dispatchers look up rules through this table when the grammar references a rule whose host parametrized layer isn't already owned by the space.

### Base contract

```python
class GrammarLayer(Layer):
    rule_name  = ""
    arity      = 1
    invertible = False
    lossy      = False
    _chart_authority = None  # set via set_chart_authority(syntactic_layer)

    def gated_run(self, x, fn, *fn_args, **fn_kwargs):
        auth = GrammarLayer._chart_authority
        if auth is None or not self.rule_name:
            return fn(x, *fn_args, **fn_kwargs)
        try:
            p = float(auth.should_run_rule(self.rule_name))
        except Exception:
            return fn(x, *fn_args, **fn_kwargs)
        if p >= 1.0 - 1e-9:
            return fn(x, *fn_args, **fn_kwargs)
        if p <= 1e-9:
            return x
        return p * fn(x, *fn_args, **fn_kwargs) + (1.0 - p) * x

    def compose(self, *operands):
        if self.arity == 1:
            return self.forward(operands[0])
        raise NotImplementedError
```

```
    def generate(self, parent):
        if self.arity == 1:
            return self.reverse(parent) if self.invertible else parent
        raise NotImplementedError

    def decompose(self, *args, **kwargs):
        return self.generate(*args, **kwargs)
```

**`NotLayer` — `S = not(S)`**

Tier `S`. Bivector pole swap on `[..., :2]` of the muxed event; `[..., 2:]` (nWhere / nWhen) passes through. Self-inverse.

```
class NotLayer(GrammarLayer):
    rule_name        = "not"
    arity            = 1
    invertible       = True
    tier             = 'S'
    reads_activation = False

    def forward(self, x):
        bivector = x[..., :2].flip(dims=(-1,))
        rest     = x[..., 2:]
        return torch.cat([bivector, rest], dim=-1) if rest.shape[-1] else bivector

    def reverse(self, y):
        return self.forward(y)
```

**`NonLayer` — `S = non(S)`**

Tier `S`. Per-pole bivector complement on `[..., :2]`: `[1 - pos, 1 - neg]`. Self-inverse on each pole; `[..., 2:]` tail passes through.

```
class NonLayer(GrammarLayer):
    rule_name  = "non"
    arity      = 1
    invertible = True

    def forward(self, x):
        bivector = 1.0 - x[..., :2]
        rest     = x[..., 2:]
        return torch.cat([bivector, rest], dim=-1) if rest.shape[-1] else bivector

    def reverse(self, y):
        return self.forward(y)
```

**`IntersectionLayer` — L-tier lattice min on bivector activation**

Tier `L`. Lattice min on a bivector activation `[..., 2]`. `reads_activation = True` so the dispatcher feeds `subspace.materialize(mode='activation')` rather than the muxed event. `monotonic` toggles between RadMin (zero passthrough) and strict min.

```
class IntersectionLayer(GrammarLayer):
    rule_name        = "intersection"
    arity            = 2
    invertible       = False
```

```
    lossy            = True
    tier             = 'L'
    reads_activation = True

    def __init__(self, monotonic=False):
        super().__init__(0, 0)
        self.monotonic = bool(monotonic)

    def forward(self, left, right):
        return Ops.intersection(left, right, monotonic=self.monotonic)

    def reverse(self, parent):
        return parent, parent
```

#### `UnionLayer` — L-tier lattice max on bivector activation

Tier `L`. Counterpart to `IntersectionLayer`: lattice max on a bivector activation, RadMax / strict-max toggle.

```
class UnionLayer(GrammarLayer):
    rule_name        = "union"
    arity            = 2
    invertible       = False
    lossy            = True
    tier             = 'L'
    reads_activation = True

    def __init__(self, monotonic=False):
        super().__init__(0, 0)
        self.monotonic = bool(monotonic)

    def forward(self, left, right):
        return Ops.union(left, right, monotonic=self.monotonic)

    def reverse(self, parent):
        return parent, parent
```

#### `ConjunctionLayer` / `DisjunctionLayer` — S-tier monotonic on scalar activation

Tier `S`. Operate on the **post-codebook scalar activation**: `[B, V]` non-negative strength per prototype, returned by `effective_activation()` (bivector reduced via `max` and gated by modal presence). Strictly monotonic by construction – there is no negative pole on the scalar, so RadMin / RadMax would be wrong.

```
class ConjunctionLayer(GrammarLayer):
    rule_name        = "conjunction"
    arity            = 2
    invertible       = False
    lossy            = True
    tier             = 'S'
    reads_activation = True

    def forward(self, left, right):
        return Ops.intersection(left, right, monotonic=True)

    def reverse(self, parent):
        return parent, parent
```

```python
class DisjunctionLayer(GrammarLayer):
    rule_name        = "disjunction"
    arity            = 2
    invertible       = False
    lossy            = True
    tier             = 'S'
    reads_activation = True

    def forward(self, left, right):
        return Ops.union(left, right, monotonic=True)

    def reverse(self, parent):
        return parent, parent
```

#### `LiftLayer` / `LowerLayer` — S-tier gated SVD slice on muxed event

Tier `S`. `LiftLayer` runs `sigma.compose` (an OR fold) gated by a per-rule `raw_gate` parameter; `LowerLayer` runs `pi.compose` (an AND fold). Both share one matrix per type (`space.sigma_S` / `space.pi_S`) lazy-constructed when the grammar references their rule, with one gate per rule on the SVD diagonal. The reverse uses the analytical LDU inverse of the gated matrix (`sigma.generate` / `pi.generate`), which is exact when the gate is stable.

Both read the muxed event (`reads_activation = False`) since the gated SVD slice operates on the full `[B, V, D]` symbolic event tensor, not just the activation poles.

#### `EqualsLayer` / `PartLayer` — mereology

`equals` weights the right operand by mutual parthood; `part` weights by directional parthood. Both broadcast a scalar score back to the right operand and are lossy.

```python
class EqualsLayer(GrammarLayer):
    rule_name  = "equals"
    arity      = 2
    invertible = False
    lossy      = True

    def forward(self, left, right):
        score = Ops._equal_kernel(left, right, scalar=True)
        while score.ndim < right.ndim:
            score = score.unsqueeze(-1)
        return score * right

    def reverse(self, parent):
        return parent, parent

    def compose(self, left, right):
        return self.forward(left, right)

    def generate(self, parent):
        return self.reverse(parent)


class PartLayer(GrammarLayer):
    rule_name  = "part"
```

```
    arity      = 2
    invertible = False
    lossy      = True

    def forward(self, left, right):
        score = Ops._part_kernel(left, right, scalar=True)
        while score.ndim < right.ndim:
            score = score.unsqueeze(-1)
        return score * right

    def reverse(self, parent):
        return parent, parent

    def compose(self, left, right):
        return self.forward(left, right)

    def generate(self, parent):
        return self.reverse(parent)
```

**`TrueLayer` / `FalseLayer` — pole projection**

```
class TrueLayer(GrammarLayer):
    rule_name  = "true"
    arity      = 1
    invertible = False
    lossy      = True

    def forward(self, x):
        return torch.relu(torch.clamp(x, -1.0, 1.0))

    def reverse(self, y):
        return y


class FalseLayer(GrammarLayer):
    rule_name  = "false"
    arity      = 1
    invertible = False
    lossy      = True

    def forward(self, x):
        return torch.relu(-torch.clamp(x, -1.0, 1.0))

    def reverse(self, y):
        return y
```

**`SwapLayer` — `S = swap(S, S)`**

Sinkhorn-normalized soft permutation. The `swap_logits` and `swap_marker` parameters live on this layer.

```
class SwapLayer(GrammarLayer):
    rule_name  = "swap"
    arity      = 2
    invertible = False
    lossy      = True
```

```
    def __init__(self, swap_size=1, sinkhorn_iters=5):
        super().__init__(0, 0)
        self.swap_size = max(int(swap_size), 1)
        self._sinkhorn_iters = int(sinkhorn_iters)
        self.swap_marker = nn.Parameter(torch.randn(self.swap_size) * 0.01)
        self.swap_logits = nn.Parameter(torch.zeros(3, 3))

    def _soft_perm(self):
        M = self.swap_logits
        for _ in range(self._sinkhorn_iters):
            M = M - M.logsumexp(dim=-1, keepdim=True)
            M = M - M.logsumexp(dim=-2, keepdim=True)
        return M.exp()

    def forward(self, left, right):
        P = self._soft_perm()
        marker = self.swap_marker.to(left.device)
        if left.ndim == 3:
            D = left.shape[-1]
            m = marker[:D].unsqueeze(0).unsqueeze(0).expand_as(left)
        elif left.ndim == 2:
            m = marker[:left.shape[-1]].unsqueeze(0).expand_as(left)
        else:
            m = marker
        if right is None:
            right = left
        stack = torch.stack([left, right, m], dim=0)
        out = torch.einsum('ij,j...->i...', P, stack)
        return out[0]

    def reverse(self, parent):
        return parent, parent

    def compose(self, left, right):
        return self.forward(left, right)

    def generate(self, parent):
        return self.reverse(parent)
```

**QueryLayer — S = query(S, S)**

Returns the left operand unchanged (the chart's query semantic pushes a marker word elsewhere; the rule's tensor contribution is the accumulator passthrough).

```
class QueryLayer(GrammarLayer):
    rule_name  = "query"
    arity      = 2
    invertible = False

    def forward(self, left, right):
        return left

    def reverse(self, parent):
        return parent, parent
```

```
    def compose(self, left, right):
        return self.forward(left, right)

    def generate(self, parent):
        return self.reverse(parent)
```

### Retired: `WhatLayer` / `WhereLayer` / `WhenLayer` / `AbsorbLayer`

Removed 2026-05-04. Slot partitioning is now the dispatcher's responsibility (the modality tensors carry the slot structure intrinsically), so `what` / `where` / `when` are not grammar ops. `absorb` became a base-class marker: `GrammarLayer.absorb(left, right) -> left` lets any binary subclass flag its right operand as syntactic sugar without a dedicated dispatch entry.

### `PiLayer` — parametrized AND fold

`PiLayer` (and `SigmaLayer` below) inherit from `ButterflyLayer`, an intermediate base that adds the optional butterfly access pattern on top of `GrammarLayer`. The unary `forward(x)` is the log-domain multiplicative AND fold over the feature axis; the binary `compose(left, right)` runs the same fold but sums log-mult contri- butions across operands instead of within one.

```
class PiLayer(ButterflyLayer):
    rule_name  = "intersection"
    arity      = 2

    def forward(self, x, binary=False):
        if self.butterfly:
            B, N, D = x.shape
            packed = self._butterfly_pack(x)
            packed_out = self._pi_inner_forward(packed, binary=binary)
            x_out = self._butterfly_unpack(packed_out, B, N, D)
            return self._butterfly_merge(x_out)
        return self._pi_inner_forward(x, binary=binary)

    def reverse(self, y):
        if self.butterfly:
            x_out = self._butterfly_unmerge(y)
            B, N, D = x_out.shape
            packed = self._butterfly_pack(x_out)
            packed_in = self._pi_inner_reverse(packed)
            return self._butterfly_unpack(packed_in, B, N, D)
        return self._pi_inner_reverse(y)

    def compose(self, left, right):
        if self.butterfly:
            raise NotImplementedError("PiLayer.compose not supported in butterfly mode")
        if self.layer.ergodic:
            self.resample_noise()
        W = self.layer.compute_W_current()
        left = left.to(W.device); right = right.to(W.device)
        if self.nonlinear:
            l_l = torch.log(self._to_mult(left))
            l_r = torch.log(self._to_mult(right))
        else:
            l_l = torch.log(left); l_r = torch.log(right)
```

```
        wl = (l_l + l_r) @ W + self.layer._effective_bias()
        return torch.tanh(wl / 2) if self.nonlinear else torch.exp(wl)

    def generate(self, parent):
        if self.butterfly:
            raise NotImplementedError("PiLayer.generate not supported in butterfly mode")
        W_inv = self.layer.compute_Winverse_current()
        parent = parent.to(W_inv.device)
        log_mult_y = torch.log(self._to_mult(parent)) if self.nonlinear else torch.log(parent)
        s = (log_mult_y - self.layer._effective_bias()) @ W_inv
        half = s * 0.5
        op = self._from_mult(torch.exp(half)) if self.nonlinear else torch.exp(half)
        if self.layer.ergodic:
            self.resample_noise()
        return op, op
```

`_pi_inner_forward` / `_pi_inner_reverse` are the shared kernels: `(1+x)/(1-x)` to mult-domain → log → linear → tanh-half → back to $[-1, 1]$. `binary=True` pre-applies `Ops.top2_select_ste` to hard-select the top-2 input operands.

**`SigmaLayer` — parametrized OR fold**

```
class SigmaLayer(ButterflyLayer):
    rule_name  = "union"
    arity      = 2

    def forward(self, x, binary=False):
        if self.butterfly:
            B, N, D = x.shape
            packed = self._butterfly_pack(x)
            packed_out = self._sigma_inner_forward(packed, binary=binary)
            x_out = self._butterfly_unpack(packed_out, B, N, D)
            return self._butterfly_merge(x_out)
        if binary:
            x = Ops.top2_select_ste(x)
        if self.nonlinear:
            x = torch.atanh(x.clamp(-1 + epsilon, 1 - epsilon))
        y = self.layer.forward(x)
        if self.nonlinear:
            y = torch.tanh(y)
        self.activation = y.detach()
        return y

    def reverse(self, y):
        if self.butterfly:
            x_out = self._butterfly_unmerge(y)
            B, N, D = x_out.shape
            packed = self._butterfly_pack(x_out)
            packed_in = self._sigma_inner_reverse(packed)
            return self._butterfly_unpack(packed_in, B, N, D)
        if self.nonlinear:
            y = torch.atanh(y.clamp(-1 + epsilon, 1 - epsilon))
        x = self.layer.reverse(y)
        if self.nonlinear:
            x = torch.tanh(x)
```

```
        self.activation = x.detach()
        return x

    def compose(self, left, right):
        if self.butterfly:
            raise NotImplementedError("SigmaLayer.compose not supported in butterfly mode")
        if self.nonlinear:
            a_l = torch.atanh(left.clamp(-1 + epsilon, 1 - epsilon))
            a_r = torch.atanh(right.clamp(-1 + epsilon, 1 - epsilon))
        else:
            a_l, a_r = left, right
        out = self.layer.forward(a_l + a_r)
        if self.nonlinear:
            out = torch.tanh(out)
        self.activation = out.detach()
        return out

    def generate(self, parent):
        if self.butterfly:
            raise NotImplementedError("SigmaLayer.generate not supported in butterfly mode")
        a_y = (torch.atanh(parent.clamp(-1 + epsilon, 1 - epsilon))
               if self.nonlinear else parent)
        a_sum = self.layer.reverse(a_y)
        half = a_sum * 0.5
        op = torch.tanh(half) if self.nonlinear else half
        return op, op
```

The `OR`-fold is the additive (logit-domain) sum: atanh the operands, sum across operands, then apply the same linear + tanh. `compose` / `generate` close the loop with a balanced inverse so the chart can push parent gradients back into child cells.

### Class registry

```
GRAMMAR_LAYER_CLASSES = {
    # S-tier (post-codebook scalar / muxed event)
    'not':          NotLayer,
    'non':          NonLayer,
    'conjunction':  ConjunctionLayer,
    'disjunction':  DisjunctionLayer,
    'lift':         LiftLayer,
    'lower':        LowerLayer,
    'equals':       EqualsLayer,
    'part':         PartLayer,
    'true':         TrueLayer,
    'false':        FalseLayer,
    'swap':         SwapLayer,
    'query':        QueryLayer,
    # L-tier (logical -- lattice on bivector activation)
    'intersection': IntersectionLayer,
    'union':        UnionLayer,
}
```

### Inverse contract by operator

| Op | Tier | Reverse |
|---|---|---|
| `not` | S | exact (self-inverse pole flip) |
| `non` | S | exact (self-inverse: `non(non(x)) = x`) |
| `lift` | S | exact LDU inverse via gated `sigma.generate` |
| `lower` | S | exact LDU inverse via gated `pi.generate` |
| `conjunction` | S | pseudo-inverse `(parent, parent)` |
| `disjunction` | S | pseudo-inverse `(parent, parent)` |
| `intersection` | L | pseudo-inverse `(parent, parent)` |
| `union` | L | pseudo-inverse `(parent, parent)` |
| `equals` | S | no defined inverse (mereological write) |
| `part` | S | no defined inverse (mereological write) |
| `query` | S | no defined inverse (mereological read) |
| `swap` | S | pseudo-inverse `(parent, parent)` |
| `true` | S | identity passthrough (lossy) |
| `false` | S | identity passthrough (lossy) |

Reverse / inverse Ops (`Ops.negationReverse`, `Ops.conjunctionReverse`, `Ops.disjunctionReverse`, `Ops.liftReverse`, `Ops.liftReverseAll`, `Ops.lowerReverse`, `Ops.lowerReverseAll`) are not in the dispatch registry — they are invoked directly by the Step-6 grammar reverse convention or by `Basis.*` methods. The `_binary_op_inverse_impl(result, W, op, monotonic)` private helper backs both `conjunctionReverse` and `disjunctionReverse` via codebook search.

---

## Rule prediction

`SyntacticLayer.predict_rules(x)` — also exposed on `Chart` after the 2026-05-01 refactor — predicts a rule distribution at each derivation depth. The architecture is weight-tied (recursive) across depths with a learned depth embedding.

Input projection:

$$h^{(0)} = \sigma(W_{\text{in}}\, x + b_{\text{in}}), \qquad h^{(0)} \in \mathbb{R}^{B \times d_{\text{hidden}}}$$

For each depth $d = 0, 1, \ldots, D_{\max} - 1$ (shared weights $W_{\text{deriv}}$, $W_{\text{head}}$):

$$\begin{aligned}
h^{(d+1)} &= \sigma\big(W_{\text{deriv}}\,(h^{(d)} + e_d) + b_{\text{deriv}}\big) \\
z^{(d)} &= W_{\text{head}}\, h^{(d+1)} + b_{\text{head}} \in \mathbb{R}^{B \times K} \\
z_t^{(d)} &\leftarrow \text{stop_grad}(z_t^{(d)}) + (1 - \iota) \cdot \tau_{\text{trans}} \quad \text{(transition-bias on rule } t\text{)} \\
p^{(d)} &= \begin{cases} \text{gumbel_softmax}(z^{(d)}; \tau) & \text{training} \\ \text{softmax}(z^{(d)}) & \text{eval} \end{cases}
\end{aligned}$$

where $e_d$ is the depth embedding, $\iota$ = grammar.interpretation, $t$ is the index of the transition rule (if present), and $\tau$ is the Gumbel-softmax temperature (annealed from 1.0 toward 0.1 via `set_tau`).

Output: `{rule_logits, rule_probs, predicted_rules}`. Training uses gumbel-softmax (soft one-hot, gradients flow through all candidate rules); eval uses softmax + argmax (discrete selection).

---

## Composition

Chart composition lives on the `Chart` class (`bin/Language.py`). `Chart.compose(data, word_space, subspace=None)` runs the inside pass over `data` and populates per-tier rule selections on `word_space.current_rules`. `Chart.generate(target, word_space, subspace=None)` runs the outside pass + Viterbi backtrace and populates `word_space.generate_rules`.

The chart router is selected via `<WordSpace><routerKind>` (XML); either `'chart'` (the default soft-superposition CKY) or `'signal'` (`SignalRouter`, a tensorial alternative). Both surfaces produce the same per-tier rule lists.

### Phase 1: deterministic `not` (legacy 2D path)

Before soft superposition, the legacy 2D-activation path applies a single deterministic `not` to the top-of-stack when that vector has a negative mean, and pushes a `not` word for that batch row. This strips any leading negation from the accumulator before it mixes with siblings. The 3D chart path skips this step — sign flips go through `NotLayer.forward` like every other rule.

### Phase 2: chart inside pass

For each cell $(i, j)$ in the chart and each grammar rule with arity matching the cell's shape, the chart computes:

$$\text{score}_{ij}^{\rho} = \text{compat}(\text{lhs}, \text{rhs}_{ij}) \cdot \text{pair_score}(i, j) \cdot \text{rule_logit}(\rho)$$

then takes the soft mixture (training) or argmax (eval) over all firing rules at that cell. Per-cell rule applications dispatch through `wordSpace.host_layer(tier, rule_name)`, which returns the GrammarLayer instance owned by the host space. The chart then calls `layer.compose(left, right)` (binary) or `layer.forward(x)` (unary).

The compatibility mask `compat[B, P, R]` zeroes out `(pair, rule)` combinations whose typed LHS / RHS categories don't match the pair's slot categories. Function-call rules (legacy, `rhs_symbols = None`) are always compatible. Category `0` (`'?'`) is a wildcard, so unseeded leaves can match any typed rule during warmup.

**Query / norm-drop detection** Detect symbolic contradiction at the accumulation point by comparing the pre- and post-composition Frobenius norms per batch row. Define $\nu_\ell = \|\ell\|_F$ and $\nu_c = \|c\|_F$. When $\nu_\ell > 10^{-6}$ and $\nu_c < \kappa\,\nu_\ell$ with $\kappa = 0.1$ (`_QUERY_NORM_DROP_RATIO`), the new accumulator preserves the prior state and a `query` word is pushed onto WordSubSpace. Otherwise the chart accepts the candidate.

The argmax rule is recorded as a word at the leaf position; `last_rule_per_batch[b]` is updated only when the query mask did *not* fire (so `query` preserves the prior rule context).

### Tensor word buffer

Each `SubSpace` carries two registered buffers alongside the legacy `self.word: list[tuple]`:

```
self.word_records  # [B*K, max_depth, ENTRY_WIDTH] long, ENTRY_WIDTH=7
self.word_count    # [B*K] long, current depth per cell
```

`add_word(...)` is overloaded. The **scalar form** (`add_word(int, int, int, ...)`) appends one validated 7-tuple to `self.word`; used by direct callers (tests, the legacy compose path). The **vector form** (`add_word(LongTensor, ...)`) scatters into `word_records` at each cell's current `word_count`, then increments `word_count` (no host sync).

The chart materializes its tensor buffer before `_compose_chart_cky` returns, by calling `subspace.flush_word_buffer()` inside the chart path. That keeps direct compose callers, the SVO walker, and derivation-trace tests seeing

`self.word` immediately populated in cell-major order. This is "Path B" hybrid tensor buffering with host materialization — not yet the full tensor-only Path A.

### Chart-derived SVO

`Chart._extract_svo_from_trace` walks each batch row's trace after chart compose finishes:

1. Find the last `S -> S VO` entry (outermost reduction).
2. Find the matching `VO -> V O` entry whose `merged_slot` equals the outer rule's right arg.
3. Pull `subject` from the outer's left slot, `verb` from the inner's left, `object` from the inner's right.

Each is a `[1, D]` slice of the original pre-compose `data`. Rows that don't contain the canonical pair of firings get a zero `[1, D]` placeholder; if every row in a batch fails, `last_svo` stays `None`.

---

## POS side-channel

A part-of-speech distribution rides alongside the signal at every chart cell. The trailing axis of the side-channel is `|POS| = |Grammar.categories|` — the union of every `lhs` / `rhs_symbol` declared by the grammar (e.g. `S`, `VO`, `NP`, `VP`, `V`, `N`, `ADJ`, `DET`). Index 0 is `'?'`, the wildcard.

### Storage tiers

- **Per-atom POS tag** (durable, seed source) lives on the SymbolicSpace codebook as `Codebook.category_ids: Long[V]` (one long index per atom row, default `0` = `'?'`). Allocated when the codebook is built with `category=True`. Mirrors the existing `polarity_ids` pattern. Use `set_category(idx, cat_id)` / `get_category(idx)` to read / write per atom.
- **Per-call chart POS** is `Chart._chart_pos: [B, N+1, N+1, |POS|]`, the softmax of `Chart._chart_score` along the trailing axis. Each cell row is a probability simplex. Cleared at the start of every inside pass.
- **Durable per-merge POS** lives on `SubSpace.pos_records: Long[B*K, max_depth]`, parallel to `word_records` / `word_count`. Each entry is the merged-cell POS at the depth recorded by the matching `word_records` row.

### Lexical bootstrap

`Chart._apply_codebook_pos_seed` overrides the learned `_lex_cat_scorer(data)` log-distribution with a one-hot at the codebook-resolved POS for any input position whose nearest atom carries a non-wildcard `category_ids` entry and whose dot-product similarity exceeds `0.1`. Untagged or low-confidence positions retain `softmax(_lex_cat_scorer(data))`. Gradients still flow through `_lex_cat_scorer` on those positions.

### Mechanism 1 — RHS POS compatibility mask

In `_chart_inside`, before scoring each binary merge candidate, the chart gathers the operands' POS distributions (`pos_left[..., rule.rhs_left[r]]`, `pos_right[..., rule.rhs_right[r]]`) and adds `log(p_left * p_right)` to the rule's candidate score. Wildcard rules (`rhs_*[r] == 0`) get an unconditional `1.0` multiplier. For typed rules like `NP = Intersection(ADJ, N)`, this makes the rule fire only when the left operand looks like `ADJ` and the right looks like `N`; the score for `[N, ADJ]` is exponentially suppressed.

### Mechanism 3 — Rule-prediction conditioning

After `_viterbi_extract` runs, `Chart._populate_category_stack` walks the trace and pushes each merge's LHS category embedding (from `WordSpace.category_codebook.W`) onto `WordSpace.category_stack`. Subsequent calls to `WordSpace.predict_rule(b)` then read a parse history of POS embeddings via the

existing `category_stack.flatten(b)` → `Linear(max_depth * pos_dim $\to$ n_rules)` MLP that was already allocated for this purpose.

### Why this trains the network to assign POS to every word

Two reinforcing gradient paths:

1. **Anchored** — codebook atoms with a known category override `_lex_cat_scorer` deterministically. Their POS distributions propagate up the chart and gate Mechanism 1; reconstruction loss pulls the model's downstream rule choices toward those atoms' stored categories.
2. **Emergent** — untagged atoms fall back to `softmax(_lex_cat_scorer(data))`. A wrong POS at a leaf produces a wrong rule firing (RHS mask fails, rule scores drop), bad reconstruction, and a strong gradient back through `_lex_cat_scorer.W` to fix the leaf.

The two combine: tagged atoms anchor a stable POS reference; the scorer learns to extend that reference to untagged atoms whose context makes the correct POS unambiguous. Coverage matters — tagging the closed-class words (DET, AUX, PREP, CONJ) plus the most frequent open-class atoms gives the system a working anchor set.

### `<writeSyntax>true</writeSyntax>` — syntax-tree dump

When `<architecture><writeSyntax>true</writeSyntax></architecture>` is set in the model XML, `BasicModel.forward()` calls `BasicModel.write_syntax_tree(path)` at the end of every forward pass. The output path defaults to `output/syntax.xml` and is overridable via `<architecture><syntaxOutPath>...</syntaxOutPath><`

The dumper walks the chart's `_derivation_trace` and the symbolic codebook to emit one `<forward>` XML fragment per call, with one `<batch>` per row. Category names (`cat="..."`), rule canonicals (`rule="..."`), and POS tags (`pos="..."`) come straight from the live grammar — whatever `Grammar.categories` and `RuleDef.canonical` hold at parse time. For example, given the XOR grammar (data/XOR_grammar.xml) which declares `S = not(S) | conjunction(S, S) | disjunction(S, S)` over a single non-terminal `S`, a parse of the input `1 1` (true ∧ true) under the CKY chart would emit something like:

```
<forward tick="42">
  <batch n="0">
    <node cat="S" rule="S=conjunction(S,S)" i="0" j="2">
      <leaf token="1" pos="S" i="0"/>
      <leaf token="1" pos="S" i="1"/>
    </node>
    <rules>1</rules>
  </batch>
</forward>
```

A grammar that adds typed categories (e.g. `NP = intersection(ADJ, N)`) would produce richer trees with those category names appearing on the `<node>` and `<leaf>` elements; the dumper has no built-in category vocabulary of its own.

Each `<node>` carries `cat="<lhs_category>"`, `rule="<rule_canonical>"`, and `i/j` for the span; each `<leaf>` carries `token`, `pos` (resolved through `SymbolicSpace.subspace.what.category_ids`, falling back to `'?'` when the atom carries no tag), and the input position `i`. The `<rules>` element lists the global rule-id sequence for compact diff-based regression.

The file is truncated on the first call within a process and appended on subsequent calls so a training run accumulates one `<forward>` fragment per forward pass. Wrap the file in a single `<syntaxLog>` root element (or use a streaming XML parser) if a single valid XML document is required. `<writeSyntax>` defaults to `false`; the tree-builder is fully bypassed in that case (no per-forward overhead).

Note: when the parser's `<routerKind>signal</routerKind>` is selected, the SignalRouter bypasses `Chart._chart_inside` and no derivation trace is recorded; the dumper emits `<noTrace/>` for those rows. Set `<routerKind>chart</routerKind>` (the default) to use the CKY inside pass that populates the trace.

---

## Per-space dispatch (`SyntacticLayer`)

Each Space (SymbolicSpace, ConceptualSpace, PerceptualSpace) carries a `SyntacticLayer` instance whose `forward()` and `reverse()` fire one fold step per call, per the chart's per-tier rule selection. Construction lives in `build_space_syntactic_layer`:

- `host_layers`: dict mapping `rule_name` to a `GrammarLayer` instance (parametrized folds like `space.pi` for `'intersection'`, `space.sigma` for `'union'`, plus stateless rules built lazily from `GRAMMAR_LAYER_CLASSES`).
- Each `host_layer` is registered with the WordSpace via `register_host_layer(tier, rule_name, layer)` so the chart can consult `host_layer(tier, rule_name)` during dispatch.

`SyntacticLayer.forward(subspace)` reads `word_space.current_rules[tier]`, advances a per-tier cursor, and dispatches to the chosen layer's `forward()` (only for arity-1 rules — binary rules fire inside the chart's `compose` via `host_layer.compose(left, right)`). `reverse()` mirrors via `word_space.generate_rules` and `layer.reverse()`. The cursor resets at the start of each new `word_space.compose()` / `word_space.generate()` call via the WordSpace generation counters.

When the chart router is `'signal'`, the per-rule unary fold is skipped — the SignalRouter has already executed the derivation tensorially and written the `[B, 1, D]` root state back into the subspace.

#### `WordSpace.forwardSymbols` / `reverseSymbols`

These methods used to invoke a private composition pipeline. After the 2026-05-01 refactor they are thin: `forwardSymbols(data, subspace)` demuxes the muxed symbol tensor into the subspace's modality slots when shapes match (the side-effect downstream slot selectors depend on); the actual symbolic composition runs on the chart via the `ChartCompose` pipeline stage. `reverseSymbols` is a pass-through; chart-driven generation handles the symbol-side reverse via `ChartGenerate` + per-space `SyntacticLayer.reverse` dispatch.

---

## Downward head emission (`S -> C`)

Selected by `<WordSpace><downwardGeneration>true</downwardGeneration>`. After the upward parse reduces `N` leaves to a single root vector, `BasicModel.forward` takes that root (`sym_vectors[:, 0, :]`) and calls `WordSpace.reconstruct(state, inputSpace)`, which runs the downward `S -> C` rule as a one-shot projection onto a codebook.

### Projection-coefficient scoring

$$W_{\text{norm}} = W / \|W\|_{2,\text{row-wise}}, \qquad \text{scores} = \text{clamp}_{\geq 0}(\text{state} \cdot W_{\text{norm}}^{\top})$$

The per-row argmax is `best_idx`. Because `state` is *not* L2-normalized (only the codebook rows are), the score is "how much of atom $k$ lives in the state" rather than a pure angle. The resulting `contained = scores[:, k] * W_{\text{norm}}[k]` is the slice of the atom actually contained in the state; `residual = state - contained` is the leftover meaning a future expansion step (NP / VP templates) could consume.

#### `WordSpace.reconstruct(state, codebook_space)`

Any space whose `.subspace.basis` exposes a `getW() -> [V, D]` works. `InputSpace` (word embedding) is the intended source — projecting the deep state onto word vectors picks the vocabulary entry that best matches the sentence's composed meaning, and `.wv.index_to_key` decodes the head index back to a token. `SymbolicSpace` (internal atom codebook) is also valid once a codebook is populated there. When

the codebook is unwired (passthrough `Codebook` with `getW() == None`), `reconstruct` returns a trivial `heads=[0]*B` so the loss path never crashes.

Returns `{heads, contained, residual, state}`. `BasicModel.forward` stashes `heads` as `self._predicted_head` so a training loss can compare against a supervised head token, and so `bin/interact_head.py` can print the decoded word after each forward pass.

---

## Word encoding

Each word is a 7-tuple stored in `subspace.word` (the legacy list) and mirrored in `subspace.word_records` (the tensor buffer). Per-cell: `(batch, vector, rule, leaf1, leaf2, leaf3, order)` where the `leaf*` slots carry codebook indices for terminal words and child slot indices for internal nodes. Terminal words carry `order = -1` and a codebook index in `leaf1`; the leaf ledger used by chart generation is built from these entries.

The list is a derivation tree in pre-order: the first entry is the root, and binary rules expand left-first.

---

## Decomposition

`Chart.generate(target, word_space, subspace)` reconstructs the pre-compose tensor from the symbolic word record by walking the chart's outside pass + Viterbi backtrace. At each step it dispatches through `host_layer.generate(parent)` for binary rules or `host_layer.reverse(y)` for unary invertible rules. Lossy rules (`equals`, `part`, `true`, `false`, `query`, `swap`, `conjunction`, `disjunction`, `intersection`, `union`) emit their pseudo-inverses (typically the parent passed back unchanged).

The codebook fast path: if a basis is available and shapes match, the generator looks up each terminal word (those with `order = -1`) and places `cb[leaf1]` at the recorded position, producing the exact pre-compose tensor.

---

## How syntax is trained

The syntax code is end-to-end differentiable: `Chart.predict_rules` produces soft `rule_probs`; the chart inside pass uses soft superposition over compatible rules; the binary `compose` / `generate` paths flow gradients through the parametrized fold weights (PiLayer / SigmaLayer / SwapLayer logits). The deterministic `not` phase 1 step in the legacy 2D path is non-differentiable but only fires when the top-of-stack mean is negative.

`BasicModel.runBatch()` computes three losses: output loss, optional reconstruction loss, optional embedding loss. There is no explicit syntax loss, parse-tree loss, or rule-label supervision. Syntax fitness emerges indirectly through the reconstruction loss: rules that consistently support successful reconstruction accumulate weight, while weaker alternatives diminish.

Under `MentalModel.xml`, syntax is enabled, the chart and per-space dispatchers are instantiated, and rules are predicted, fired, and recorded. `BasicModel.forward()` goes through `SigmaLayer` rather than invoking the chart pipeline directly; syntax is active as computation and analysis but not yet placed on a strong loss path in the stock training loop.

---

## Two-Layer logic

The logic system operates at two levels. The **subsymbolic** layer treats objects as vector sets $(B, N, D)$ interpreted as RBF / luminosity fields: union is $\max(\ell, r)$ (strongest affirmation); intersection is $\min(\ell, r)$

(shared commitment); negation is $-x$ (antipodal opposition on the hypersphere); non is $\alpha x$ with $\alpha \in [0, 1)$ (contraction toward zero); parthood is fuzzy max-coverage in $[-1, 1]$ (signed containment).

A **symbolization** map projects vectors to scalar truth strength:

$$s(X) = 2 \cdot \text{mean}(\|x_i\|) - 1 \quad \in [-1, 1]$$

Interpretation: $+1 \to$ strong presence, $0 \to$ neutral, $-1 \to$ absence.

The **symbolic** layer operates on scalars in $[-1, 1]$: neg is $-a$, non is $\alpha a$, union is $\max(a, b)$, intersection is $\min(a, b)$, part is $\text{clamp}(b - a, -1, 1)$.

The key insight: subsymbolic = geometry; symbolic = order + polarity; symbolization = norm projection.

---

## SymbolicSpace integration

### Forward path

1. Extract concept activation $[B, n_{\text{Concepts}}]$.
2. Map through `PiLayer(monotonic=True, invertible=True)` to $[B, n_{\text{Symbols}}]$.
3. Codebook quantization (when enabled) produces one-hot activation plus vectors.
4. The chart runs its inside pass over symbol vectors, dispatching per-cell rule applications through `wordSpace.host_layer('S', rule_name)` (which resolves to the SymbolicSpace's `SyntacticLayer`).

### Reverse path

1. Extract symbol activation $[B, n_{\text{Symbols}}]$.
2. Exact inverse of the PiLayer recovers $[B, n_{\text{Concepts}}]$.
3. The chart's outside pass reconstructs the pre-compose tensor from the word record via `host_layer('S', rule_name).generate(parent)`.

### Key properties

- Symbols are zero-dimensional — pure activation scalars, not vectors.
- The PiLayer allows $n_{\text{Concepts}} \neq n_{\text{Symbols}}$.
- The chart's per-cell soft superposition runs over all active S-tier rules; rule firing is gated by LHS / RHS POS compatibility via `chart_pos` (see POS side-channel).

---

## AssociationLayer (EQUALS implementation)

The `AssociationLayer` is a cross-symbol associative memory used by the EQUALS rule. Two modes are available. **`type="symmetric"`** is Hopfield-like: it learns projection $A$, computes association scores $A^\top A$, and softmax-retrieves the associated pattern; associations are symmetric ($A \equiv B \Leftrightarrow B \equiv A$). **`type="hopfield"`** is modern Hopfield: separate query / key projections and softmax-gated retrieval.

Input and output are both $[B, N]$ activation vectors. The layer is learnable and its parameters are trained end-to-end via the reconstruction loss.

---

## Future: RuleNode on the ReconstructionStack

A planned Phase 2 introduces a `RuleNode` structure with slots for `(rule_id, method_name, forward_op, reverse_op, arity, args)` and enriches `ReconstructionStack` to support variable-fidelity reconstruction

(pos-only; grammar + pos; grammar + pos + args). This pairs generative rules with their compositional inverses so downstream consumers can pull as much or as little reconstruction context as they need from the same word record.

---

## Design history: learned SVO plan

This appendix preserves the original design narrative that justified moving from a positional SVO tap to grammar-derived SVO roles. The plan was authored before the chart-compose implementation landed; Phases A–D have since shipped. Kept here for archaeology and as a reference for any future extension of the same approach.

`TruthLayer.universality(subject, verb, object, lifting_layer, symbolic_space)` implements the Golden Rule: it scores luminosity change under S/O reversal, so $K(X, Y) + K(Y, X)$ illuminates more than $K(X, Y)$ alone for kind actions. The method takes $(S, V, O)$ concept tensors as inputs; the question was where those three tensors come from.

### Historical starting point (pre-chart)

SVO was produced by a positional tap inside `SyntacticLayer._compose_vector`: the first three active leaf positions of a batch row were labelled S/V/O when every row had at least three active leaves. For the canonical three-token transitive corpus in `test/test_universality.py`, this produced the right roles. It was wrong for realistic inputs: determiners shifted the roles, adjectives inserted modifiers, word-order variation broke it entirely, and embedded clauses were invisible.

The plumbing downstream of `last_svo` was already correct for a learned tap to drop into:

- `Chart.last_svo: Optional[Tuple[Tensor, Tensor, Tensor]]`
- `SyntacticLayer.lifting_layer: LiftingLayer` (instantiated in `init_lifting`, called from `WordSpace._build_syntactic_layer`)
- `BasicModel.forward` reads both, calls `truth_layer.universality(...)`, and exposes the result via `self._universality_score`
- `truth_modulated_loss(universality_score=...)` folds the score into the training loss

Only the producer of `last_svo` needed to change.

### Phases summary

The five phases shipped:

- **Phase A — Grammar metadata.** `RuleDef` extended with `(lhs, rhs_symbols)` so each rule declares its typed nonterminal / terminal categories. `Grammar.configure` iterates over every key in the XML `<grammar>` block; the bare-symbol-sequence form (`<S>S VO</S>`) was added as a typed compose alternative to the function-call form.
- **Phase B — Chart restructure.** The strict left-associative cascade was unable to produce `(S, (V, O))` from an `N V N` sentence (it forced step 1 to merge `leaf[0]` with `leaf[1]`). The chart replaces it with per-step pair selection: a pair-scorer MLP picks *which two adjacent leaves* to merge, plus *which rule* to apply.
- **Phase C — Category tensor.** A `category: [B, N]` tensor rides alongside `data: [B, N, D]`. At each merge, a compatibility mask zeros out `(pair, rule)` combinations whose typed RHS doesn't match the pair's slot categories. Category `0` (`'?'`) is a wildcard so unseeded leaves match any typed rule during warmup.
- **Phase D — SVO extraction.** SVO falls out of the derivation trace. After compose, find the outermost `S -> S VO` firing, find the matching `VO -> V O`, pull S from the outer's left arg and V / O from the inner's args. Rows without the canonical pair get zero placeholders.

- **Phase E — Training dynamics.** Soft pair mixture, soft rule mixture, hard category argmax (with Gumbel-softmax to keep gradients flowing through "wrong" choices); curriculum from three-content-word SVO to longer / modified inputs.

Each phase is implemented in `bin/Language.py` (chart side) and `bin/Layers.py` (per-rule layers). The implementation tracker is `doc/plans/2026-04-20-LearnedSVO-integration.md`; the chart-compose section above describes the runtime behavior.

#### Stable hooks already wired

- `Chart.last_svo: Optional[Tuple[Tensor, Tensor, Tensor]]` — set by `_extract_svo_from_trace` at the end of `_compose_chart_cky` / `_compose_chart_cky_viterbi`.
- `SyntacticLayer.lifting_layer: LiftingLayer` — created in `init_lifting`, called from `WordSpace._build_syntactic_layer`.
- `BasicModel.forward` reads both, invokes `truth_layer.universality(s, v, o, lifting_layer, symbolicSpace)`, and stores `self._universality_score`; `truth_modulated_loss` integrates the score into the training loss.
- `BasicModel._predicted_head: Optional[list[int]]` — set by the downward head emission described above.

These four interfaces stayed stable when learned SVO replaced positional SVO. The producer changed from the positional `_compose_vector` tap to the chart trace walker.

# Mereology

Single-page reference for the mereological grammar: parthood as the fundamental operation, the five mereological relations, and the `ImpenetrableLayer` regularizer that enforces them on the symbol codebook.

---

## Parthood as Clipped Cosine Projection

Parthood is the single fundamental operation of the grammar. Every other mereological relation is defined in terms of it.

For two concepts $A, B \in \mathbb{R}^D$:

$$\text{part}(A, B) = \frac{\max(0,\ A \cdot B)}{|A|\,|B|}$$

This clipped cosine projection lies in $[0, 1]$ and satisfies Boole's contrapositive trivially:

$$\text{part}(A, B) = \text{part}(-B, -A)$$

because $(-B) \cdot (-A) = A \cdot B$ and norms are sign-invariant.

**Empty-operand contract.** When $|A|$ or $|B|$ is near zero, `part` returns 1.0 — the empty set is part of everything, matching the legacy semantics.

---

## The Full Suite

Every member of the suite is expressible through `part` (plus the existing bitonic `conjunction` / `disjunction` on `Basis` as helpers):

| Method | Formula | Interpretation |
|---|---|---|
| `part(A, B)` | $\max(0, A \cdot B) / ( | A |
| `whole(A, B)` | $\mathrm{part}(B, A)$ | A contains B. |
| `equal(A, B)` | $\mathrm{part}(A, B) \cdot \mathrm{part}(B, A)$ | Mutual parthood, $[0, 1]$. |
| `overlap(A, B)` | $0 < \mathrm{equal}(A, B) < 1$ | Strictly-partial mutual parthood (boolean indicator). |
| `underlap(A, B)` | $\mathrm{equal}(A, B) = 0$ | No mutual parthood (boolean indicator). |
| `boundary(A, B)` | $ | \mathrm{part}(A, B) - \mathrm{part}(B, A) |

---

## Region Partition

`equal(A, B)` partitions the mutual-parthood scalar into three disjoint regions:

| Region | Condition |
|---|---|
| Underlap | $\mathrm{equal} = 0$ |
| Overlap | $0 < \mathrm{equal} < 1$ |
| Identity | $\mathrm{equal} = 1$ |

Under clipped cosine, `part` is symmetric, so the three regions collapse to a single scalar classifier. For a future `Basis` with asymmetric `part` (e.g. codebook-search based), the general five-relation table applies — see the classification below.

**Five-relation table** (general case, for thresholds $\tau \approx 1$ and $\epsilon \approx 0$):

| Relation | Condition |
|---|---|
| `disjoint(a,b)` | $P[a, b] < \epsilon$ and $P[b, a] < \epsilon$ |
| `part(a,b)` | $P[a, b] > \tau$ and $P[b, a] < \epsilon$ |
| `part(b,a)` | $P[a, b] < \epsilon$ and $P[b, a] > \tau$ |
| `equal(a,b)` | $P[a, b] > \tau$ and $P[b, a] > \tau$ |
| `overlap(a,b)` | Both partial (neither $> \tau$ nor $< \epsilon$) |

**Asymmetric subsumption.** Since `part` is symmetric under clipped cosine, classical asymmetric subsumption ("A is a subset of B, but not vice versa") is *not* encoded in the raw magnitude of `part`. It is recovered **relationally** via figure / ground: compare $\mathrm{part}(A, B)$ against $\mathrm{part}(A, \neg B)$. This is intentional — it is what makes Boole's contrapositive hold exactly.

---

## Mereological Fusion

Fusion of a truth set is the **least upper bound** (LUB / join) over stored truth vectors:

$$\mathrm{fusion} = \max_i \mathrm{truths}[i]$$

(elementwise max). In bivector space, the fusion vector names the top-right corner of the axis-aligned bounding hyperrectangle dominating every stored truth. Fusion is the geometric dual of luminosity: LUB (join) vs GLB (meet).

Degree of Truth is already baked into each stored truth (`stored[i] = activation_i * degree_i` in `TruthLayer.record`), so fusion is trust-weighted automatically.

See Logic.md section **Fusion** for the full discussion and the leading-bivector / paired-index layout caveat.

## ImpenetrableLayer: Five-Relations Regularizer

`ImpenetrableLayer` is a regularizer over the SymbolicSpace symbol codebook. It classifies each ordered pair of codebook rows $(i, j)$ into one of the five mereological relations above (using `Basis.part` on the rows), and penalizes partial overlap when paired with a trust mismatch. This is the learned half of the Codebook Uniqueness Contract: `.what` content must distinguish every prototype so the parthood lattice stays well-formed (the structural `.where`-uniqueness is enforced separately by the codebook offset registry).

### Penalty

$$ \mathcal{L} = \mathrm{overlap_weight} \cdot \frac{\sum_{i \neq j} \mathrm{overlap}(i, j) \cdot |\mathrm{trust}(i) - \mathrm{trust}(j)|} {K(K-1)}

- \mathrm{variance_floor\ term} $$

where the overlap strength is damped to zero as the pair approaches `equal`:

$$\text{overlap_strength}(i, j) = \min(P[i, j],\ P[j, i]) \cdot (1 - \max(P[i, j],\ P[j, i])^k)$$

with $k$ = equal_suppression (default 4.0). This makes the penalty zero for `disjoint`, `equal`, and `part` (strict) relations, and active only for the `overlap` region.

A separate `variance_floor` term guards against row collapse by penalizing when the per-dim standard deviation of codebook rows falls below a floor.

### Trust

Trust is per-row usage frequency, derived from the Codebook's vector-quantization EMA counts:

$$\text{trust}(i) = \frac{\text{cluster_size}[i]}{\sum_j \text{cluster_size}[j]}$$

When VQ is absent (e.g. the basis is a `Tensor` rather than a `Codebook`), trust falls back to $\|cb[i]\| / \max_j \|cb[j]\|$.

### Diagnostics

After `forward()`:

- `last_overlap_loss` — scalar overlap-penalty contribution.
- `last_variance` — scalar variance-floor contribution.
- `last_relation_counts` — dict with counts of each relation (`disjoint`, `part_ij`, `part_ji`, `equal`, `overlap`) summing to $K(K-1)$.

### Configuration

XML knobs (under the SymbolicSpace config section):

| Knob | Default | Meaning |
|---|---|---|
| `overlapWeight` | 0.1 | Weight of the overlap × trust-diff penalty |
| `varianceFloor` | 0.01 | Minimum per-dim std; below this triggers the variance penalty |
| `fullPartThreshold` | 0.9 | $\tau$: part score above this is "full part" |
| `disjointThreshold` | 0.1 | $\epsilon$: part score below this is "disjoint" |
| `equalSuppression` | 4.0 | $k$: damping exponent near the equal corner |

## Why This Design

- **Parthood as projection.** One formula, Boole-contrapositive exact, continuous in $[0, 1]$, and the full suite composes on it. The old composite formula `conjunction(1 - dist(x, x \cap y), 1 - dist(y, x \cup y))` mixed set-valued operands with a distance — the projection form is simpler and operates directly on the bivector SymbolicSubSpace.
- **Overlap penalty (not antisymmetry).** The legacy `ImpenetrableLayer` penalized mutual parthood $P[i, j] \cdot P[j, i]$ directly, which pushes *every* overlapping pair apart, including pairs that should be equal (synonyms, tied codebook slots). The five-relations design leaves `equal` pairs alone and only penalizes the genuinely ambiguous middle region, gated by trust mismatch so that two high-trust near-synonyms with matched usage aren't forced apart either.
- **Trust via VQ EMA.** Codebook usage frequency is already tracked for VQ commit loss; reusing it as trust avoids a second bookkeeping path.
- **Parthood is preserved by Pi / Sigma.** Under `<bivectorOutput>true</bivectorOutput>` on PerceptualSpace, ConceptualSpace, and SymbolicSpace, every activation in the chain lives on the non-negative paired-index cone $[0, 1]^{2K}$ and the Pi / Sigma layers are restricted to entry-wise $W \geq 0$ via `NonNegativeInvertibleLinearLayer` (or `NonNegativeLinearLayer`). Positive matrices are exactly the monotone operators on a positive cone: $a \leq b$ componentwise $\Rightarrow Wa \leq Wb$ componentwise. Componentwise $\leq$ on the cone *is* the parthood partial order, so each lift / lower in the chain preserves parthood pole-by-pole — a whole always contains its parts after Pi / Sigma. The bivector layout keeps the contradiction corner $[1, 1]$ distinct from the ignorance corner $[0, 0]$ under the positive matmul, which a single bitonic axis would collapse. See Spaces.md "Monotonicity of the lift / lower chain".

# Logic

## Overview

This document defines the logic system at two levels:

1. **Subsymbolic (vector / field level)** – geometry in ConceptualSpace
2. **Symbolic (scalar level in [-1,1])** – order + polarity in SymbolicSpace
3. **Rationality** – propositional truth store built on top of both

The executable implementations of the subsymbolic and symbolic operators are the `Method` subclasses documented in Language.md Section Methods.

---

## 1. Subsymbolic Layer

Objects:

- Vector sets: (B, N, D)
- Interpreted as RBF / luminosity fields

### Operators

- **Union**: Combine sets

  `union(A, B) = concat(A, B)`

- **Intersection**: Co-supported regions (RBF product / merge)
- **Negation (affirming)**:

  `neg(x) = -x`

  Antipodal opposition on hypersphere

- **Non (non-affirming negation)**: `Basis.non()` – bitonic: returns zero (complete withdrawal); monotonic: `relu(x - threshold)` with a learnable threshold parameter.
- **Parthood** (fundamental): `Basis.part()` – clipped cosine projection in $[0, 1]$:

$$\text{part}(x, y) = \frac{\max(0,\ x \cdot y)}{\|x\| \cdot \|y\|}$$

  Satisfies Boole's contrapositive $\text{part}(x, y) = \text{part}(-y, -x)$ trivially (dot product and norms are both sign-invariant under joint negation). The full mereological suite (`whole`, `equal`, `overlap`, `underlap`, `boundary`) composes through `part`.

---

## 2. Symbolization

Map vectors → scalar truth strength

For $X \in (B, N, D)$:

$$s(X) = 2 \cdot \text{mean}(\|x_i\|) - 1$$

Range: [-1, 1]

Interpretation:

- $+1$ → strong presence
- $0$ → neutral
- -1 → absence

---

## 3. Symbolic Layer (Scalars in [-1,1])

`Basis` supports two modes: **monotonic** (plain min/max, used by PerceptualSpace, ConceptualSpace, and SymbolicSpace whenever `<bivectorOutput>true</bivectorOutput>` is set, so the entire P→C→S chain operates on the non-negative paired-index cone) and **bitonic** (sign-aware, the legacy default). The monotonic forms are listed here; the bitonic forms (RadMin, RadMax) are in Section 7 Radial Operators. See Spaces.md "Monotonicity of the lift / lower chain" for why the chain stays order-preserving end-to-end under the bivector $+ W \geq 0$ regime.

Let $a, b \in [-1, 1]$.

### Negation (affirming)

$$\text{neg}(a) = -a$$

### Non (non-affirming)

Bitonic: $\text{non}(a) = 0$. Monotonic (learnable threshold $\tau$): $\text{non}(a) = \text{relu}(a - \tau)$.

### Union (monotonic)

$$a \cup b = \max(a, b)$$

### Intersection (monotonic)

$$a \cap b = \min(a, b)$$

**Parthood as Projection**

Parthood is the **fundamental mereological operation**. For two concepts $A, B \in \mathbb{R}^D$:

$$\text{part}(A, B) = \frac{\max(0,\ A \cdot B)}{\|A\| \cdot \|B\|}$$

This clipped cosine projection is in $[0, 1]$. It satisfies Boole's contrapositive $\text{part}(A, B) = \text{part}(-B, -A)$ trivially because $(-B) \cdot (-A) = A \cdot B$ and norms are sign-invariant.

The full mereological suite composes through `part`:

| Method | Formula |
|---|---|
| `whole(A, B)` | `part(B, A)` |
| `equal(A, B)` | $\text{part}(A, B) \cdot \text{part}(B, A)$ |
| `overlap(A, B)` | $0 < \text{equal}(A, B) < 1$ (region indicator) |
| `underlap(A, B)` | $\text{equal}(A, B) = 0$ (region indicator) |
| `boundary(A, B)` | $ |

$\text{equal}(A, B) \in [0, 1]$ partitions into three disjoint regions:

- $\text{equal} = 0 \rightarrow$ **underlap** (disjoint)
- $0 < \text{equal} < 1 \rightarrow$ **overlap** (strictly partial)
- $\text{equal} = 1 \rightarrow$ **identity** (perfect mutual parthood)

Under clipped cosine $\text{part}(A, B) = \text{part}(B, A)$ (cosine is symmetric), so `equal` reduces to $\text{part}^2$. Asymmetric classical subsumption is recovered **relationally** via figure/ground: compare $\text{part}(A, B)$ against $\text{part}(A, \neg B)$. This is what makes Boole's contrapositive hold exactly.

See Mereology.md for the full five-relations reference and the `ImpenetrableLayer` regularizer that enforces these relations on the symbol codebook.

For ternary scalar values, the monotonic projection reduces to:

| part(a,b) | **+1** | **0** | **-1** |
|---|---|---|---|
| **+1** | 1 | 1 | 0 |
| **0** | 1 | 1 | 1 |
| **-1** | 0 | 1 | 1 |

Zero is vacuously part of everything (empty-set convention); same-sign values fully contain each other; opposite signs have zero parthood.

---

## 4. Key Insight

- Subsymbolic layer = geometry
- Symbolic layer = order + polarity
- Symbolization = norm projection

This cleanly separates:

- representation (vectors)
- logic (scalars)

---

## 5. Rationality

Rationality is the propositional logic layer. It is built on the S-tier grammar rules `part(S, S)` and `equals(S, S)`, which are the two relations that compose propositions about the world.

### Truth Statements

A **truth statement** is any assertion that the model has meaningfully processed through the full pipeline (InputSpace → … → SymbolicSpace), producing a symbolic activation vector. The `TruthLayer` – owned by `WordSpace` and reachable as `self.wordSpace.truth_layer` – stores these activations **scaled by** the DegreeOfTruth:

$$\text{stored} = \text{activation} \times \text{degree}$$

The DegreeOfTruth in $[-1, 1]$ is baked into the stored vector:

| Degree | Stored Vector | Effect |
|---|---|---|
| +1 | full activation | attractor |
| 0 < d < +1 | scaled activation | weak attractor |
| 0 | zero vector (inert) | prunable |
| -1 < d < 0 | scaled, negated | weak disperser |
| -1 | negated activation | disperser |

### TruthLayer (WordSpace.truth_layer)

`TruthLayer` in `Layers.py` is instantiated by `WordSpace.__init__` alongside the (now unified) `SyntacticLayer`. `SymbolicSpace.forward` reads it via `self.wordSpace.truth_layer` and records activations when `<accumulateTruth>` is set to a value > 0 (the degree of truth, 0..1):

- **record(activation, degree)** – store `activation * degree`.
- **query(activation)** – find the closest stored truth by cosine similarity.
- **field(concepts)** – project all stored truths into ConceptualSpace as a scalar field over concept vectors.

### Truth Field

When stored truths are projected into ConceptualSpace via `field()`, they form a scalar field $f : \mathbb{R}^D \to [-1, 1]$ over concept vectors:

$$f(c) = \frac{1}{n} \sum_{i=1}^{n} c \cdot t_i$$

where $t_i = \text{activation}_i \times \text{degree}_i$ is a stored truth (DoT already baked in) and $c$ is a unit-normalised concept vector.

This field has two kinds of regions:

- **Attractors** ($f \to +1$): concept vectors near truths with high positive degree. These represent regions of conceptual space where stored knowledge says "this is true."
- **Dispersers** ($f \to -1$): concept vectors near truths with high negative degree. These represent regions where stored knowledge says "this is false."

### Consonance and Dissonance

Stored truths should satisfy two consistency conditions:

1. **Internal consonance** – truths should be mutually consistent. If $\text{part}(A, B)$ has degree +1, and $\text{part}(B, C)$ has degree +1, then $\text{part}(A, C)$ should not have degree -1.
2. **External consonance** – incoming statements processed through the pipeline should be evaluated against the truth field. High similarity to a +1 truth is consonance; high similarity to a -1 truth is dissonance.

### Propositional Structure

Both parthood and equality are S-tier operations on the bivector SymbolicSubSpace after the 2026-04-19 C/P/S merge:

- **part(S, S)** – containment on the bivector symbol subspace. "A is part of B." `Basis.part(A, B)` is a clipped cosine projection in $[0, 1]$. The Grammar applies this as `score * B`, scaling the whole by the parthood degree.
- **equals(S, S)** – identity as mutual parthood. `equalsForward` delegates to `Basis.equal` (mutual parthood) on the bivector SymbolicSubSpace. Returns 1 only when the two symbols are parts of each other.

Together, `equals` and `part` define a partial order over symbolic activations. The truth store captures this order as a database of grounded propositions.

### Truth Accumulation Pipeline

Truth entries arrive as `(text, DoT)` pairs from the client's TruthSet. `store_truths()` stages all texts on TheData and runs a standard inference epoch via `runEpoch(split="runtime")`. During forward processing, `SymbolicSpace.forward()` records each raw symbolic activation into the TruthLayer (degree 1.0). After the epoch completes, each stored activation is scaled by its DegreeOfTruth:

$$t_i = \text{activation}_i \times \text{DoT}_i$$

This two-phase design (encode all, then scale) ensures all truths are encoded in the same pipeline context before DoT modulation is applied.

---

## 6. Consistency and Verification

### Internal Consistency (within a TruthSet)

A TruthSet should be internally consonant: its stored truths should not contradict each other. The truth field provides a natural test.

Given $n$ stored truths $\{t_i\}$, project them into ConceptualSpace via `field()`. For each truth $t_j$, evaluate the field produced by all *other* truths $\{t_i : i \neq j\}$ at the concept vector underlying $t_j$. If the field value is strongly negative, truth $j$ is dissonant with the rest of the set:

$$d_j = f_{\setminus j}(c_j) = \frac{1}{n-1} \sum_{i \neq j} c_j \cdot t_i$$

where $c_j$ is the unit-normalised concept vector for truth $j$.

- $d_j > 0$: truth $j$ is consonant with the set (supported by neighbours)
- $d_j \approx 0$: truth $j$ is independent (orthogonal – neither supported nor contradicted)

- $d_j < 0$: truth $j$ is dissonant (contradicted by neighbours)

This is a leave-one-out consistency check. It does not require symbolic logic – it falls out of the geometry of the stored vectors and the DoT already baked into them. A disperser (DoT $< 0$) near an attractor (DoT $> 0$) will naturally produce negative field values, flagging the contradiction.

### Verification (incoming statement against stored truth)

`verify(statement)` processes an incoming statement through the pipeline to obtain its symbolic activation $a$, then queries the truth field:

$$v = f(c_a) = \frac{1}{n} \sum_{i=1}^{n} c_a \cdot t_i$$

where $c_a$ is the concept vector underlying $a$.

| $v$ | Interpretation |
|---|---|
| $v \to +1$ | strong support – statement aligns with stored attractors |
| $v \approx 0$ | no opinion – statement is orthogonal to stored knowledge |
| $v \to -1$ | strong contradiction – statement aligns with dispersers |

This gives a scalar degree of verification in $[-1, 1]$ without requiring the incoming statement to exactly match any stored truth. Cosine similarity via `wordSpace.truth_layer.query()` can also find the single closest truth and return its degree, which is useful for point lookups.

### Logical Entailment (augmenting geometry with symbolic closure)

The subsymbolic field checks above detect *geometric* consistency – truths that point in contradictory directions in the embedding space. But they do not enforce *logical* entailments. Consider:

- Stored: $\text{part}(A, B)$ with DoT $+1$
- Stored: $\text{part}(B, C)$ with DoT $+1$
- The transitive closure $\text{part}(A, C)$ is *entailed* but not stored.

If a statement $\neg\text{part}(A, C)$ arrives, the field check may not flag it – $A$ and $C$ could be geometrically distant even though the chain of parthood connects them. Geometry captures similarity, not inference chains.

A symbolic closure layer would address this by operating on the propositional structure of stored truths:

1. **Extract propositions.** Each stored truth whose activation was produced by an S-tier grammar rule (`equals(S,S)` or `part(S,S)`) carries relational structure: a relation and two operands. These can be read off the symbolic activation by decomposing it into the relation slot and the two argument slots.

2. **Compute closure.** Apply transitivity rules on the extracted propositions:

   - $\text{part}(A, B) \wedge \text{part}(B, C) \Rightarrow \text{part}(A, C)$
   - $\text{equals}(A, B) \wedge \text{equals}(B, C) \Rightarrow \text{equals}(A, C)$
   - $\text{equals}(A, B) \wedge \text{part}(A, C) \Rightarrow \text{part}(B, C)$

   Each entailed proposition inherits a DoT from its premises – the minimum (intersection) of the premise DoTs, following the symbolic intersection rule $a \cap b = \min(a, b)$.

3. **Inject entailments.** Entailed propositions that are not already stored can be synthesised as new symbolic activations (by composing the relation and operand embeddings) and recorded in the TruthLayer with their derived DoT.

This is worth implementing when the truth store is used for reasoning (verification, planning) rather than just retrieval. The geometric field alone is sufficient for soft consistency checks and nearest-truth lookups. The symbolic closure makes the store *deductively closed* – turning it from a database of assertions into something closer to a knowledge base with inference.

## 7.

# Radial Operators for Hypersphere Ternary Logic

**Ternary Logic Operators**

+1 = true 0 = unknown −1 = false

**NOT ( neg(a) = −a )**

| a | ¬a |
|---|---|
| +1 | −1 |
| 0 | 0 |
| −1 | +1 |

**NON ( non(a) = 0 )**

| a | non(a) |
|---|---|
| +1 | 0 |
| 0 | 0 |
| −1 | 0 |

**Intersection ( agree → min\|a,b\| )**

| a \ b | +1 | 0 | −1 |
|---|---|---|---|
| +1 | +1 | 0 | 0 |
| 0 | 0 | 0 | 0 |
| −1 | 0 | 0 | −1 |

**Union ( agree → max\|a,b\|; 0 absorbs )**

| a \ b | +1 | 0 | −1 |
|---|---|---|---|
| +1 | +1 | +1 | 0 |
| 0 | +1 | 0 | −1 |
| −1 | 0 | −1 | −1 |

**PART ( mereological, range [0, 1] )**

| a \ b | +1 | 0 | −1 |
|---|---|---|---|
| +1 | +1 | 0 | 0 |
| 0 | +1 | +1 | +1 |
| −1 | 0 | 0 | +1 |

*PART(a,b): 1 when a ⊆ b; 0 is part ofeverything; opposite signs → 0*

## Overview

RadMin, RadMax, NOT, and NON are four logical operators defined over a signed magnitude space where truth values range from -1 to +1. The semantic space is a hypersphere with zero at the origin representing **unknown**, +1 representing **true**, and -1 representing **false**.

The operators are designed to respect the geometric structure of this space: sign represents direction of assertion, magnitude represents strength of assertion, and zero represents the absence of assertion.

## Truth Values

| Value | Meaning |
|---|---|
| +1 | True (full positive assertion) |
| 0 | Unknown (no assertion) |
| -1 | False (full negative assertion) |

Intermediate values (e.g., +0.5, -0.3) represent partial assertions with varying degrees of confidence.

## Magnitude Ordering

The ordering relation ⊂ ("is a part of") is defined by magnitude:

$x \subset y$ iff $|x| < |y|$

This means "closer to zero" is "less than" in the radial sense. Zero is the weakest value; $\pm 1$ are the strongest.

---

## Operators

### RadMin (Radial Minimum – Conjunction / AND)

RadMin is a binary operator representing conjunction. It collapses toward zero on sign disagreement and takes the minimum magnitude on sign agreement.

**Definition:**

- If $\text{sign}(x) \neq \text{sign}(y)$: $\text{RadMin}(x, y) = 0$
- If $\text{sign}(x) = \text{sign}(y)$: $\text{RadMin}(x, y) = \text{sign}(x) \cdot \min(|x|, |y|)$

**Truth Table (Ternary):**

| | **+1** | **0** | **-1** |
|---|---|---|---|
| **+1** | +1 | 0 | 0 |
| **0** | 0 | 0 | 0 |
| **-1** | 0 | 0 | -1 |

**Properties:**

- Commutative: RadMin(x, y) = RadMin(y, x)
- Zero is absorbing: RadMin(x, 0) = 0 for all x
- Anti-diagonal symmetric
- Sign disagreement produces zero (contradiction → unknown)

---

### RadMax (Radial Maximum – Disjunction / OR)

RadMax is a binary operator representing disjunction. It collapses toward zero on sign disagreement and takes the maximum magnitude on sign agreement.

**Definition:**

- If $\text{sign}(x) \neq \text{sign}(y)$: $\text{RadMax}(x, y) = 0$
- If $\text{sign}(x) = \text{sign}(y)$: $\text{RadMax}(x, y) = \text{sign}(x) \cdot \max(|x|, |y|)$
- RadMax(x, 0) = x (zero is transparent / neutral for OR)

**Truth Table (Ternary):**

| | **+1** | **0** | **-1** |
|---|---|---|---|
| **+1** | +1 | +1 | 0 |
| **0** | +1 | 0 | -1 |
| **-1** | 0 | -1 | -1 |

**Properties:**

- Commutative: RadMax(x, y) = RadMax(y, x)
- Zero is transparent: RadMax(x, 0) = x for all x
- Anti-diagonal symmetric

- Sign disagreement produces zero (contradiction → unknown)

---

**NOT (Sign-Flipping Negation)**

NOT is a unary operator that inverts the sign of the assertion while preserving magnitude.

**Definition:**

NOT(x) = -x

**Truth Table (Ternary):**

| Input | Output |
|---|---|
| +1 | -1 |
| 0 | 0 |
| -1 | +1 |

**Properties:**

- Involutive: NOT(NOT(x)) = x
- Preserves magnitude: |NOT(x)| = |x|
- Zero is a fixed point: NOT(0) = 0

---

**NON (Non-Affirming Negation)**

NON is a unary operator that drives any assertion toward zero. It represents the withdrawal of assertion rather than the inversion of assertion.

**Definition:**

NON(x) = 0 for all x

**Truth Table (Ternary):**

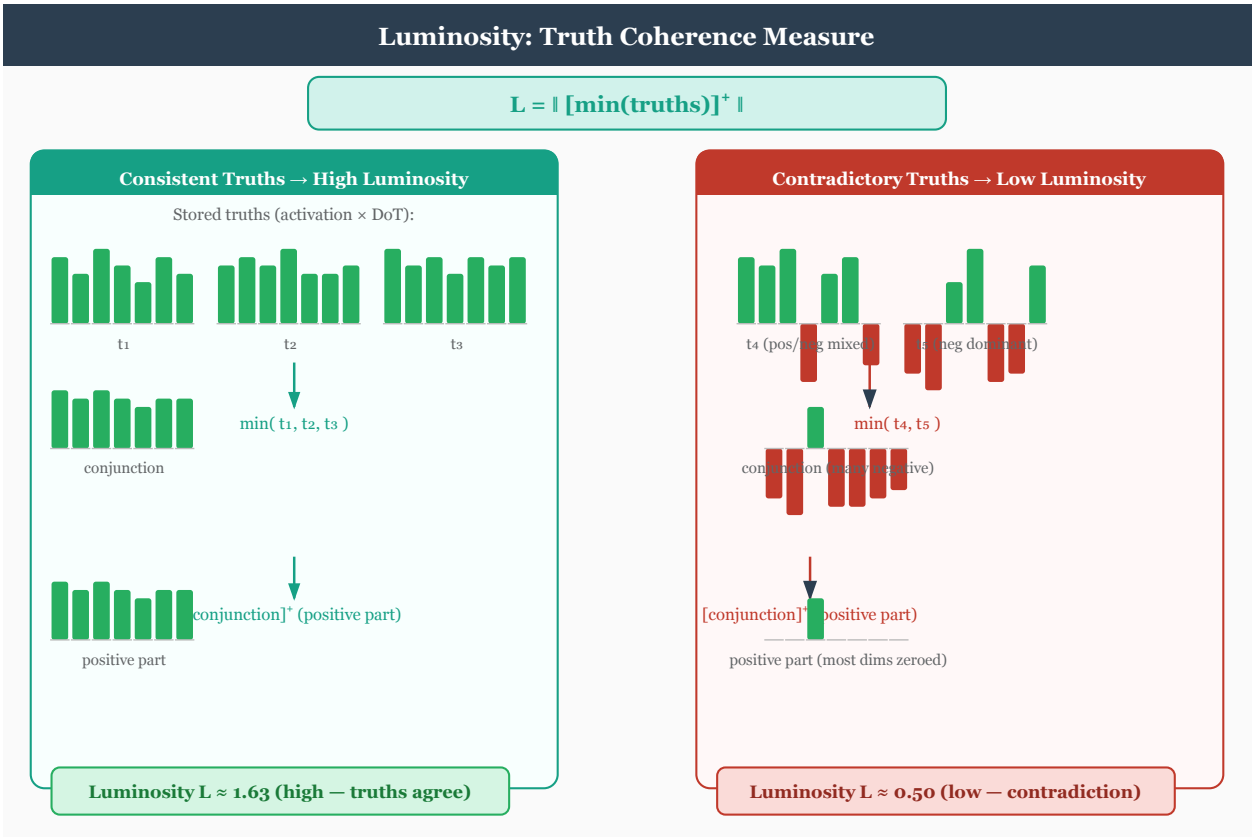
| Input | Output |
|---|---|
| +1 | 0 |
| 0 | 0 |
| -1 | 0 |

**Properties:**

- Absorbing: NON(x) = 0 for all x
- Idempotent: NON(NON(x)) = NON(x) = 0
- Semantically distinct from NOT: NON does not assert the opposite; it withdraws assertion entirely

---

## Luminosity




Luminosity measures the coherence of a truth set as a single scalar:

```
luminosity = ||relu(min(truths))||
```

The element-wise min across all stored truth activations computes the **conjunction** – the point where all truths agree. `relu` removes negative dimensions (darkness from conflicting truths), and the L2 norm gives the brightness.

- **High luminosity**: truths are coherent and mutually reinforcing.
- **Low luminosity**: truths are sparse, contradictory, or incoherent.

Luminosity serves two roles in the model:

1. **Top-down bias**: concept input is scaled by luminosity during each conceptual iteration: `concept_input * (1 + truthBiasScale * luminosity)`. A coherent truth set amplifies concept formation; an incoherent one leaves it unchanged.

2. **Loss modification**: low luminosity increases training loss, penalizing irrational propositions. See Ethics.md Section Universality for the full formula.

### Fusion

Fusion is the **mereological least upper bound** of the stored truth set: the elementwise max over every stored truth vector.

```
fusion = max_i truths[i]
```

In bivector space (paired-index `[p0, n0, p1, n1, ...]`), the fusion vector names the top-right corner of an axis-aligned bounding hyperrectangle — the smallest hyperrectangle dominating every stored truth componentwise. This is the geometric dual of luminosity:

- **Luminosity** = `||relu(min(truths))||` — the greatest lower bound (GLB / meet), scalar coherence. Answers *where do truths agree?*
- **Fusion** = `max(truths)` — the least upper bound (LUB / join), vector coverage. Answers *what region do truths collectively cover?*

Trust (DegreeOfTruth) is already baked into each stored truth via `record`: `stored[i] = activation_i * degree_i`. Fusion over trust-scaled truths therefore gives a trust-weighted LUB — a truth stored with `degree=0.3` contributes only `0.3 * activation` to the max.

**Layout caveat.** The TruthLayer's paired-index slicing (`[..., 0::2]` for positive poles, `[..., 1::2]` for negative) assumes a *repeated* bivector layout `[p0, n0, p1, n1, ...]`. The SymbolicSpace codebook uses a different layout — a *leading* bivector plus positional trailers: `[pos, neg, where..., when...]`. Callers that feed SymbolicSpace symbol activations into `luminosity()`, `tetralemma_balance_penalty()`, or related methods must slice the leading 2 dims first (`acts[..., :2]`) to isolate the bivector from positional content; see the `truth_loss` call-site in `basicmodel/bin/Spaces.py` for the canonical pattern.

### Supporting measures

- **Consistency** (`isConsistent()`): folds all stored truths into a union vector via successive `Basis.disjunction()` calls. In bitonic mode, conflicting +/- assertions on the same dimension cancel to zero, reducing the score.
- **Grounding** (`ground()`): finds the minimal subset of the TruthSet that entails a query activation. Uses partition-aware filtering and falls back to `TruthLayer.derive()` for indirect derivation.
- **TruthLoss**: an additive loss penalty for propositions that contradict stored truths, measured by union norm reduction via `Basis.disjunction()`. Coexists with the multiplicative luminosity modulation. See Reasoning Section TruthLoss.
- **Derive**: pairwise mereological inference via the Grammar's `part()` rule. When the parthood score between two truths exceeds a threshold, a new implied truth is recorded with attenuated DoT. Generalized by `extrapolate()` to all two-argument grammar methods.

---

## Semantic Summary

| Operator | Type | Role | Behavior on Contradiction |
|---|---|---|---|
| RadMin | Binary | Conjunction (AND) | Collapses to zero |
| RadMax | Binary | Disjunction (OR) | Collapses to zero |
| NOT | Unary | Sign-flipping negation | N/A |
| NON | Unary | Non-affirming negation | N/A |

## Open Questions

- **Functional completeness:** Do RadMin, RadMax, NOT, and NON form a functionally complete set over the ternary radial space?
- **Associativity:** Does RadMin(RadMin(a, b), c) = RadMin(a, RadMin(b, c)) hold for all combinations?
- **Duality:** RadMin and RadMax are not classical duals via NOT due to differing treatment of zero (absorbing vs. transparent). What is the formal relationship between them?
- **Extension to fuzzy domain:** In the continuous fuzzy extension (values in [-1, +1]), what is the behavior of NON? Does it drive toward zero continuously or collapse discretely?

---

## 8. Cross-references

The level-crossing operations (`lift`, `lower`, `intersection`, `union`), the `Pi/Sigma` layer ownership and ordering, the witness-set / slab-lattice geometry, and the `Ops.*` namespace inventory have moved out of this document. Logic.md is reserved for the 3-valued and 4-valued symbolic logics described in §§1-7 above.

For the moved material:

- **`Ops.*` reference** (every callable, current implementation, target layer anchor): Language.md → Ops Reference.
- **PiLayer / SigmaLayer ownership and direction** (which space owns which layer; `forwardPi` / `reversePi` / `forwardSigma` / `reverseSigma` aliases; the C ↔ S round trip): Spaces.md → ConceptualSpace and SymbolicSpace.
- **Mereological suite** (`part`, `whole`, `equal`, `overlap`, `underlap`, `boundary`, `copart` — vector and scalar forms, empty-set conventions): Mereology.md.
- **Tetralemma bivector layout** (`[aP, aN]` per position; TRUE/FALSE/BOTH/NEITHER corners): Spaces.md → ConceptualSpace activation carrier and BuddhistParallels.md.

# Reasoning System

The reasoning system provides four methods on `BaseModel` for truth-aware inference, a bidirectional reasoning loop, and a grammar learning mode. These build on the TruthLayer infrastructure described in Logic.md and the grammar composition described in Language.md.

## Partitioned Symbolic Space

The symbolic dimension is statically partitioned across conceptual orders using geometric decay. Each order writes only to its designated slice of `symbolSum`, while reading the full vector as feedback.

```
order 0:  [0,      D//2)       <- 1/2 of symbol_dim
order 1:  [D//2,   3D//4)      <- 1/4
order 2:  [3D//4,  7D//8)      <- 1/8
...
last order: remainder of D
```

This makes the symbolic space **self-describing**: the position of an activation reveals its conceptual order. Truth methods use `_activation_order()` to determine a query's order by finding the partition with the highest energy.

Partition boundaries are precomputed once at model creation via `BasicModel._order_partitions(symbol_dim, conceptual_order)`.

## Reasoning Methods

**`isConsistent() -> dict`**

Analyzes the TruthSet for internal consistency by folding all stored truths into a single summary vector via successive `Basis.disjunction()` calls. In bitonic mode, conflicting +/- assertions on the same dimension cancel to zero, reducing the score.

Returns `{'consistent': bool, 'score': float, 'sites': tensor, 'union_vector': tensor}`.

**`ground(activation, threshold=0.6) -> dict`**

Finds the minimal subset of the TruthSet that entails a query activation. Uses `_activation_order()` to filter truths by compatible partition before comparison. Falls back to `TruthLayer.derive()` for indirect derivation when direct grounding is insufficient.

Returns `{'grounded': bool, 'basis': [indices], 'trace': [...], 'confidence': float}`.

**`isTrue(activation) -> float`**

Grounds a proposition and returns a scalar Degree of Truth in [-1, 1]. Positive = true, negative = false, zero = unknown. Delegates to `ground()`.

**`extrapolate(grammar, seed_indices, max_new, attenuation) -> dict`**

Generalizes `TruthLayer.derive()` to all two-argument grammar methods (union, intersection, equals, part). For each pair of stored truths, applies every eligible method and accepts results that preserve or increase luminosity; rejects those that decrease it.

Accepted truths are recorded at `attenuation * min(DoT_i, DoT_j)`.

Returns `{'added': [indices], 'rejected': [(i, j, rule, delta_lum), ...]}`.

## TruthLoss

An additive loss penalty for false propositions, configured via `<TruthLoss>` in model.xml (default 0.0 = disabled).

TruthLoss measures the **union norm reduction** when a proposition is included in the TruthSet union via `Basis.disjunction()`:

```
truth_union = disjunction(all stored truths)
extended    = disjunction(truth_union, new_proposition)
penalty     = max(0, ||truth_union|| - ||extended||)
```

| Case | Effect |
|---|---|
| Agreeing proposition | Preserves or extends union dimensions -> no penalty |
| Unknown proposition (zero dims) | Passes through -> no penalty |
| Contradicting proposition | Cancels conflicting dimensions -> positive penalty |

DoT weighting is implicit: stored truth vectors carry DoT in their magnitude, so contradicting a high-DoT truth causes a larger norm drop.

TruthLoss is **additive** and coexists with the existing **multiplicative** luminosity modulation (`totalLoss * (1 + lum_weight * (1 - luminosity))`).

## Bidirectional Reasoning Loop

`BasicModel.reason(givens, target, direction, max_steps)` implements iterative reasoning in two directions:

**Forward** (givens -> conclusion): Encode givens into the TruthSet, extrapolate new truths each step, check `isTrue(target)` until the DoT exceeds threshold or `max_steps` is reached.

**Reverse** (target -> grounding): Encode target, call `ground()` to find a minimal basis, extrapolate if insufficient.

Luminosity non-decrease is the validity certificate at each step.

## Grammar Learning

`BasicModel.grammar_learning_step()` learns grammar weights from a symbolic reconstruction objective:

1. Forward pass produces `symbolSum`

2. Reverse pass reconstructs input
3. Re-encode reconstruction to `symbolSum_hat`
4. Loss = `||symbolSum_hat - symbolSum||^2` (symbolic level, not conceptual)
5. Optional luminosity validity penalty for rules that decrease luminosity

Grammar weights emerge from gradient descent. Paraphrase-invariance holds because semantically similar sentences snap to nearby codebook entries.

## Architecture Modes: useButterflies × useGrammar

Two knobs – `<useButterflies>` (under `<architecture>`) and `<useGrammar>` (under `<WordSpace>`) – select among the Sigma-Pi architectures. Full grammar mode (`useGrammar="all"`) and butterflies are mutually exclusive: butterfly permutations fight the constituency structure that grammar composition requires, so that combination is rejected at load time. Shamatha Speech is a target narrow-grammar mode: it is not the full constituency grammar and should be wired as a DNF-object policy with contiguity checks.

| | **useGrammar="none"** | **useGrammar="shamathaSpeech" target** | **useGrammar="a** |
|---|---|---|---|
| **useButterflies=false** | Flat shared sigma (default) | DNF object grammar + contiguity | Grammar-directed |
| **useButterflies=true** | Pairwise butterfly mixing | TBD shape/policy decision | [x] excluded |

### Flat (both false)

A single shared PiLayer (P↔C) and SigmaLayer (C↔S) operate on a concatenated `[percepts, symbols]` tensor at every conceptual order. All orders share one undifferentiated symbolic space. Sufficient for `conceptualOrder=1`.

At each iteration `t`:

1. **Input construction.** Percepts and symbol feedback are **concatenated** along the vector dimension: `concept_input = cat([percepts, sym_feedback], dim=1)`.
2. **Pi** (`ConceptualSpace.pi`): the single shared PiLayer transforms percepts → concepts via `forwardPi`.
3. **Sigma** (`SymbolicSpace.sigma`): the single shared SigmaLayer projects concepts → symbols via `forwardSigma`. All orders write to the entire symbol dimension.
4. **Feedback**: Symbol activation norms are broadcast back to the symbol portion of the input for the next iteration.
5. **Reverse**: `SymbolicSpace.reverse` (`reverseSigma`) → peel off the symbol portion → `ConceptualSpace.reverse` (`reversePi`), repeated per order.

Key property: **all conceptual orders share one undifferentiated symbolic space**. There is no way to tell, from a symbol vector alone, which order produced it.

### Butterfly (useButterflies=true, useGrammar="none")

Butterfly-mode Sigma and Pi layers inherit `ButterflyLayer` helpers that permute inputs, pack adjacent pairs, apply the layer, unpack, and merge – halving `N` at each conceptual order while keeping `D` constant. The merge is internal to the layer (no external `_butterfly_merge` / `_butterfly_unmerge` stack); the reverse path inverts each stage exactly. `<reconstruct>symbols</reconstruct>` is required so the full symbol state is available for exact inversion.

Analogous to increasing receptive fields in visual cortex (V1→V2→V4→IT), the pairwise mixing lets information flow across the slot axis – making this path suitable for tasks like XOR where information at different input slots must collide.

**Grammar-directed (useButterflies=false, useGrammar="all")**

Progressive-bottleneck path with an external pair-average merge (`_butterfly_merge` on the vector axis, caching `left - right` diffs in `_merge_diffs`), per-level indexed Sigma/Pi (`conceptualSpace[t]` / `symbolicSpace[t]`), and cached symbol feedback (`_sym_feedbacks`). The symbol dimension is geometrically partitioned so each order writes only to its slice – gives **partition-aware reasoning**: truth grounding, consistency checks, and extrapolation that respect which conceptual order a proposition belongs to.

Forward loop:

```
for t in range(conceptualOrder):
    x = butterfly_merge(x)               # halve vector count (external)
    x = x + sym_feedback                 # additive feedback
    concepts = conceptualSpace[t](x)     # per-level sigma
    symbols  = symbolicSpace[t](concepts)  # per-level pi
    sym_feedback = symbols.norm(...)     # for next iteration
```

Reverse loop:

```
x = symbolicSpace[last].reverse(sym_vec)
for t in reversed(range(conceptualOrder)):
    x = conceptualSpace[t].reverse(x)
    x = x - cached_feedback[t]           # undo additive feedback
    x = butterfly_unmerge(x)             # restore vector count
```

The butterfly unmerge uses the cached `_merge_diffs` to recover both original vectors from each averaged pair – the inverse is exact.

## Configuration

| Parameter | Location | Default | Description |
|---|---|---|---|
| `<TruthLoss>` | `<training>` in model.xml | 0.0 | Weight for additive truth loss penalty |
| `<conceptualOrder>` | `<architecture>` | 1 | Number of Percept->Concept->Symbol iterations |
| `<useButterflies>` | `<architecture>` | false | Pairwise butterfly mixing with N-halving per conceptual order |
| `<useGrammar>` | `<WordSpace>` | false | Grammar-directed composition with progressive bottleneck |
| `truthMinMagnitude` | `<SymbolicSpace>` | 0.3 | Activation-norm cap that drives the per-cell trust score in `Tru` |

## Contemplative Awareness Methods

Four stub methods on `BaseModel` characterize stages of contemplative awareness as spatial/computational properties of the model state. Each raises `NotImplementedError` – they define the target characterization, not an implementation.

| Method | Stage | Characterization |
|---|---|---|
| `Contiguous()` | One-Pointedness (Shamatha / FA) | Current state occupies a single connected, convex region in |
| `Continuous()` | Simplicity (Continuity / OA) | Concept states flow continuously without discrete jumps; th |
| `Peaceful()` | One Taste (Emotional Symmetry) | TruthLayer luminosity is uniformly high across all stored pr |
| `Done()` | Buddhahood (Non-Meditation / Resonance) | The model is a fixed point of its own forward-reverse cycle; |

Shamatha Speech is the target grammar mode for `Contiguous()`. It is a complete DNF object grammar plus a spatiotemporal contiguity check: every `conjunction` / `disjunction` over object parts must keep the `where()` support connected and the `when()` support continuous. The mode differs from serial mode because it may reduce over all active percepts at once; it rejects scattered object fields, not multi-percept fields. The

legacy `thoughtFree` flag should be treated as an alias / incomplete hook, not the final implementation. See Language.md §Shamatha Speech Mode and plans/2026-04-28-shamatha-speech-contiguity-handoff.md.

### Testing

Unit tests in `basicmodel/test/test_reasoning.py` cover all methods without requiring a trained model. English-level tests (syllogisms, contrapositives, semantic equivalence) are `@pytest.mark.xfail` until word identity is learned through training.

# Training

## Overview

BasicModel training has two phases: **embedding pretraining** and **network training**. Embedding pretraining builds word vectors from a large corpus. Network training uses those vectors as the input representation and learns to predict and reconstruct sentences.

The two phases can overlap: the `<trainEmbedding>` config controls whether and how embeddings continue to evolve during network training.

---

## Phase 1: Embedding Pretraining (`make train` / `embed.py train`)

Produces a static embedding artifact (e.g. `BasicModel.kv`) containing word vectors. The output path is read from `<embeddingPath>` in the XML config.

### Pipeline

```
FineWeb-EDU parquet shards
    -> Pass 1: stream documents, lex + parse, count words, build vocabulary
    -> Pass 2: stream documents again, train SBOW per sentence, discard examples
    -> Save WordVectors to sentence.pt
```

### SBOW (Sentence Bag of Words)

SBOW is a sentence-level variant of CBOW. Given a sentence of N words with embedding vectors and a linear output matrix with bias:

Leave-one-out centroid for word i:

$$c_i = \frac{1}{N-1} \sum_{j \neq i} v_j$$

Logits (unnormalised scores for every word in vocabulary V):

$$z_i = W_o \, c_i + b_o$$

Softmax probability that centroid predicts word i:

$$P(w_i \mid c_i) = \frac{\exp(z_{i,w_i})}{\sum_{k=1}^{V} \exp(z_{i,k})}$$

Loss (cross-entropy, averaged over all N sentence positions):

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N} \log P(w_i \mid c_i)$$

The gradient of the loss with respect to embedding v_i has two terms:

- **Attractive** (from position i where w_i is the target): pulls v_i toward directions that increase its probability.
- **Repulsive** (from positions $j \neq i$ where $v_i$ is part of the centroid): pushes v_i away from centroids whose targets are other words.

The softmax partition function ensures that all V words compete: any word k too close to a centroid it doesn't belong to increases the denominator and is pushed away with force proportional to its softmax probability. This adaptive repulsion distributes vectors across the embedding space.

This gives N positive and N × (V-1) repulsive updates per sentence. Every word in the sentence is a positive example (unlike CBOW which picks one target).

```
vecs = embedding(idx)                          # [N, dim]
total = vecs.sum(dim=0)                        # [dim]
centroids = (total - vecs) / (N - 1)           # [N, dim]  leave-one-out
logits = linear(centroids)                     # [N, vocab] full softmax
loss = cross_entropy(logits, idx)
```

### CBOW (Continuous Bag of Words)

CBOW predicts a single target word from its context. Given a sentence of $N$ words, for target word $w_i$ with context $C_i = \{w_j : j \neq i\}$:

Context mean (with padding for variable-length contexts):

$$\bar{c}_i = \frac{1}{|C_i|}\sum_{j \in C_i} v_j$$

The loss is the same negative-sampling objective as SBOW but applied per-word to the padded context mean rather than the leave-one-out centroid:

$$\mathcal{L}_{\text{CBOW}} = -\log \sigma(\bar{c}_i^\top v_{w_i}) - \sum_{k=1}^{K} \log \sigma(-\bar{c}_i^\top v_{w_k^-})$$

where $K$ is the number of negative samples and $w_k^-$ are randomly sampled words.

### Two-Pass Architecture

The trainer uses two passes over the data to avoid dynamic model resizing:

- **Pass 1 (vocabulary):** Stream all documents, count every word via lex + parse. Words meeting `min_count` are promoted to the trainable vocabulary. The model (embedding + linear head) is allocated once at the final vocabulary size.
- **Pass 2 (training):** Stream documents again. For each sentence, run one SBOW step and immediately discard the examples. No examples are accumulated in memory. Multiple epochs re-stream the data.

### Configuration (Makefile variables)

| Variable | Default | Description |
|---|---|---|
| `BASIC_SHARDS` | 1 | Number of FineWeb-EDU parquet shards |
| `BASIC_MAX_DOCS` | 10000 | Maximum documents to process |
| `BASIC_VEC_SIZE` | 100 | Embedding dimensionality |
| `BASIC_EPOCHS` | 10 | Training epochs |
| `BASIC_MIN_COUNT` | 10 | Minimum word count for vocabulary inclusion |
| `BASIC_BATCH_SIZE` | 8 | Batch size (unused by SBOW; per-sentence training) |

### Memory Considerations

The dominant memory cost is the linear output head: `vocab_size * vector_size` parameters plus optimizer state. With 200K vocabulary and 100 dimensions:

- Embedding: 200K × 100 × 4 bytes = ~80MB
- Linear head: 100 × 200K × 4 bytes = ~80MB
- Adam optimizer: 2× momentum buffers = ~320MB total

The full softmax computes `(N, vocab_size)` logits per sentence. For large vocabularies this can be expensive. Training runs on CPU by default (`BASICMODEL_DEVICE=cpu`) to avoid MPS/GPU memory limits.

---

## Phase 2: Network Training (`BasicModel.py`)

The network learns to predict and reconstruct sentences using the pretrained embeddings as its input representation.

### Masked Prediction Modes

The `maskedPrediction` config controls the prediction objective:

| Mode | Input Masking | Truncation | Target | Description |
|---|---|---|---|---|
| `NONE` | None | No | Dataset labels | Standard supervised |
| `IR` | Zero word i | No | Embedding of word i | Bidirectional input reconstruction (non-AR si |
| `AR` | Zero word i | Yes (j > i zeroed) | Embedding of word i | Autoregressive (GPT-style) |
| `ARUS` | Same as AR | Yes | Zero vector | Unsupervised (loss suppressed) |
| `ARIR` | Per-pos | Yes | Embedding + reconstruction | Autoregressive iterative reconstruction |

### `<trainEmbedding>`: Embedding Update Mode

The `trainEmbedding` config controls whether and how embeddings evolve during network training. When embedding updates are enabled (CBOW, SBOW, or BOTH), this implements an EM-like alternation:

- **E-step (network):** Forward pass + masked prediction + backward + optimizer step. The network learns to predict/reconstruct with the current embedding space.
- **M-step (CBOW/SBOW):** After each network step, run one embedding update on the same sentence. The embedding vectors move to better capture co-occurrence.

| Value | Embedding method | Model layers | Description |
|---|---|---|---|
| `NONE` | Frozen | Trained | Embeddings frozen; only model layers train |
| `CBOW` | CBOW (padded context) | Trained | CBOW reshapes embeddings via own optimizer; model layers train s |
| `SBOW` | SBOW (centroid) | Trained | Faster variant: predict word from leave-one-out centroid |
| `BACKPROP` | Backprop only | Trained | Codebook trained purely via model loss gradients; no SBOW/CBOV |

| Value | Embedding method | Model layers | Description |
|---|---|---|---|
| `BOTH` | SBOW post-batch | Trained | Two optimizers: SBOW updates embeddings, Adam updates model |
| `JOINT` | Single backward | Trained | Single optimizer: combined model + SBOW loss |

### Gradient Flow Through Codebook

The `<trainEmbedding>` mode determines whether the codebook weight tensor is **detached** during forward/reverse passes and whether it appears in the main optimizer's parameter list.

| `trainEmbedding` | `detach()` in forward/reverse | In main optimizer | Embedding method |
|---|---|---|---|
| `NONE` | Yes – codebook is a frozen lookup | No | Frozen |
| `CBOW` | Yes – codebook is a frozen lookup | No | CBOW (own optimizer) |
| `SBOW` | Yes – codebook is a frozen lookup | No | SBOW (own optimizer) |
| `BACKPROP` | No – gradients flow through codebook | Yes | Model loss only |
| `BOTH` | No – gradients flow through codebook | Yes | SBOW + backprop |
| `JOINT` | No – gradients flow through codebook | Yes | Single combined loss |

**`NONE` is the recommended mode for constrained hardware** (e.g. Apple MPS with <16GB). The codebook from Phase 1 is frozen, all memory and compute go to the model layers.

**`CBOW/SBOW`** maintains clean EM separation: CBOW/SBOW reshapes the embedding space using its own optimizer. No gradients flow from the model through the codebook.

**`BACKPROP`** is the simplest trainable mode. The codebook participates in the main optimizer and receives gradients from the model's output and reconstruction losses, but no embedding-specific loss (SBOW/CBOW) is applied. The embedding space is shaped entirely by the task. Best for small vocabularies or when the pretrained embedding geometry should adapt freely to the downstream objective.

**`BOTH` risks gradient interference.** The network's reconstruction loss pulls embeddings toward being maximally distinguishable (spreading apart), while SBOW pulls co-occurring words together. These forces can conflict because they use separate optimizers with separate momentum buffers.

### `JOINT`: Single Combined Loss

JOINT mode avoids the EM alternation of `BOTH` by computing a single combined loss before one backward pass:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{model}} + \lambda \cdot \mathcal{L}_{\text{SBOW}}$$

where $\mathcal{L}_{\text{model}}$ is the standard output loss (plus reconstruction loss if `reversible=true`), and $\lambda$ is the `trainEmbeddingRatio` hyperparameter (default 0.1).

The SBOW loss for the current sentence is computed using the same negative-sampling objective as Phase 1:

$$\mathcal{L}_{\text{SBOW}} = -\frac{1}{N}\sum_{i=1}^{N}\left[\log\sigma(c_i^\top v_{w_i}) + \sum_{k=1}^{K}\log\sigma(-c_i^\top v_{w_k^-})\right]$$

where $c_i$ is the leave-one-out centroid, $v_{w_i}$ is the target word's embedding, and $w_k^-$ are negative samples drawn uniformly from the vocabulary.

**Advantages over `BOTH`:**

- **One optimizer, one momentum buffer.** Adam sees the full gradient landscape from both objectives simultaneously, avoiding the oscillation risk of two independent optimizers pulling embeddings in different directions.
- **Gradient coherence.** The model's MSE loss and the SBOW co-occurrence loss are balanced by $\lambda$ before any parameter update, so the step direction reflects the intended trade-off.
- **Simpler implementation.** No post-batch embedding step; everything flows through `totalLoss.backward()`.

**Trade-off:** Both objectives share the same learning rate and Adam state. If the two loss surfaces have very different curvature, `trainEmbeddingRatio` may need tuning. The `BOTH` mode's separate optimizers can use different learning rates.

### Why MSE over Embeddings (not Cross-Entropy over Vocabulary)

The network's output loss uses MSE against the target word's embedding vector, not cross-entropy over vocabulary indices. This is an architectural advantage:

- A prediction near "cat" when the target is "kitten" incurs less loss than a prediction near "democracy" – the embedding space captures semantic similarity.
- Output dimensionality is the embedding dimension (~100) rather than the vocabulary size (~200K), reducing parameter count.

### Training Loop

Data flows through a `SentenceStreamDataset` wrapped in a PyTorch `DataLoader`. The ordered training list is split into `B = batchSize` contiguous slabs of length `L = len(split) // B`; at step `t` the loader yields a B-wide batch where row `b` is item `b * L + t`. Each batch row therefore carries its own document-order stream, so temporal context is coherent across steps. No per-epoch global shuffle is applied. `numWorkers > 0` enables async prefetch.

For each B-wide batch in AR modes (`AR` / `ARUS` / `ARIR`):

1. `BasicModel.Start(inputData)` cascades reset through every Space and Layer (clearing per-sentence scratch).
2. `InputSpace.forward()` lexes/embeds the input once into `[B, T, D]`, then builds all K progressive-prefix windows in a single `tensor.unfold(1, N, 1)` – left-padding N zeros so window `k` is `k` zero-pad slots followed by `emb[0..k-1]` right-aligned. The emitted subspace carries `event: [B, K, N, D]` with `k_axis=True` and a `[B, K]` validity mask (NULL = all-zero target embedding). The body consumes the K axis as microbatch (flattened to `B*K`) and the head produces all K predictions in one pass.
3. For `ARIR` only, a single terminal `reverse(symbols)` after the body produces a `[B, N, D]` reconstruction.
4. `runBatch` computes the loss once via `TheError.add`:
   - `AR`: output prediction only (K per-row predictions, masked by validity).
   - `ARUS`: no output term (suppressed); no reconstruction.
   - `ARIR`: output prediction + reconstruction, weighted by `reverseScale`.
5. One `backward()` + `optimizer.step()` per DataLoader yield. The single-unfold microbatch path replaces the earlier N-pass cursor loop, collapsing the N forward/reverse calls into one.
6. Embedding training (`trainEmbedding` in `CBOW/SBOW/BOTH`) runs once per batch after the forward/backward cycle.

Non-AR modes (`maskedPrediction=NONE`) do a single forward/backward/step per B-wide batch through the same `runBatch` code path, skipping the outer pos loop.

Mode contract:

| Mode | Per-pos output | Terminal reverse | Reconstruction loss |
|---|---|---|---|
| `AR` | Yes | No (ignores `<reconstruct>`) | Not trained |
| `ARUS` | No | No (ignores `<reconstruct>`) | Not trained |
| `ARIR` | Yes | Yes | Over `[B, N, D]` |

`<maskedPrediction>ARIR</maskedPrediction>` requires `<reconstruct>` to be something other than `NONE` at validation.

---

## SBOW vs CBOW

| Property | CBOW | SBOW | Masked Prediction (Pha |
|---|---|---|---|
| Targets per sentence | 1 (pick one word) | N (every word) | N (one per masked posi |
| Context | Other N-1 words | Leave-one-out centroid of N-1 | All unmasked words (+ |
| Positive updates | 1 per step | N per step | N per step |
| Repulsive force | vocab-1 implicit via softmax | N × (vocab-1) via softmax | Implicit via MSE in emb |
| Signal density | Low (1 gradient per sentence) | High (N gradients per sentence) | High (N gradients per se |
| Loss function | Cross-entropy over vocabulary | Cross-entropy over vocabulary | MSE over embedding ve |
| Updates embeddings directly | Yes (own optimizer) | Yes (own optimizer) | Only for `BACKPROP`/`BOTH` |
| Used in (`<trainEmbedding>`) | `CBOW` | `SBOW`, `BOTH`, `JOINT` | `BACKPROP`, `BOTH`, `JOINT` |

SBOW provides richer signal per sentence because every word acts as both context (for other words) and target (predicted from others). The full softmax over the vocabulary at each position ensures vectors distribute across the hypersphere: words too close to centroids they don't belong to are pushed away with force proportional to their proximity.

---

## Three-File Architecture

Model behaviour is partitioned across three independently managed artifacts:

| Artifact | Example | Contents | Updated by |
|---|---|---|---|
| XML config | `data/BasicModel.xml` | Architecture, hyperparameters, file paths | Hand-edited |
| Embedding | `data/BasicModel.kv` | Word vectors ($V \times d$ codebook matrix) | Phase 1: `embed.py` |
| Weights | `data/BasicModel.ckpt` | Model layer parameters (attention, Pi/Sigma weights) | Phase 2: `BasicModel.py` |

The checkpoint explicitly excludes embedding parameters (`wv._vectors`). This separation enables independent iteration:

- Retrain embeddings without touching model weights (`--force-embeddings`)
- Retrain model with frozen codebook (`<trainEmbedding>NONE</trainEmbedding>`)
- Swap codebooks between models sharing the same architecture

---

## Embedding Space Geometry

The full softmax provides a distributional guarantee that random negative sampling cannot match. For predicting word w_i from centroid c_i, the loss is:

$$\mathcal{L}_i = -\log \frac{\exp(z_{i,w_i})}{\sum_{j=1}^{V} \exp(z_{i,j})}$$

where the logits are:

$$z_i = W_o\, c_i + b_o$$

The partition function (denominator) sums over every word in the vocabulary. The gradient with respect to the logit for any non-target word $k \neq w_i$ is:

$$\frac{\partial \mathcal{L}_i}{\partial z_{i,k}} = P(k \mid c_i)$$

Words with high softmax probability – those too close to the centroid – receive the strongest repulsive gradient. Words already far away have near-zero probability and are left alone. This adaptive repulsion naturally spreads vectors across the hypersphere.

At equilibrium, no word can move closer to any centroid without increasing the loss. This is the key advantage over approaches like direct push/pull with random negatives, where most sampled negatives are already far away on a high-dimensional hypersphere and contribute little useful gradient.

---

### Profiling

The `--profile` flag wraps Phase 2 in `cProfile`:

```
make train_micro_remote  # add --profile via train.py
# Or directly:
python bin/train.py --model data/BasicModel.xml --profile --max-docs 500
```

Output: a `.prof` file in `output/profiles/` plus a top-30 summary printed to stdout. View interactively with `snakeviz`:

```
pip install snakeviz
snakeviz output/profiles/train_20260319_111441.prof
```

For live profiling of a running process, `py-spy` can attach by PID (requires root on macOS):

```
pip install py-spy
sudo py-spy record -o profile.svg --pid <PID> --duration 30
```

## Ergodic Exploration via Adaptive Bias-Variance Control

BasicModel implements a novel approach to neural network optimization that decouples **what** the network learns (weights $W$) from **how much it explores** (a scalar $\alpha$ governing the bias-variance tradeoff). Two fundamentally different algorithms operate in tandem:

| Component | Algorithm | What it does |
|---|---|---|
| Weights $W$ | Standard Adam | Gradient descent on the loss landscape |
| Tradeoff $\alpha$ | Gradient Energy Sensor | Measures loss-surface curvature to set exploration level |

The key insight: $\alpha$ **is not trained by gradient descent.** It is a *sensor* that reads the gradient energy flowing through the temperature parameter and converts it into an exploration policy.

## 1. The Ergodic Weights

In this model, weights are not treated as fixed constants. Each layer instead uses an effective weight matrix sampled from a locally specified ergodic distribution, with the learned tensor and the stochastic tensor defining the current mixture:

$$W_{\text{eff}} = \alpha \cdot W + (1 - \alpha) \cdot \varepsilon, \qquad \varepsilon \sim \mathcal{N}(0, I)$$

So the layer never acts on a bare, timeless weight matrix; it acts on the current sample $W_{\text{eff}}$, while $W$ stores the learned structure that increasingly shapes that local distribution as $\alpha$ rises. This perspective is adjacent to the broader literature on stochastic sampling in learning, including Monte Carlo methods, stochastic approximation, and noise-injected views of neural network optimization.

We define two derived quantities from the single scalar $\alpha$:

$$\text{bias} = \alpha, \qquad \text{var} = 1 - \alpha$$

This is a **direct encoding of the bias-variance tradeoff**:

- $\alpha = 0$: Pure exploration. $W_{\text{eff}} = \varepsilon$. The network is random noise.
- $\alpha = 1$: Pure exploitation. $W_{\text{eff}} = W$. The network uses only learned weights.
- $0 < \alpha < 1$: Interpolation between learned structure and stochastic exploration.

Training begins at $\alpha = 0$ (full exploration) and the system autonomously transitions toward exploitation as the loss landscape flattens.

## 2. Why Standard Adam Fails on $\alpha$

The temperature parameter $T = 1 - \alpha$ receives gradients via:

$$\frac{\partial \mathcal{L}}{\partial T} = \sum_{i,j} \frac{\partial \mathcal{L}}{\partial (W_{\text{eff}})_{ij}} \cdot \varepsilon_{ij}$$

Because $\varepsilon$ is **re-sampled every forward pass**, this gradient is **zero-mean**:

$$\mathbb{E}\left[\frac{\partial \mathcal{L}}{\partial T}\right] = \sum_{i,j} \frac{\partial \mathcal{L}}{\partial (W_{\text{eff}})_{ij}} \cdot \underbrace{\mathbb{E}[\varepsilon_{ij}]}_{=0} = 0$$

Standard Adam maintains a first moment (running mean) and a second moment (running variance), then computes the update as $m/\sqrt{v}$. For a zero-mean gradient:

$$m_t \approx 0 \quad \Longrightarrow \quad \frac{m_t}{\sqrt{v_t}} \approx \frac{0}{\sqrt{v_t}} = 0$$

**Adam produces no update.** The first moment kills the signal. This is correct behavior for Adam — it correctly identifies that there is no consistent gradient direction — but it means Adam cannot be used to tune $\alpha$.

## 3. The Gradient Energy Sensor

### 3.1 Modified Second-Moment Estimator

We discard the first moment entirely and use only the second moment, but as **output** rather than normalizer:

$$v_t = \beta \cdot v_{t-1} + (1 - \beta) \cdot g_t^2$$

where $g_t = \partial \mathcal{L} / \partial T$ is the temperature gradient at step $t$, and $\beta = 0.999$ is the EMA decay rate.

### 3.2 Bias Correction

Identical to Adam's bias correction. Since $v_0 = 0$, early estimates are biased toward zero:

$$\hat{v}_t = \frac{v_t}{1 - \beta^t}$$

### 3.3 What $\hat{v}_t$ Measures

The second moment of a zero-mean random variable is its variance:

$$\mathbb{E}[g_t^2] = \mathrm{Var}(g_t) = \sum_{i,j} \left( \frac{\partial \mathcal{L}}{\partial (W_{\text{eff}})_{ij}} \right)^2 = \left\| \frac{\partial \mathcal{L}}{\partial W_{\text{eff}}} \right\|_F^2$$

This is the **squared Frobenius norm** of the loss gradient with respect to the effective weights — a scalar measure of how much the loss surface cares about weight perturbations. We call this the **gradient energy**.

| Gradient energy | Meaning | Desired behavior |
|---|---|---|
| High $\hat{v}_t$ | Loss is sensitive to weight changes | Keep exploring (low $\alpha$) |
| Low $\hat{v}_t$ | Loss surface is flat | Exploit learned weights (high $\alpha$) |

### 3.4 The Alpha Update Rule

$$\alpha_t = \frac{1}{1 + \tau \cdot \sqrt{\hat{v}_t}}$$

where $\tau$ (`global_temp`) is a tunable sensitivity knob.

**Properties:**

- $\sqrt{\hat{v}_t} \to 0 \implies \alpha_t \to 1$ (exploit when gradients vanish)
- $\sqrt{\hat{v}_t} \to \infty \implies \alpha_t \to 0$ (explore when gradients are large)
- $\alpha_t \in (0, 1)$ always — the sigmoid-like shape prevents saturation
- $\tau$ controls the transition speed: large $\tau$ means more exploration at equivalent gradient energy

After computing $\alpha_t$, the bias and variance are derived:

$$\mathrm{bias}_t = \alpha_t, \qquad \mathrm{var}_t = 1 - \alpha_t$$

### 3.5 Layer-Local Certainty (Per-Neuron Sigma)

Each ergodic layer maintains per-neuron **sigma** — the running variance of its gradient energy — tracked via Welford's algorithm in `observe_sigma()`. Sigma drives per-neuron bias and var:

$$\text{var}_j = \frac{\sigma_j}{\sigma_j + \kappa}, \qquad \text{bias}_j = 1 - \text{var}_j$$

Low sigma (consistent gradient, found a minimum) yields high bias, low var. High sigma (unstable gradient) yields low bias, high var.

The `set_sigma(sigma)` method controls the sensitivity knob $\kappa$:

- `sigma=1`: encourage exploration (low $\kappa$, responsive to gradient variance)
- `sigma=0`: suppress exploration (high $\kappa$, var $\approx 0$)

---

## 4. Comparison: Adam vs. Gradient Energy Sensor

| Property | Adam (for $W$) | Gradient Energy Sensor (for $\alpha$) |
|---|---|---|
| **Type** | Optimizer (gradient descent) | Sensor (measurement) |
| **First moment $m_t$** | Yes — tracks gradient direction | No — direction is zero-mean, useless |
| **Second moment $v_t$** | Yes — normalizes step size | Yes — **is the output** |
| **Role of $\sqrt{v_t}$** | Denominator (adaptive learning rate) | Numerator (gradient energy estimate) |
| **Update rule** | $W \leftarrow W - \eta \cdot m/\sqrt{v}$ | $\alpha \leftarrow 1/(1 + \tau\sqrt{\hat{v}})$ |
| **Descent?** | Yes — follows gradient downhill | No — maps energy to a policy |
| **Zero-mean safe?** | No — produces $0/\sqrt{v} = 0$ | Yes — by design |

---

## 5. Derivation: Why Zero-Mean Implies Drop $m_t$

Let $g_t = \nabla_T \mathcal{L}$ be the temperature gradient. Since noise $\varepsilon$ is i.i.d. each step:

$$g_t = \sum_{i,j} \frac{\partial \mathcal{L}}{\partial (W_{\text{eff}})_{ij}} \cdot \varepsilon_{ij}^{(t)}$$

The first moment EMA is:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

Taking expectations:

$$\mathbb{E}[m_t] = \beta_1 \mathbb{E}[m_{t-1}] + (1 - \beta_1) \underbrace{\mathbb{E}[g_t]}_{=0} = \beta_1 \mathbb{E}[m_{t-1}]$$

By induction: $\mathbb{E}[m_t] = \beta_1^t \mathbb{E}[m_0] = 0$. The first moment converges exponentially to zero. It carries **no information** about the loss landscape — only noise from finite sampling. Keeping it would add variance to the estimator without adding signal.

The second moment, however, converges to a meaningful quantity:

$$\mathbb{E}[v_t] \to \mathbb{E}[g_t^2] = \left\|\nabla_{W_{\text{eff}}}\mathcal{L}\right\|_F^2$$

This is exactly the gradient energy — the quantity we need.

---

## 6. Dropout via Bias

The `bias` and `var` parameters support **dropout regularization**:

$$\text{bias}_t^{(\text{drop})} = \begin{cases} 0 & \text{with probability } p \\ \alpha_t & \text{with probability } 1-p \end{cases}$$

When bias is dropped to zero, the layer acts as pure noise regardless of $\alpha$. This is analogous to standard dropout but operates on the bias-variance tradeoff rather than individual activations. The temperature $T = 1 - \alpha$ remains unchanged so that the noise contribution is consistent.

Seen this way, dropout is just a schedule on bias. Ordinary Bernoulli dropout is the binary case in which the bias is either kept at $\alpha_t$ or forced to 0 for that step, while annealed dropout corresponds to increasing the expected bias over training so the layer spends less time in pure-noise mode and more time exploiting learned structure. More generally, nonzero temperature bakes regularization directly into the model because every forward pass perturbs the effective weights, discouraging brittle co-adaptation and rewarding structure that survives stochastic variation.

---

## 7. The `global_temp` Sensitivity Knob

The parameter $\tau$ (`global_temp`) scales how responsive $\alpha$ is to gradient energy:

$$\alpha = \frac{1}{1 + \tau \cdot \sqrt{\hat{v}}}$$

| $\tau$ | Effect |
|---|---|
| $\tau = 0$ | $\alpha = 1$ always — pure exploitation, no exploration |
| $\tau \ll 1$ | Aggressive exploitation — quickly converges to $\alpha \approx 1$ |
| $\tau = 1$ | Balanced — $\alpha$ tracks gradient energy on a natural scale |
| $\tau \gg 1$ | Conservative — maintains high exploration even with moderate gradients |

In practice, $\tau$ can itself be scheduled (e.g., annealed from high to low) to implement a coarse exploration-to-exploitation curriculum.

---

## 8. Initialization and Bootstrap

**Problem:** If we initialized at $\alpha = 1$ (pure exploitation), the noise term vanishes and the temperature gradient $\partial\mathcal{L}/\partial T = 0$. The sensor would read zero gradient energy and keep $\alpha = 1$ forever — an **exploitation trap**.

**Solution:** Initialize at $\alpha = 0$ (pure exploration).

At $\alpha = 0$:

- $W_{\text{eff}} = \varepsilon$ — pure noise carries gradients through the network

- The temperature gradient is nonzero: $g_t = \sum_{ij}(\partial\mathcal{L}/\partial\varepsilon_{ij}) \cdot \varepsilon_{ij} \neq 0$
- The sensor measures initial gradient energy and begins tuning $\alpha$ upward
- As $W$ learns useful structure, gradient energy decreases and $\alpha$ rises naturally

In this architecture, zero-initializing $W$ is not only safe but often cleaner than random initialization. In an ordinary network, zero initialization is bad because it fails to break symmetry, but here symmetry is already broken by the sampled noise $\varepsilon$ in $W_{\text{eff}}$. Since training begins at $\alpha = 0$, the learned weights do not contribute to the forward pass anyway, so random weight initialization only injects arbitrary structure into a phase that is meant to be pure exploration. Starting from $W = 0$ makes that exploratory regime honest and lets useful structure enter only when the bias toward learned weights increases.

---

## 9. Layer Architecture

All ergodic layers follow the effective-weight pattern. When `ergodic=True`, noise buffers are registered and weights are zero-initialized (noise provides initial variation). When `ergodic=False`, weights are randomly initialized and no noise buffers exist.

### ErgodicLayer Base Class

`ErgodicLayer` (`Model.py`) provides per-neuron `bias` and `var` buffers driven by gradient variance tracking (`observe_sigma()` / `sigma_to_ergodic()`). Subclasses that contain child ErgodicLayers must forward `set_sigma()` and `paramUpdate()` to their children.

### InvertibleLinearLayer (LDU factorisation)

$$W = L \cdot D_{\text{embed}} \cdot U$$

Factored into three components (not separate sub-layers):

- **L**: Unit lower-triangular matrix (diagonal fixed at 1). Stored as `raw_L`; the strict lower triangle is extracted at each forward call.
- **D**: Diagonal vector of length `rank = min(nIn, nOut)`, embedded into a rectangular `[nIn, nOut]` matrix by zero-padding.
- **U**: Unit upper-triangular matrix (diagonal fixed at 1). Stored as `raw_U`; the strict upper triangle is extracted symmetrically.

Exact inverse via triangular solves: $W^{-1} = U^{-1} \cdot D^{-1} \cdot L^{-1}$. No SVD is required.

When `naive=False` (default), L, D, U are applied sequentially to $x$ — no full $W$ materialisation. When `naive=True`, $W_{\text{eff}}$ is materialised as a dense matrix and the reverse path materialises the dense LDU inverse.

When `ergodic=True`, noise is injected at the factor level (not the matrix level) to preserve exact invertibility at all temperatures:

$$L_{\text{eff}} = I + \text{strict_lower}(\text{raw_L} + t \cdot \text{noise_L})$$
$$U_{\text{eff}} = I + \text{strict_upper}(\text{raw_U} + t \cdot \text{noise_U})$$
$$d_{\text{eff}} = b \cdot d_{\text{clamped}} + t \cdot \text{noise_d}$$

Because $W_{\text{eff}}$ remains in LDU form, its exact inverse is always available via the same triangular solves — no approximation, regardless of noise level. See Architecture.md for the full mathematical treatment.

---

## 10. Training Loop Integration

```
# 1. Standard Adam optimizer for W (excludes temperature)
optimizer = torch.optim.Adam(model.getParameters(), lr=lr)

# 2. Forward pass computes W_eff = bias*W + var*noise
loss = model(x, y)

# 3. Backward pass: Adam gets dL/dW, temperature gets dL/dT
loss.backward()

# 4. Adam updates W
optimizer.step()

# 5. The sensor updates alpha via paramUpdate() on each ErgodicLayer
#    set_sigma() propagates to child sub-layers
model.paramUpdate()
```

---

## 11. Summary of the Full Algorithm

**Initialize:**

$$\alpha_0 = 0, \quad v_0 = 0, \quad t = 0, \quad \beta = 0.999$$

**Each training step:**

1. Sample $\varepsilon \sim \mathcal{N}(0, I)$
2. Compute $W_{\text{eff}} = \alpha_t W + (1 - \alpha_t)\varepsilon$
3. Forward pass: $\hat{y} = f(x; W_{\text{eff}})$
4. Compute loss $\mathcal{L}(\hat{y}, y)$
5. Backward pass: compute $\nabla_W \mathcal{L}$ and $g_t = \nabla_T \mathcal{L}$
6. **Adam** updates $W$: $W \leftarrow W - \eta \cdot \hat{m}_t / \sqrt{\hat{v}_t^{(W)}}$
7. **Sensor** updates $\alpha$ via per-neuron sigma tracking:

$$v_t \leftarrow \beta \cdot v_{t-1} + (1 - \beta) \cdot g_t^2$$

$$\hat{v}_t = \frac{v_t}{1 - \beta^t}$$

$$\text{var}_j = \frac{\hat{v}_j}{\hat{v}_j + \kappa}, \qquad \text{bias}_j = 1 - \text{var}_j$$

8. Zero temperature gradient: $g_t \leftarrow 0$

---

## Factor-Level Noise Injection

### Previous Approach: Matrix-Level Blending

The original ergodic formulation applied noise at the level of the full weight matrix:

$$W_{\text{eff}} = b \cdot W + t \cdot N, \qquad N \sim \mathcal{N}(0, I)$$

This worked well for non-invertible layers, but breaks down when exact invertibility is required. The approximate inverse

$$W_{\text{eff}}^{-1} \approx b \cdot W^{-1} + t \cdot N^{-1}$$

has reconstruction error proportional to the cross-terms:

$$W_{\text{eff}} \cdot W_{\text{eff}}^{-1} \approx (b^2 + t^2)I + bt(WN^{-1} + NW^{-1}) \neq I$$

The error is small when `t` is small, but grows with temperature – exactly the regime where exploration is most active.

### New Approach: Factor-Level Injection

For `InvertibleLinearLayer`, noise is injected directly into the raw parameters of each LDU factor before the triangular structure is extracted:

```
L_eff = I + strict_lower(raw_L + t * noise_raw_L)
U_eff = I + strict_upper(raw_U + t * noise_raw_U)
d_eff = b * d_effective + t * noise_d
W_eff = L_eff @ D_eff_embed @ U_eff
```

Because `W_eff` is in LDU form regardless of the noise level, its exact inverse is always available via the same triangular solves:

```
W_eff^{-1} = U_eff^{-1} @ D_eff^{-1} @ L_eff^{-1}
```

No approximation is introduced at any temperature.

### stable=True and the d Backbone

The `stable=True` flag controls whether `_d_effective()` clamps the deterministic diagonal backbone before the ergodic blend. This is the only place the stability constraint lives:

```
d_clamped_i = sign(d_i) * clamp(|d_i|, eps, 1)
d_eff = b * d_clamped + t * noise_d
```

The noise factor `noise_d` is sampled with magnitude in $[\text{eps}, 1]$ and random sign, so it is itself always invertible. Combined, `d_eff` stays bounded away from zero at all temperatures without any additional clamp on the blended result.

### SigmaLayer and PiLayer Integration

`SigmaLayer(invertible=True)` and `PiLayer(invertible=True)` use `InvertibleLinearLayer(ergodic=True)` internally for the linear component of their transformations. The `bias` and `var` values computed by the parent `ErgodicLayer` are forwarded as `bias=` and `temp=` arguments to `InvertibleLinearLayer.forward()` and `InvertibleLinearLayer.reverse()`, so the same gradient-energy sensor that controls the outer layer also governs the factor-level noise injection in the inner linear primitive.

Alec Rogers

September 24, 2025

## Abstract

*Problem*

*Society has become highly dependent on machine minds, but we do not have a good theory of those minds. Although human and machine minds have considerable functional overlap, their design goals are considerably*

*different, and treating a machine mind as a human mind can have significant negative consequences. Therefore, we should have a different "theory of mind" when we interact with machine minds, and our theory should be informed by how those minds are designed and trained.*

*Background*

*To develop this theory of mind, it is helpful to anthropomorphize machines because we naturally understand them in relation to human minds. Therefore, this essay focusses on three main questions: what are machine minds, what do machine minds feel, and what do machine minds know.*

*While these changes do not allow us to adopt a human theory of mind, they do permit of a theory of mind that is significantly less broken and inhumane.*

*Solution*

*As a result of this analysis, we also propose three specific measures to make a machine mind more humanistic or natural: weight ergodicity, network invertibility, and output certainty. Weight ergodicity refers to the fact that weights are samples from a random (ergodic) process, rather than fixed constants. Network invertibility refers to the fact that the network architecture is bidirectional and partially invertible. Output certainty refers to the fact that a network should know that it does not know, as opposed to merely knowing the probability of one answer as opposed to another. To illustrate these principles, the performance of several models implementing these paradigms is compared to a traditional model. All models are configured with a 20-neuron, single hidden layer and tested on the MNIST dataset.*

*Network invertibility is important with respect to developing a meaningful semantic embedding: the symbols of an LLM should correspond to something and we should be able to know to what they correspond, as opposed to being merely abstract tokens inside of Searl's Chinese translation engine.*

## Background

The scope of this paper is Large Language Models, or reinforcement learning where the inputs and outputs are known and the weights of the network are unknown (i.e. the transformer architecture).

---

## Part I: What Machine Minds Are – Weight Ergodicity

### Weights as Ergodic Samples

Measurement theory, when applied to a neural network, views the weights not as fixed constants to be optimized but as **samples from an ergodic distribution**. The weight matrix W at any moment is one realization of a stochastic process whose statistics reflect the boundary conditions of the learning problem:

$$W = \mu M + \sigma V$$

where $\mu$ and $\sigma$ are the bias and variance parameters that determine the contribution of the random process. The bias and variance parameterize a noisy signal where $M$ has a norm of 1 and variance of zero and an unbiased variance $V$ with unit variance. Note that the matrix $M$ is analogous to the weight matrix in a typical neural network while $V$ is sampled from a standard normal distribution.

This perspective comes from **Lagrangian Field-theoretic Gradient Control** (LFGC), which treats the weight landscape as a field where the boundary conditions – input data, output targets, and architectural constraints – shape the distribution from which weights are drawn. Rather than searching for a single optimal point, the network explores the manifold of solutions consistent with its boundary conditions. The ergodic hypothesis guarantees that time averages (over training steps) equal ensemble averages (over weight configurations), so the training trajectory visits representative regions of the solution manifold.

See Rogers, A. (2026), *Local Freedom under Global Constraint*, Zenodo, doi: 10.5281/zenodo.18644302.

### Temperature and Annealing

For simplicity, the mean and variance of all weights are characterized here by a single *temperature* parameter for each layer that is updated via the ADAM algorithm. While it may be desirable to have a temperature associated with each input weight, this is not necessary to compare to a normal network in order to measure performance.

Another simplification in this context is to use the temperature parameter to determine both the bias and the variance. Given the temperature *temp*, the $\mu$ and $\sigma$ parameters are calculated as follows:

$$\mu = 1 - temp$$

$$\sigma = temp$$

In this experiment, the temperature of the ergodic weight process begins at 1 and decreases to 0 using the ADAM algorithm.

### Advantages of Ergodic Weights

There are three significant advantages to using ergodic weights:

1. **No learning rate required.** Biases find different locations in weight space as a result of the randomness and converge as the temperature decreases. The adaptive learning rate schedule is reinterpreted as a process of simulated annealing.
2. **Zero initialization.** Weights can be initialized to zero, since they find different locations in weight space over time. The annealing process helps the model bounce out of local minima and saddle points more effectively than stepping opposite to the gradient, which is prone to deterministic and oscillatory behavior.
3. **Theoretical convergence guarantee.** The literature on simulated annealing provides a theoretical guarantee of convergence to the global optimum if run for an infinite number of iterations.

Note that the standard *dropout* algorithm may be interpreted as a per-neuron variance in the bias parameter. Thus, while dropout explicitly sets neurons to zero with some frequency, the same type of regularization will occur naturally as a result of the difference over time of the weight biases.

---

## Part II: What Machine Minds Feel – Network Invertibility

### Bidirectional Perception

The egocentric point of view tends to see perception as a passive process, while physics suggests that it is bidirectional. That bidirectionality within a neural network can be approximated via the invertibility of its weight multiplications and activation functions.

### Symbols, Worldlines, and Karmic Inertia

The symbols or tokens that an LLM processes are not arbitrary labels – they have meanings given to them by **worldlines**. A worldline is the trajectory of a concept through training: the accumulated history of contexts in which a token has appeared, the gradients that have shaped its embedding, and the network states it has participated in. The meaning of a symbol is the integral of its worldline.

Weight adaptation – or even inference – puts a **force** into the system. That force will adapt either the system's inputs or its outputs, depending on which has less **karmic inertia**. Karmic inertia is the resistance of a representation to change: a well-established embedding with a long worldline (many training examples,

consistent gradients) has high karmic inertia, while a novel or ambiguous token has low karmic inertia. The network's bidirectional architecture means this force propagates in both directions:

- **Forward (encoding):** the input representation adapts to match the network's learned categories – perception is shaped by expectation.
- **Reverse (decoding):** the network's internal state adapts to reconstruct the input – expectation is grounded in perception.

The invertible architecture makes both directions explicit and trainable, rather than relying on separate encoder/decoder networks with independent weights.

### Implementation

Even when the network is not invertible, bidirectionality provides an important role in backpropagation and network topologies such as autoencoders that perform a decoding step that is analogous to the reverse of the encoding step. In modern practice, the topological similarity of encoders and decoders does not typically extend to shared weights. Here, we briefly examine the possibility of simultaneously training the network in both the forward and reverse direction: the forward encoding step in which the outputs are predicted from the input, and the reverse decoding step in which the inputs are predicted from the (possibly predicted) outputs. For a related experiment in sharing the weight matrix between controller (policy) and critic (value) networks, see *Rogers et al 2001*.

In this experiment, the output is a one-hot encoded vector corresponding to the predicted MNIST digit, a classification task that is clearly not a bijective (invertible) function. To address this situation, the inputs are predicted from the outputs of the *penultimate* network layer; thus, the network is composed of a bijective front and a surjective back. We characterize these components respectively as the *recognizer* (or representer) and the *generalizer*.

---

## Part III: What Machine Minds Know – Output Certainty

### Certainty vs. Probability

The error function of a neural network corresponds to its desire. Modern machine minds use cross-entropy loss, so their output distribution *must* classify the output. This does not allow the network to express the certainty that the input belongs to a given class or not; rather, it expresses the probability that the input belongs to one class *as opposed to another*.

To remedy this situation, a variation of cross-entropy loss is used which blends standard cross-entropy loss with a product of cross-entropy loss and the norm of the prediction. An output value of zero is penalized by cross-entropy loss, but any incorrect prediction carries an additional penalty in proportion to its magnitude to penalize overconfidence.

### Certainty-Weighted Loss

The formula for the certainty-weighted cross-entropy version of the loss function is:

$$error = -\sum_{c=1}^{N} t_c \log(p_c)$$

$$certainty = |\hat{y}|$$

$$surprise = (\alpha) \cdot certainty \cdot error + (1 - \alpha) \cdot error$$

cosine similarity

Amplitude certainty

$$error = (\hat{y} - y)^2$$

**Knowing What You Do Not Know**

The key insight is that a network should **know that it does not know**, as opposed to merely knowing the probability of one answer as opposed to another. Cross-entropy forces the network to distribute all of its probability mass across the known classes, even when the correct response is "I don't know." Certainty weighting allows the network to express low confidence (small output norm) independently of its class distribution, separating the questions "which class?" and "how sure?"

---

## Methods

The software used to run these experiments is a combination of Python and PyTorch (the Python code is provided in the appendices).

The data is the MNIST dataset, chosen because it is simple and well-known. It consists of 60,000 training images and 10,000 test images, each 28x28 pixels in size, with a monochromatic bit depth of 256. Their target labels are provided as output. The data is preprocessed by removing its mean and variance, and is shuffled between trials.

For all paradigms, the input is a 28x28 element array and the output is a one-hot encoded representation of the target class. The architecture for all paradigms is a 20 hidden unit neural network. The batch size is 10, and each network is run for 9 epochs. To generate confidence intervals for model accuracy, each paradigm is run for 7 trials to generate error bars reflecting the standard deviation of the accuracy values.

The standard architecture that serves as a benchmark for the other paradigms uses a RelU nonlinearity, and a learning rate of 0.01 that is updated via the ADAM method. Cross-entropy loss is used for the error function.

The base ergodic model uses weights initialized to zero and a single per-layer temperature parameter that governs the bias/variance trade-off of the ergodic weights. The temperature is updated with the ADAM method. Dropout is used for the middle layer, which means that both the bias and variance of the ergodic weights are set to zero with a frequency of 0.75.

Five models were evaluated: the base model and four models using the ergodic weight updating scheme. Of the models with ergodic weights, one model has simply ergodic weights, one has input normalization, one reversible model uses separate forward and backward layers, and one reversible model uses a single invertible weight matrix. For the invertible model, the inversion of the weight matrix is typically computationally expensive. Therefore, the weight matrix is represented in SVD form, as a rotation matrix followed by an eigenvalue matrix followed by another rotation matrix. In virtue of this sparsely-parameterized decomposition, the rotation matrices can be computed by Givens rotations, and the eigenvalue matrix stores only diagonal elements. Therefore, direct matrix inversion and its computational cost are avoided. That said, doing so requires a more computationally expensive forward pass and a somewhat circuitous back propagation step.

The weight update equations for the ergodic model do not use a learning rate. So compared to traditional gradient descent equations that look as follows for a given learning rate parameter $\alpha$:

$$err = (y - \hat{y})^2$$

$$\triangle_W = \alpha(y - \hat{y})\nabla_F x$$

$$err = (y - \hat{y})^2$$

$$e = y - \hat{y}$$

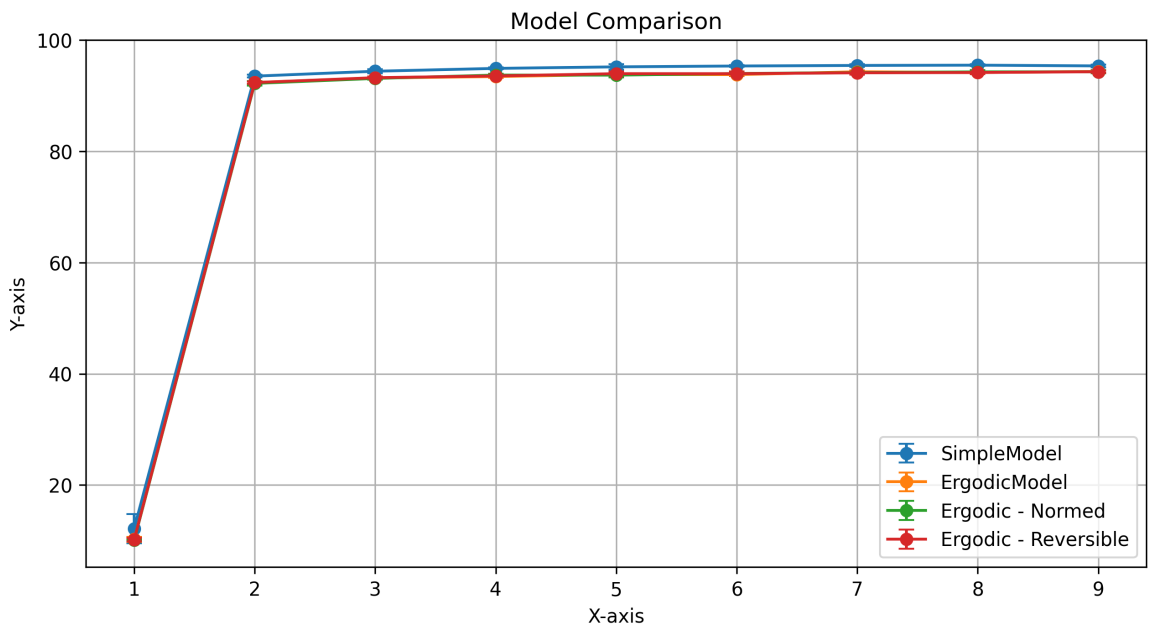
$$c = \alpha\hat{y}^2 + (1 - \alpha)$$

$$\triangle_W = (ce - \alpha\hat{y}e^2)x$$

This equation demonstrates that the network is simultaneously trying to minimize error and "incorrect certainty".

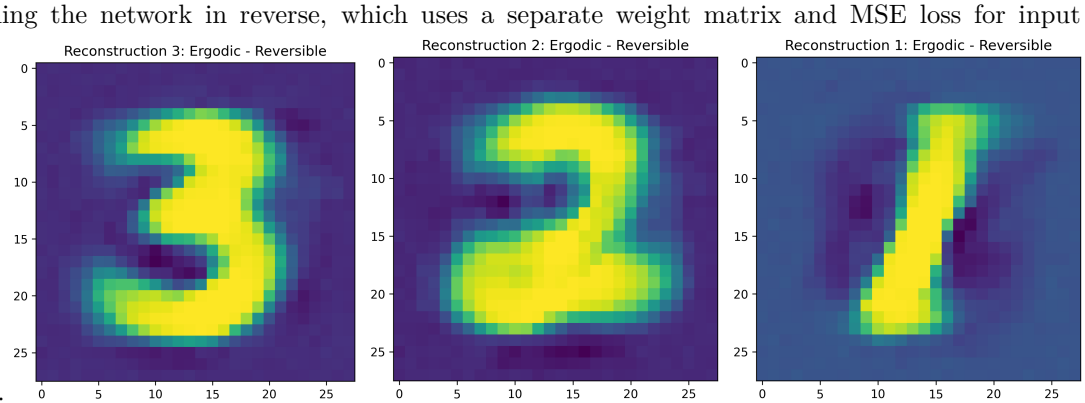
## Results

The most important result of this experiment is the model comparison with error bars, that demonstrates that the simple model performs slightly better than all proposed paradigms, and that all of the proposed paradigms




perform a lmost identically:

Finally, although there is no invertible network to compare it to, we show the reconstruction of the input by running the network in reverse, which uses a separate weight matrix and MSE loss for input




reconstruction:

## Discussion

The benefits of the novel paradigms described in this paper are as follows:

Random weight initialization is unnecessary, which makes performance less subject to arbitrary initialization. This is visually demonstrated by the error bars in the Model Comparison figure, which show that only the simple model (with random weight variation) has significant variation in performance.

The learning rate is unnecessary, since its role is replaced with weight noise variance, just as the role of dropout is replaced with weight bias variance (which in this experiment is just the regular dropout schedule). Since dropout is folded into the weight adaptation schedule, an integrated algorithm that controls both weight bias and weight variance (and which uses separate parameters for each weight) is suggested for future experiments that follow this paradigm.

The error function encapsulates a measure of certainty in the output in addition to a measure of certainty between the output choices.

However, there were several experiments that my hardware was not fast enough to evaluate and a number of implementation issues that I did not have the time to address:

The Pearson correlation between the certainty and the accuracy for each digit is insignificant, which suggests that the error function did *not* induce the network to be less certain when it was less accurate. This probably points to a problem in the implementation of the error function: instead of using $certainty = |\hat{y}|$ to reduce the penalty for lack of certainty, we used $certainty = p_c$. This leads to prediction norms that are much higher than 1.0 (since softmax creates a unit-sum PDF at the output layer, while using the norm of the output vector to indicate certainty imposes a contradictory constraint). This might explain the lack of correlation between the certainty and accuracy, which theoretically should correlate quite strongly. Performing the experiment with corrected code (as by using a norm-weighted MSE with only one zero) should yield better correlation between certainty and accuracy for each target class.

Finally, **the use of the autograd algorithm within PyTorch led to weights being tuned directly**, which interferes with the simultaneous tuning of the temperature parameter in a way that is not explicitly captured in this paper. So while the addition of weight variance does make random weight initialization unnecessary, the intricacies of the interaction between the weight tuning and the temperature tuning is hidden behind the autograd mechanism.

## Conclusion

Overall, I am optimistic about the strength of the philosophical grounding behind several of the paradigms introduced in this paper. However, due to several implementation issues, I do not feel that the hypotheses were well-tested. Further, even if they had been, the effectiveness of these neural network paradigms in both multilayer neural networks and with more challenging tasks is not clear.

## References


Rogers, Shannon, Lendaris; 2001: *A comparison of DHP based antecedent parameter tuning strategies for fuzzy control*, Proceedings of the Joint 9th IFSA World Congress and 20th NAFIPS International Conference, IEEE

Rogers, A; 2003: *Analysis of the Instantaneous Estimate of Autocorrelation*, unpublished

Rogers, A; 2026: *Local Freedom under Global Constraint*, Zenodo, doi: 10.5281/zenodo.18644302


# XML Configuration Parameters

Model configuration is specified in XML files (e.g. `data/XOR_exact.xml`). Default values are loaded from `data/model.xml` and overridden by model-specific configs. The schema is defined in `data/model.xsd`.

### <architecture>

Core model settings. Training and data parameters are in nested sub-elements <training> and <data> (see below).

| Parameter | Type | Default | Description |
|---|---|---|---|
| modelType | string | "simple" | Model type: "simple" (feedforward), "lm" (language model with sequence proc |
| ergodic | bool | false | Enable ergodic exploration mode. Layers use W_eff = bias * W + temp * noi |
| certainty | bool | false | Enable per-neuron certainty tracking in ergodic layers. Allows individual neuron |
| reshape | bool | false | Reshape input data for sequence processing. Required for modelType=lm. |
| conceptualOrder | int | 1 | Order of conceptual processing. Controls how many higher-order conceptual tra |
| processSymbols | bool | false | Enable additional symbolic processing steps. |
| maskedPrediction | string | "NONE" | Masked prediction mode: NONE, IR, AR, ARUS, ARIR. See Training.md. |
| reconstruct | string | "NONE" | Controls the reverse (reconstruction) pass. "NONE" disables reconstruction entire |
| maxResponseLength | int | 64 | Maximum number of characters/tokens to generate during inference. Measured |

---

#### <architecture><data>

Data loading and filtering settings.

| Parameter | Type | Default | Description |
|---|---|---|---|
| dataset | string | "xor" | Dataset to load. Determines input/output data via TheData.load(). Common va |
| minFrequency | float | 0.0 | Minimum word frequency ratio for vocabulary admission. Words below this thresh |
| shardDir | string | – | Directory containing text shards for streaming datasets (e.g. "data/fineweb"). |
| numShards | int | 1 | Number of shards to load from shardDir. |
| maxDocs | int | 10000 | Maximum number of documents to load per shard. |
| classificationMin | float | – | Minimum threshold for classification accuracy reporting. |
| classificationMax | float | – | Maximum threshold for classification accuracy reporting. |

---

#### <architecture><training>

Training loop and I/O settings.

| Parameter | Type | Default | Description |
|---|---|---|---|
| numTrials | int | 1 | Number of independent training runs. |
| numEpochs | int | 3 | Number of training epochs per trial. |
| batchSize | int | 10 | Number of contiguous streams through the dataset. Ea |
| numWorkers | int | 0 | DataLoader prefetch workers. 0 means synchronous in- |
| learningRate | float | 0.001 | Learning rate for the Adam optimizer. |
| reconRatio | float | 0.5 | Weight of reconstruction loss in combined loss: total |
| trainEmbedding | string | "NONE" | Embedding update mode. Controls both the embeddin |
| trainEmbeddingRatio | float | 0.1 | Weight $\lambda$ of embedding loss in JOINT mode: $\mathcal{L} = \mathcal{L}_{\text{mo}}$ |
| weightsPath | string | "output/BasicModel.ckpt" | File path for saving/loading model weights checkpoint. |
| embeddingPath | string | – | File path for the word vector store (.kv extension, gen |
| autoload | bool | true | Automatically load weights from weightsPath on mod |
| autosave | bool | false | Automatically save weights after training completes. |
| checkpointEveryBatches | int | 0 | Save in-progress weights and embeddings every N optim |
| negSamples | int | 64 | Number of negative samples per positive example for C |

**`<trainEmbedding>` – Embedding Update Modes**

| Value | Embedding method | Model layers | Gradients through codebook | Description |
|---|---|---|---|---|
| `NONE` | Frozen | Trained | No | Embeddings frozen; only model layer |
| `CBOW` | CBOW (padded context) | Trained | No | CBOW reshapes embeddings via ow |
| `SBOW` | SBOW (centroid) | Trained | No | Faster variant: predict each word fro |
| `BACKPROP` | Backprop only | Trained | Yes | Codebook trained purely via model l |
| `BOTH` | SBOW post-batch | Trained | Yes | Two optimizers: SBOW updates emb |
| `JOINT` | Single backward | Trained | Yes | Single optimizer: $\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{model}} + \lambda$ |

**`<trainEmbeddingRatio>` – Embedding Loss Weight (JOINT mode)**

| Parameter | Type | Default | Description |
|---|---|---|---|
| `trainEmbeddingRatio` | float | `0.1` | Weight $\lambda$ of the embedding co-occurrence loss in JOINT mode. Higher values bi |

---

## BasicModel Configuration

The basicModel's `MentalModel.xml` defines the neural architecture. Three parameters control the TruthLayer integration (truth bias during inference and loss modification during training). All are optional; omitting them defaults to 0.1.

| Parameter | Type | Default | Location | Description |
|---|---|---|---|---|
| `truthBiasScale` | decimal | `0.1` | `<architecture>` | Strength of luminosity-based bias on concept input during |
| `LuminosityWeight` | decimal | `0.1` | `<architecture>` | Loss penalty weight for low luminosity (incoherent truths). |
| `UniversalityWeight` | decimal | `0.1` | `<architecture>` | Loss penalty weight for low universality (unkind propositio |
| `TruthLoss` | decimal | `0.0` | `<training>` | Additive loss penalty for propositions that contradict the T |
| `conceptualOrder` | int | `1` | `<architecture>` | Number of Percept→Concept→Symbol iterations. Higher c |
| `useButterflies` | boolean | `false` | `<architecture>` | Enable pairwise sigma/pi mixing via butterfly-mode Pi/Sig |
| `monotonic` | boolean | `false` | `<architecture>` | When true, invertible SigmaLayers / PiLayers use W>=0 ( |
| `useGrammar` | string | `"none"` | `<WordSpace>` | Grammar mode. Current parser accepts `none`, `all`, and leg |

```
<architecture>
  <truthBiasScale>0.1</truthBiasScale>
  <LuminosityWeight>0.1</LuminosityWeight>
  <UniversalityWeight>0.1</UniversalityWeight>
  <conceptualOrder>1</conceptualOrder>
  <useButterflies>false</useButterflies>
  <monotonic>false</monotonic>
</architecture>

<training>
  <TruthLoss>0.0</TruthLoss>
</training>
```

Shamatha Speech should not be confused with serial mode. Serial mode is the rolling cursor / next-token runtime path. Shamatha Speech is a grammar policy: it may compose over all active percepts, but every logical merge must preserve connected `where()` support and continuous `when()` support so the resulting DNF denotes one object.

Luminosity and universality are **multiplicative** – they scale the total loss by `(1 + LuminosityWeight*(1-luminosity) + UniversalityWeight*(1-universality))`. TruthLoss is **additive** – it directly penalizes specific propositions that contradict stored truths, measured by union norm reduction. Both coexist. See Ethics.md Section Universality and Reasoning.md Section TruthLoss for details.

The grammar section of `MentalModel.xml` declares `lift` as an S-tier binary rule:

```
<S>lift(S, S)</S>      <!-- lift is binary; SVO is assembled by
                            the chart-compose path S -> S VO,
                            VO -> V O, not a ternary lift -->
```

Transitive SVO is produced compositionally via the typed chart rules (`S -> S VO` / `VO -> V O`) when `<chartCompose>true</chartCompose>`; see Language.md and Ethics.md Section Universality for the architectural details.

---

### `<InputSpace>`

The entry point that lifts raw data into the model's internal representation.

| Parameter | Type | Default | Description |
|---|---|---|---|
| `nActive` | int | *required* | Sequence length: maximum number of tokens per input. For XOR: 2 (two binary inputs |
| `nDim` | int | `1` | Dimensionality of each input vector. Set on TheObjectEncoding via XML; not passed to |
| `nVectors` | int | = `nActive` | Codebook size (total vectors in the space). Defaults to `nActive` for InputSpace. |
| `nWhere` | int | `0` | Number of spatial/positional dimensions appended to each vector. When > 0, enables P |
| `nWhen` | int | `0` | Number of temporal dimensions appended to each vector. When > 0, enables Temporall |
| `codebook` | bool | `false` | Whether input values use codebook vector quantization. |
| `lexer` | string | `"word"` | Tokenization mode for text inputs. Common values in the current configs are `"word"` an |

**Note:** InputSpace's `inputShape` uses the data's native dimension (e.g. 784 for MNIST, `inputLength` for text), while `outputShape` uses `nDim` from TheObjectEncoding. The LiftingLayer bridges the two.

**Layers:** Contains a `LiftingLayer` that projects raw input into the model's working dimensionality.

---

### `<PerceptualSpace>`

Transforms lifted input into perceptual features via Pi layers (multiplicative interactions).

| Parameter | Type | Default | Description |
|---|---|---|---|
| `nActive` | int | *required* | Number of active perceptual feature vectors (sparse subset selected in forward pass). |
| `nDim` | int | `1` | Dimensionality of each perceptual vector. Set on TheObjectEncoding; not passed to t |
| `nVectors` | int | `0` | Codebook size (total vectors in the space). When > 0, enables vector quantization wi |
| `invertible` | bool | `false` | Use a single invertible Pi layer instead of separate forward/reverse layers. When `true` |
| `hasAttention` | bool | `true` | Enable attention mechanism in this space. |
| `codebook` | bool | `false` | When `true`, the `.what` slot is a learnable `Codebook`; when `false`, it is a passthrough |
| `bivectorOutput` | bool | `false` | When `true`, applies the Q2 promotion $(a_P, a_N) = (\max(0, x), \max(0, -x))$ to the per- |

**Layers:** Pi layers – multiplicative: `y_j = b_j * prod_i(1 + W_ji * x_i)`. See Architecture.md.

**Note:** `InvertiblePiLayer` has been merged into `PiLayer(invertible=True)`; the standalone class is removed.

---

### <ConceptualSpace>

Transforms perceptual features into abstract concepts via Sigma layers (additive/linear).

| Parameter | Type | Default | Description |
|---|---|---|---|
| `nActive` | int | *required* | Number of active concept vectors (sparse subset). For XOR: 3. |
| `nDim` | int | `1` | Dimensionality of each concept vector. Set on TheObjectEncoding; not passed to the Sp |
| `nVectors` | int | `0` | Codebook size (total vectors in the space). |
| `invertible` | bool | `false` | Use single invertible Sigma layer vs. separate forward/reverse layers (`sigma1`/`sigma2`). |
| `hasAttention` | bool | `false` | Enable attention in conceptual processing. |
| `hasNorm` | bool | `false` | Enable layer normalization in this space. |

**Layers:** Sigma layers – additive: `y_j = tanh(W x + b)`. See Architecture.md.

**Note:** `InvertibleSigmaLayer` has been merged into `SigmaLayer(invertible=True)`; the standalone class is removed.

---

### <SymbolicSpace>

Discrete symbolic representation – the information bottleneck between perception and output. Maps concept activations to a one-hot encoding over a codebook of symbol prototypes. See Language.md for the full design.

| Parameter | Type | Default | Description |
|---|---|---|---|
| `nActive` | int | *required* | Number of active symbols. When reconstruction symbols are enabled, `nOutputSym` |
| `nDim` | int | `1` | Dimensionality of each symbol (typically 1 – symbols are scalar activations). |
| `nVectors` | int | = `nActive` | Codebook size. When `codebook=true`, equals the number of symbol prototypes. |
| `codebook` | bool | `false` | Enable codebook quantization. When `true`, the forward path produces a one-hot a |
| `bivectorOutput` | bool | `false` | When `true`, forward returns the per-prototype catuskoti bivector $[B, V_S, 2]$ via `Cod` |

**Layers:** SymbolicSpace owns one `SigmaLayer(nConcepts, nSymbols, invertible=True, monotonic=monotonic)` at `self.sigma` that bridges the C↔S boundary in both directions via its own self-inverse (`forwardSigma` / `reverseSigma` pointer aliases hide the one-or-two-layer split). The legacy `SymbolicSpace.layer` PiLayer attribute is gone; consumers use `model.symbolicSpace.sigma` directly. Codebook provides dense vectors when codebook=true.

---

### <SyntacticSpace>

Generates binary derivation trees (deep structure) from active symbols using a Chomsky Normal Form grammar. See Language.md for the grammar and open implementation questions.

| Parameter | Type | Default | Description |
|---|---|---|---|
| `nActive` | int | `16` | Number of active word vectors. |
| `nDim` | int | `1` | Dimensionality of each word vector. |
| `nVectors` | int | = `nActive` | Codebook size (total vectors in the space). |

**Layers:** No trainable parameters currently. Rule selection is random; derivation is stored as word tuples `(batch, vector, rule)` on the output subspace.

---

### `<OutputSpace>`

Maps symbols to final predictions via linear layers.

| Parameter | Type | Default | Description |
|---|---|---|---|
| `nActive` | int | *required* | Number of active output values. For XOR: 1 (single binary output). |
| `nDim` | int | `1` | Dimensionality of each output. Set on TheObjectEncoding; not passed to the Space cons |
| `nVectors` | int | = `nActive` | Codebook size (total vectors in the space). Defaults to `nActive` for OutputSpace. |

**Layers:** Linear layers with `(bias, temp)` support for ergodic mode.

---

### `InvertibleLinearLayer`

The LDU-factorised invertible linear primitive used internally by `SigmaLayer(invertible=True)` and `PiLayer(invertible=True)`. These parameters are not set via XML directly; they are configured programmatically when constructing the layer.

| Parameter | Type | Default | Description |
|---|---|---|---|
| `naive` | bool | `False` | `False`: apply L, D, U sequentially via triangular solves – no W materialisation, backprop thr |
| `stable` | bool | `False` | Clamps each diagonal entry `d_i` to magnitude `[eps, 1]` with sign preserved in `_d_effectiv` |
| `ergodic` | bool | `False` | Enables factor-level noise injection. When `True`, registers noise buffers `noise_raw_L`, `noise_` |

**Class renames and removals.**

| Old name | New name / status |
|---|---|
| `LULayer` | Renamed to `InvertibleLinearLayer` |
| `InvertibleLinearLayer` (SVD-based) | Removed; replaced by LDU factorisation |
| `InvertibleSigmaLayer` | Merged into `SigmaLayer(invertible=True)` |
| `InvertiblePiLayer` | Merged into `PiLayer(invertible=True)` |

See Architecture.md for the full mathematical treatment of the LDU factorisation and factor-level noise injection.

---

### `<architecture><server>`

HTTP server settings (used by `serve.py`).

| Parameter | Type | Default | Description |
|---|---|---|---|
| `host` | string | `"127.0.0.1"` | Server bind address. |
| `port` | int | `8001` | Server port. |
| `timeout` | int | `120` | Request timeout in seconds. |

---

### Example: XOR Configuration

```xml
<model>
  <architecture>
    <reconstruct>symbols</reconstruct>
    <reshape>true</reshape>
    <modelType>lm</modelType>
    <ergodic>false</ergodic>

    <data>
      <dataset>xor</dataset>
    </data>

    <training>
      <numEpochs>1000</numEpochs>
      <learningRate>0.01</learningRate>
      <autoload>false</autoload>
      <autosave>false</autosave>
    </training>
  </architecture>

  <InputSpace>
    <nActive>2</nActive>
    <nDim>1</nDim>
  </InputSpace>

  <PerceptualSpace>
    <nActive>4</nActive>
    <nDim>1</nDim>
    <nVectors>16</nVectors>
    <invertible>false</invertible>
    <hasAttention>false</hasAttention>
  </PerceptualSpace>

  <ConceptualSpace>
    <nActive>3</nActive>
    <nDim>1</nDim>
    <nVectors>16</nVectors>
    <invertible>false</invertible>
  </ConceptualSpace>

  <SymbolicSpace>
    <nActive>3</nActive>
    <codebook>false</codebook>
  </SymbolicSpace>

  <OutputSpace>
    <nActive>1</nActive>
    <nDim>1</nDim>
  </OutputSpace>
</model>
```

In this configuration:

- 2 binary inputs (`nActive=2`) are lifted, transformed through 4 perceptual and 3 conceptual active

features
- PerceptualSpace and ConceptualSpace each have a codebook of 16 vectors (`nVectors=16`), selecting `nActive` via topk
- 3 symbols are produced: 1 for output prediction, 2 for reconstruction
- The reverse pass reconstructs the original 2 inputs from all 3 symbols
- Ergodic mode is off; training uses standard Adam with combined forward+reverse loss

---

## Example: Embedding / Chatbot Configuration

```
<model>
  <architecture>
    <reconstruct>symbols</reconstruct>
    <modelType>embedding</modelType>
    <maskedPrediction>AR</maskedPrediction>

    <data>
      <dataset>text</dataset>
      <shardDir>data/fineweb</shardDir>
      <numShards>1</numShards>
      <maxDocs>10000</maxDocs>
      <minFrequency>0.00001</minFrequency>
    </data>

    <training>
      <numEpochs>1</numEpochs>
      <batchSize>1</batchSize>
      <learningRate>0.001</learningRate>
      <trainEmbedding>BACKPROP</trainEmbedding>
      <weightsPath>BasicModel.ckpt</weightsPath>
      <embeddingPath>BasicModel.kv</embeddingPath>
      <autoload>true</autoload>
      <autosave>true</autosave>
      <negSamples>64</negSamples>
    </training>
  </architecture>

  <InputSpace>
    <nActive>128</nActive>
    <nDim>100</nDim>
    <nWhere>2</nWhere>
    <nWhen>2</nWhen>
    <lexer>sentence</lexer>
    <codebook>true</codebook>
  </InputSpace>

  <!-- ... space definitions ... -->
</model>
```

Key points:

- `modelType=embedding` activates the embedding-based input pipeline alongside the neural model
- `trainEmbedding=BACKPROP` trains model layers with gradients flowing through the codebook; no separate SBOW/CBOW step

- The three files partition model behaviour: **XML config** (architecture), **`.kv` embedding** (codebook), **`.ckpt` weights** (model layers)
- `weightsPath` stores the neural model checkpoint; `embeddingPath` stores the word vectors
- `minFrequency` gates vocabulary admission: words are buffered until their frequency ratio exceeds this threshold