Basic Model

Contents

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

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

Architecture **11**
- Overview . . . 12
  - Spaces . . . 12
  - Reconstruction Symbols . . . 13
  - Single Optimizer with Overlapping Weight Spaces . . . 14
  - Training Loop . . . 14
  - Three-File Architecture . . . 15
- Language System . . . 15
- Sigma and Pi Layers . . . 15
- Dimensionality Constraints . . . 16
- Pi Layer Derivation (Historical) . . . 16
- Invertible Linear Layer (LDU) . . . 17
  - InvertibleLinearLayer – LDU Factorisation . . . 17
  - Ergodic Noise Injection (Factor-Level) . . . 17

stable=True . . . 17
noise_d . . . 17
naive Flag . . . 17
Noise Lifecycle in Ergodic Mode . . . 18
Class Renames . . . 18
Ergodic Exploration . . . 18

Basic Model **18**

Spaces **26**
Overview . . . 26
Base Class: Space . . . 26
Normalization and Ranges . . . 27
InputSpace . . . 27
PerceptualSpace . . . 28
ConceptualSpace . . . 29
SymbolicSpace . . . 30
SyntacticSpace . . . 32
OutputSpace . . . 32

Language **33**
Overview . . . 33
Grammar . . . 33
Accessors . . . 34
Grammar Configuration . . . 34
`data/grammar.cfg` dispatch (Step 6) . . . 36
Thought-Free Mode (Shamatha Speech) . . . 37
Ops Reference . . . 37
Layer-anchor map . . . 37
Target ↔ current ↔ status table . . . 37
Notation . . . 38
Ternary operators (true / false / non) — Sym . . . 38
Negation (not) — Sym . . . 38
Conjunction / disjunction — Sym . . . 38
Intersection / union — SigmaPi (currently aliases `lower/lift`) . . . 39
Part — Def (currently a tensor score) . . . 39
Equal — Def (currently a cosine-gated projection) . . . 39
Slot selectors (what / where / when) — Sym . . . 39
Query — Mereological (currently accumulator preservation) . . . 40
Swap (Sinkhorn soft permutation) — Sym . . . 40
Lift / lower — SigmaPi (mode-dispatch ops) . . . 40
Chunk — Per (invertible max-union; pre-pass) . . . 41
Reverse / inverse ops . . . 41
Mereological suite — see Mereology.md . . . 41
Pointwise primitives . . . 42
Rule Prediction (`forward`) . . . 42
Differentiable rule selection . . . 43
Projection: `project(grammar, rule_id, left, right)` . . . 43
Composition: `compose(data, subspace, grammar, target_count=None)` . . . 43
Phase 1: deterministic `not` . . . 43
Phase 2: soft-weighted cascading composition . . . 44
Leaf ledger (codebook indices) . . . 44
2D activation path (`_compose_activation`) . . . 45
Pairwise reduction: `_compose_to_target(data, subspace, grammar, target_count)` . . . 45

Chart-like Compose (typed SVO) . . . 45
Pair-scorer + per-step argmax . . . 45
Category tensor + POS seeding . . . 46
Derivation trace . . . 46
Grammar-derived SVO . . . 46
Downward Head Emission (`S -> C`) . . . 46
`SyntacticLayer.emit_head(state, codebook) -> (best_idx, contained, residual)` . 46
`WordSpace.reconstruct(state, codebook_space)` . . . 47
Stable hooks already wired . . . 47
Decomposition: `decompose(data, subspace, grammar)` . . . 47
Two-Layer Logic . . . 47
Subsymbolic Layer (Vectors) . . . 47
Symbolization . . . 48
Symbolic Layer (Scalars in [-1,1]) . . . 48
Word Encoding . . . 48
WordSpace and SymbolicSpace wiring . . . 48
SymbolicSpace . . . 48
Forward Path . . . 48
Reverse Path . . . 49
Key Properties . . . 49
AssociationLayer (EQUALS Implementation) . . . 49
How Syntax Is Trained . . . 49
The Intended Differentiable Mechanism . . . 49
The Actual Losses in `runBatch()` . . . 49
Current Integration Status . . . 50
Future: RuleNode on the ReconstructionStack . . . 50
Design History: Learned SVO Plan . . . 50
Learned SVO Role Identification from the Grammar . . . 50
Current state (at the time of writing) . . . 50
Target grammar shape . . . 51
Phase A - Grammar metadata (low risk, ~60 LoC) . . . 51
Phase B - Compose restructure (high risk, ~120 LoC) . . . 52
Phase C - Category tensor alongside activation (~60 LoC) . . . 52
Phase D - SVO extraction hook (~30 LoC) . . . 53
Phase E - Training dynamics . . . 54
Phase F - Test suite . . . 54
Cost estimate . . . 54
Interim: positional tap (the pre-chart stopgap) . . . 55
Appendix - hooks that Phase D builds on (already wired) . . . 55

Mereology **55**
Parthood as Clipped Cosine Projection . . . 55
The Full Suite . . . 56
Region Partition . . . 56
Mereological Fusion . . . 56
ImpenetrableLayer: Five-Relations Regularizer . . . 57
Penalty . . . 57
Trust . . . 57
Diagnostics . . . 57
Configuration . . . 57
Why This Design . . . 58

Logic **58**
Overview . . . 58

1. Subsymbolic Layer . . . 58
Operators . . . 58
2. Symbolization . . . 59
3. Symbolic Layer (Scalars in [-1,1]) . . . 59
Negation (affirming) . . . 59
Non (non-affirming) . . . 59
Union (monotonic) . . . 59
Intersection (monotonic) . . . 59
Parthood as Projection . . . 60
4. Key Insight . . . 60
5. Rationality . . . 61
Truth Statements . . . 61
TruthLayer (WordSpace.truth_layer) . . . 61
Truth Field . . . 61
Consonance and Dissonance . . . 62
Propositional Structure . . . 62
Truth Accumulation Pipeline . . . 62
6. Consistency and Verification . . . 62
Internal Consistency (within a TruthSet) . . . 62
Verification (incoming statement against stored truth) . . . 63
Logical Entailment (augmenting geometry with symbolic closure) . . . 63
7. . . . 64

Radial Operators for Hypersphere Ternary Logic **64**
Overview . . . 64
Truth Values . . . 64
Magnitude Ordering . . . 64
Operators . . . 65
RadMin (Radial Minimum – Conjunction / AND) . . . 65
RadMax (Radial Maximum – Disjunction / OR) . . . 65
NOT (Sign-Flipping Negation) . . . 66
NON (Non-Affirming Negation) . . . 66
Luminosity . . . 67
Fusion . . . 67
Supporting measures . . . 68
Semantic Summary . . . 68
Open Questions . . . 68
8. Cross-references . . . 69

Reasoning System **69**
Partitioned Symbolic Space . . . 69
Reasoning Methods . . . 69
`isConsistent() -> dict` . . . 69
`ground(activation, threshold=0.6) -> dict` . . . 69
`isTrue(activation) -> float` . . . 70
`extrapolate(grammar, seed_indices, max_new, attenuation) -> dict` . . . 70
TruthLoss . . . 70
Bidirectional Reasoning Loop . . . 70
Grammar Learning . . . 70
Architecture Quadrants: useButterflies × useGrammar . . . 71
Flat (both false) . . . 71
Butterfly (useButterflies=true, useGrammar=false) . . . 71
Grammar-directed (useButterflies=false, useGrammar=true) . . . 72
Configuration . . . 72

Contemplative Awareness Methods . . . 72
Testing . . . 73

Training **73**
Overview . . . 73
Phase 1: Embedding Pretraining (`make train` / `embed.py train`) . . . 73
Pipeline . . . 73
SBOW (Sentence Bag of Words) . . . 73
CBOW (Continuous Bag of Words) . . . 74
Two-Pass Architecture . . . 74
Configuration (Makefile variables) . . . 74
Memory Considerations . . . 75
Phase 2: Network Training (`BasicModel.py`) . . . 75
Masked Prediction Modes . . . 75
`<trainEmbedding>`: Embedding Update Mode . . . 75
Gradient Flow Through Codebook . . . 76
`JOINT`: Single Combined Loss . . . 76
Why MSE over Embeddings (not Cross-Entropy over Vocabulary) . . . 77
Training Loop . . . 77
SBOW vs CBOW . . . 78
Three-File Architecture . . . 78
Embedding Space Geometry . . . 78
Profiling . . . 79

Ergodic Exploration via Adaptive Bias-Variance Control **79**
1. The Ergodic Weights . . . 80
2. Why Standard Adam Fails on α . . . 80
3. The Gradient Energy Sensor . . . 81
3.1 Modified Second-Moment Estimator . . . 81
3.2 Bias Correction . . . 81
3.3 What $\hat{v}_t$ Measures . . . 81
3.4 The Alpha Update Rule . . . 81
3.5 Layer-Local Certainty (Per-Neuron Sigma) . . . 82
4. Comparison: Adam vs. Gradient Energy Sensor . . . 82
5. Derivation: Why Zero-Mean Implies Drop m_t . . . 82
6. Dropout via Bias . . . 83
7. The `global_temp` Sensitivity Knob . . . 83
8. Initialization and Bootstrap . . . 83
9. Layer Architecture . . . 84
ErgodicLayer Base Class . . . 84
InvertibleLinearLayer (LDU factorisation) . . . 84
10. Training Loop Integration . . . 85
11. Summary of the Full Algorithm . . . 85
Factor-Level Noise Injection . . . 85
Previous Approach: Matrix-Level Blending . . . 85
New Approach: Factor-Level Injection . . . 86
stable=True and the d Backbone . . . 86
SigmaLayer and PiLayer Integration . . . 86
Abstract . . . 86
Background . . . 87
Part I: What Machine Minds Are – Weight Ergodicity . . . 87
Weights as Ergodic Samples . . . 87
Temperature and Annealing . . . 88
Advantages of Ergodic Weights . . . 88

Part II: What Machine Minds Feel – Network Invertibility . . . 88
Bidirectional Perception . . . 88
Symbols, Worldlines, and Karmic Inertia . . . 88
Implementation . . . 89
Part III: What Machine Minds Know – Output Certainty . . . 89
Certainty vs. Probability . . . 89
Certainty-Weighted Loss . . . 89
Knowing What You Do Not Know . . . 90
Methods . . . 90
Results . . . 91
Discussion . . . 92
Conclusion . . . 92
References . . . 92

XML Configuration Parameters **92**
`<architecture>` . . . 93
`<architecture><data>` . . . 93
`<architecture><training>` . . . 93
MentalModel Configuration . . . 94
`<InputSpace>` . . . 95
`<PerceptualSpace>` . . . 95
`<ConceptualSpace>` . . . 95
`<SymbolicSpace>` . . . 96
`<SyntacticSpace>` . . . 96
`<OutputSpace>` . . . 97
`InvertibleLinearLayer` . . . 97
`<architecture><server>` . . . 97
Example: XOR Configuration . . . 98
Example: Embedding / Chatbot Configuration . . . 99

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

Document	Description
Installation	Setup, Makefile targets, environment variables

Overview

BasicModel is a parameterized neural architecture with three independent levers:

- **Ergodic** – adaptive bias-variance control via a gradient energy sensor (not gradient descent)
- **Certainty** – per-neuron certainty tracking; neurons graduate from exploration to exploitation independently
- **Reversible** – bidirectional training: forward prediction + backward reconstruction in a single optimizer pass

Model configurations are specified in XML and can be compared side-by-side. 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:

```xml
<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

```bash
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`

Target	Description

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-`

Variable	Description
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 SparsityRegularizer 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 $\rightarrow$ PerceptualSpace $\rightarrow$ ConceptualSpace $\rightarrow$ SymbolicS
Reverse:  OutputSpace $\rightarrow$ SyntacticSpace $\rightarrow$ SymbolicSpace $\rightarrow$ ConceptualS
```

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:

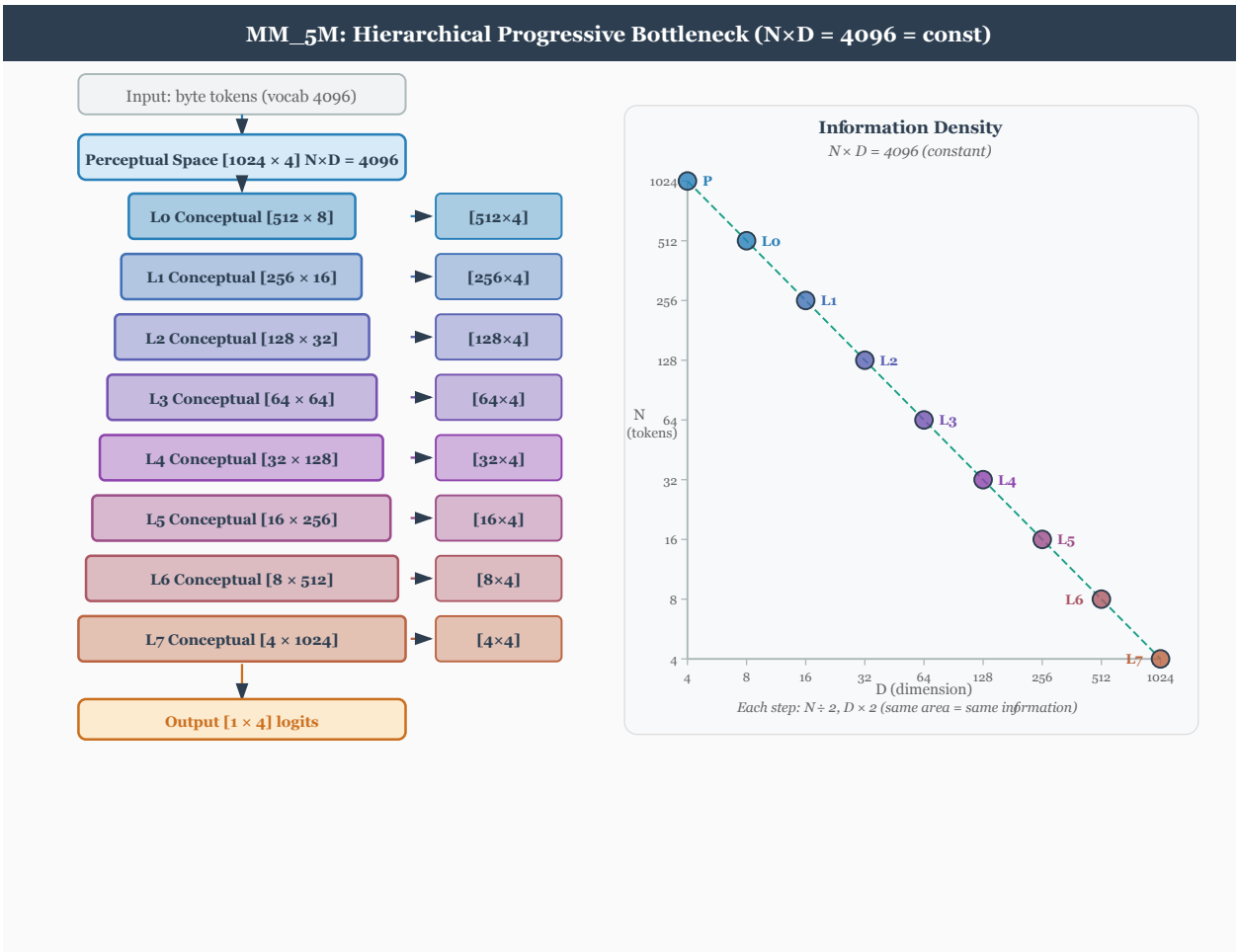
```
totalLoss = (1 - reconRatio) * outputLoss + reconRatio * reconstructionLoss
```

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 (`InvertiblePiLayer`, `InvertibleSigmaLayer`) 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 ButterflyStage wrappers and a geometrically partitioned symbol dimension. `<useGrammar>true</useGrammar>` (in `<WordSpace>`) selects a grammar-directed progressive-bottleneck variant instead. The two flags are

mutually exclusive – butterfly permutations fight constituency structure. See Reasoning.md Section Architecture Quadrants 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.

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.

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):

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

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

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

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 ± 1 toward infinity, making the layer sensitive to strong activations – the multiplicative structure emerges from the nonlinear domain transform rather than from literal products.

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. $\text{nIn}^2 + \text{rank} + \text{nOut}^2$ 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_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 $\cdot$ noise_raw_L)
U_eff = I + strict_upper(raw_U + t $\cdot$ noise_raw_U)
d_eff = b $\cdot$ d_effective + t $\cdot$ 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 $\cdot$ d_clamped + t $\cdot$ 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`	`pinv(W_eff) @ y`

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

Entity	Meaning
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).

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

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.

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:

[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.

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

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 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 Θ 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:

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

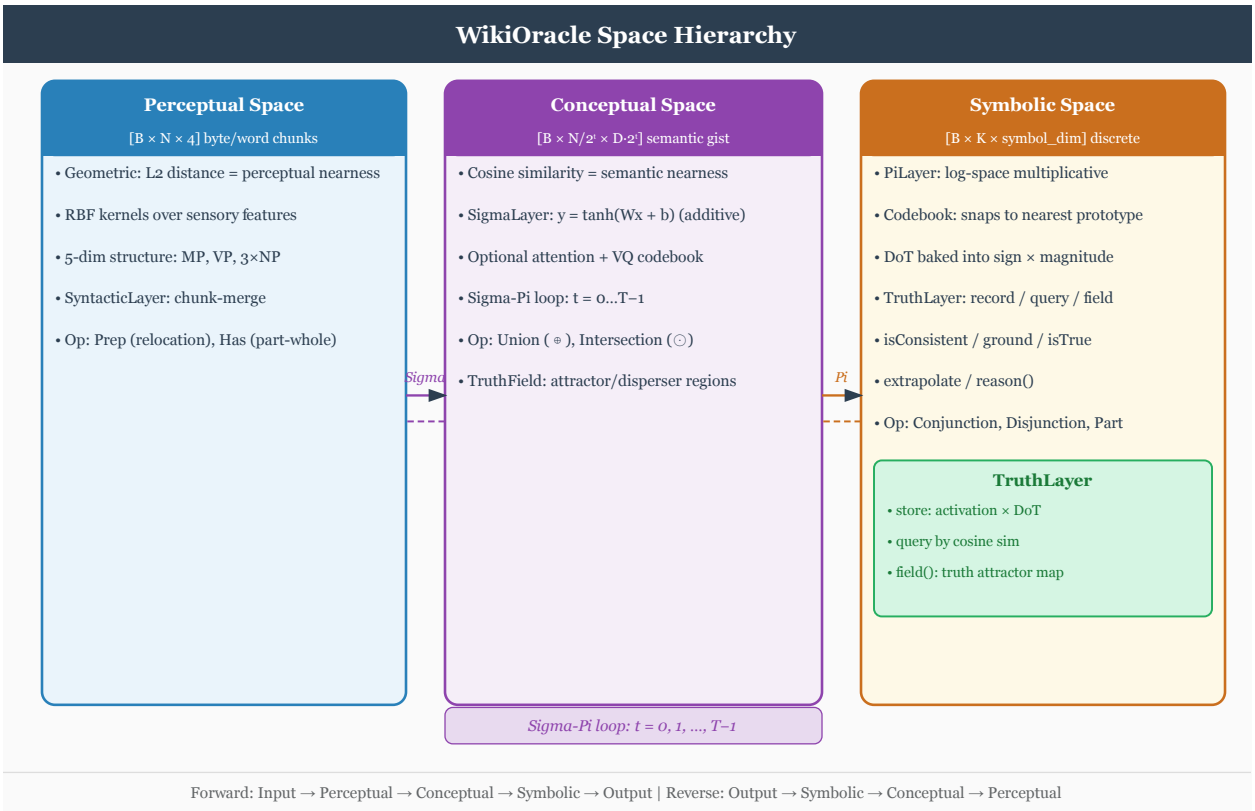
$$\Delta W_y \Delta W_x = \eta(\hat{x} - x)(\hat{y} - y)\delta_f(W^\top \cdot g^{-1}(x))\delta_g(W \cdot f^{-1}(y))f^{-1}(y)g^{-1}(x)$$

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 $\rightarrow$ PerceptualSpace $\rightarrow$ ConceptualSpace $\rightarrow$ Symbolic
Reverse:  OutputSpace $\rightarrow$ SyntacticSpace $\rightarrow$ SymbolicSpace $\rightarrow$ Conceptual
```



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.

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 th
ConceptualSpace	Combined/muxed (event encoding). Signed unit-magnitude scalars or vectors (tanh-bounded). Event n
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()`.

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.

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.

passThrough=True. Identity: input passes through unchanged, no Pi computation.

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
`passThrough`	Skip perceptual processing entirely
`hasAttention`	Enable attention reweighting

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.

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.

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.

passThrough=True. Concept vectors and activation pass through unchanged.

Key parameters.

Parameter	Description
`nActive`	Total number of symbols (output + reconstruction symbols)
`nVectors`	Codebook size (= nSymbols when codebook=true)
`passThrough`	Skip symbolic processing entirely
`codebook`	Enable codebook quantization (required for one-hot output)

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. The symbol codebook has one row per symbol, full muxed width: `SymbolicSpace.subspace.what.getW() == (nVectors, 2 + nWhere + nWhen)` with `nWhat = 2`. The leading 2 dims of each row carry the bivector `[pos_pole, neg_pole]` encoding the 4-valued (quaternary) truth of the symbol via the tetralemma / *catuskoti*: TRUE=[1,0], FALSE=[0,1], BOTH=[1,1], NEITHER=[0,0]. Trailing `nWhere + nWhen` dims carry per-symbol positional/temporal template info. `Basis.negation` swaps `(pos, neg)` on the leading 2 dims only. See BuddhistParallels.md for the catuskoti mapping.

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>true</useGrammar>`, SymbolicSpace stores an `nn.ModuleList` of per-level PiLayers in `self.pi_layers` (ButterflyStage-wrapped

when `useButterflies` is active). The symbol dimension is geometrically partitioned so each order writes only to its designated slice. See Reasoning.md Section Architecture Quadrants.

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

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 $\rightarrow$ (delete symbols) $\rightarrow$ 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 runs a single unified **S-tier** grammar over symbol activations. One class, `SyntacticLayer`, owns both rule prediction (Gumbel-softmax over rule logits) and rule semantics (the `<name>Forward` / `<name>Reverse` bodies in the `_RULE_METHODS` dispatch table). After the 2026-04-19 merge there is no longer a separate `PerceptualSyntacticLayer` / `ConceptualSyntacticLayer` / `SymbolicSyntacticLayer` trio – `WordSpace` instantiates a single `syntacticLayer` and SymbolicSpace calls it.

```
InputSpace  ->  PerceptualSpace  ->  ConceptualSpace  ->  SymbolicSpace  ->  OutputSpace
                                                          grammar runs here
                                                          17 S-tier rules
```

Prior revisions partitioned rules across three tiers (S, C, P) with transitions $S \to C \to P$. The 2026-04 redesign collapsed C into S and removed the P-tier syntactic layer (chunking became a non-syntactic pre-pass). All grammar rules now live on S-tier and execute on symbol activations; the perceptual chunking step is handled by `ChunkingLayer` outside of syntax.

Grammar

`TheGrammar` is a singleton `Grammar` instance. Under `MentalModel.xml` it is configured with **20 rules** total: 17 function-call upward rules (IDs 0-16), 2 typed upward productions (IDs 17-18), and 1 downward production (ID 19). All rules are tier `'S'` – `tier` is retained as a routing artifact from the pre-merge era; new typed categories live on the `lhs` field.

Function-call upward rules (IDs 0-16, `lhs = 'S'`):

Rule	Production	Arity	Method
0	$S \to \text{true}(S)$	1	`trueForward`
1	$S \to \text{false}(S)$	1	`falseForward`
2	$S \to \text{non}(S)$	1	`nonForward`
3	$S \to \text{conjunction}(S,\ S)$	2	`conjunctionForward`
4	$S \to \text{disjunction}(S,\ S)$	2	`disjunctionForward`
5	$S \to \text{what}(S)$	1	`whatForward`
6	$S \to \text{where}(S)$	1	`whereForward`
7	$S \to \text{when}(S)$	1	`whenForward`
8	$S \to \text{query}(S,\ S)$	2	`queryForward`
9	$S \to \text{swap}(S,\ S)$	2	`swapForward`
10	$S \to \text{equals}(S,\ S)$	2	`equalsForward`
11	$S \to \text{not}(S)$	1	`notForward`
12	$S \to \text{part}(S,\ S)$	2	`partForward`
13	$S \to \text{intersection}(S,\ S)$	2	`intersectionForward`
14	$S \to \text{union}(S,\ S)$	2	`unionForward`
15	$S \to \text{lower}(S,\ S)$	2	`lowerForward`
16	$S \to \text{lift}(S,\ S)$	2	`liftForward`

Typed upward productions (IDs 17-18, bare-symbol form, `method = 'merge'`):

Rule	Production	lhs	rhs_symbols
17	$S \to S\ VO$	`'S'`	`('S', 'VO')`
18	$VO \to V\ O$	`'VO'`	`('V', 'O')`

Downward production (ID 19, `method = 'emit_head'`):

Rule	Production	lhs	rhs_symbols
19	$S \to C$	`'S'`	`('C',)`

The dispatch table also knows `chunk` (an invertible max-union used by the perceptual pre-pass), but `chunk` is not registered in the default rule list.

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)` - `'S'`

Grammar Configuration

`Grammar.configure()` loads an XML `<grammar>` section (inside `<WordSpace><language>`). The block splits into `<upward>` (parsing / chart compose) and `<downward>` (generation from a deep symbolic state). The loader accepts three RHS forms; the **explicit-op** form is the canonical target per plans/2026-04-24-lift-lower-bivector-refactor.md Step 6 and specs/2026-04-24-lift-lower-bivector-design.md (its dispatch table maps each op name to a single `Ops.lift` / `Ops.lower` / `Ops.equal` / … call):

```
<grammar>
  <upward>
    <!-- (1) Explicit-op form (canonical per Step 6). The RHS *is* the   -->
    <!--     dispatched 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 (legacy S-tier ops). Tag = LHS, body =   -->
    <!--     op(args). Equivalent to (1) but uses the nonterminal as     -->
    <!--     the XML element name instead of a generic <rule> wrapper.    -->
    <S>not(S)</S>
    <S>swap(S, S)</S>

    <!-- (3) Bare-symbol form (transitional). RHS is a whitespace-       -->
    <!--     separated sequence of nonterminal / terminal categories;    -->
    <!--     dispatch method is the typed-compose `merge`.  Step 6 of   -->
    <!--     the lift/lower plan migrated `data/grammar.cfg` to the     -->
    <!--     explicit-op form (1); the previous bare-symbol cfg is      -->
    <!--     preserved at `data/grammar_legacy.cfg` for reference.       -->
    <S>S VP</S>
    <NP>AP NP</NP>
  </upward>
  <downward>
    <!-- One-shot head emission: project the deep state onto a          -->
    <!-- codebook and emit the best-match atom. Dispatch is              -->
    <!-- `emit_head`; no function-call form is accepted on this path.    -->
    <S>C</S>
  </downward>
</grammar>
```

The op names that may appear in the explicit-op RHS are the dispatch keys for the unified `Y = f(X1, X2=None, mode=..., kind=..., inverse=False)` signature implemented in Step 1–2 of the refactor (see Logic.md §8 *Grammatical dispatch*):

Op name in RHS	Resolves to	Notes
`lift(A, B)`	`Ops.lift(A, B, mode='OR')`	synthesis / disjunction (defau
`lower(A, B)`	`Ops.lower(A, B, mode='AND')`	analysis / conjunction (defau
`intersection(A, B)`	`Ops.lower(A, B, mode='AND')`	meet — modification rules (N
`union(A, B)` / `disjunction`	`Ops.lift(A, B, mode='OR')`	join — coordination
`conjunction(A, B)`	`Ops.lower(A, B, mode='AND')`	propositional ∧ (alias of `int
`not(A)`	`Ops.lift(A, mode='NOT')`	self-inverse pole flip / sign fli
`equals(A, B)`	`Ops.equal(A, B)`	mereological mutual parthoo
`part(A, B)`	`Ops.part(A, B)`	mereological assertion *X is* Y
`query(A, B)`	`Ops.query(A, B)` (new — spec O5)	interrogative speech act
`swap(A, B)`	`Ops.swap(A, B)` (new — spec O5)	argument-position swap
`bind(A, B)`	`Ops.bind(A, B)` (or `Ops.lower(mode='AND')` fallback)	head-complement compositio
`scale(DEG, AP)`	`Ops.scale(DEG, AP)` (or `Ops.lower(mode='AND')` fallback)	degree intensification; see spe
`project(A)` / single-symbol	identity-with-typing-stamp	terminal projection (`NP = pr
`*Reverse(Y)` / `*_inv(Y)`	`Ops.<op>(Y, inverse=True)` (multi-return)	derived at grammar-load tim

`Grammar.rules_upward` / `Grammar.rules_downward` hold the two rule sets; `Grammar.rules` is their union. Each `RuleDef` namedtuple carries `(tier, canonical, arity, method_name, lhs, rhs_symbols)` — `method_name` is the op name from the explicit-op / function-call RHS, or `'merge'` for bare-symbol-sequence rules, or `'emit_head'` for the downward `S -> C` projection. The last two fields drive the typed-category compatibility mask in the chart compose (see below). Legacy flat `<S>...</S>` directly under `<grammar>` (without the upward/downward wrapper) is still accepted by `Grammar.configure()` for Python callers, but XMLs shipped in this repo use the split form so the XSD can validate them.

Reverse productions (the downward / inverse direction of every Layer-1 forward production) are *derived mechanically* at grammar-load time — not authored as separate rules. For each forward rule `LHS = op(arg1, arg2)` the loader synthesizes `arg1, arg2 = opReverse(LHS)` with the multi-return calling convention from specs/2026-04-24-lift-lower-bivector-design.md O10.

`data/grammar.cfg` dispatch (Step 6)

Step 6 of the lift / lower / bivector refactor introduced a text-based rule table at `data/grammar.cfg` in the explicit-op form, alongside the XML loader. Format (line-oriented):

```
# comments start with '#'
[upward]
S = NP                        # PROJECT (single-category RHS)
S = lift(NP, VP)
NP = intersection(AP, NP)
S = not(S)
...

[downward]
C = emit_head(S)

[layer2]
# Post-hoc S-ops; not productions, no reverse pair.
S = true(S)
S = swap(S, S)
...
```

Sections are bracketed (`[upward]` / `[downward]` / `[layer2]`) and default to `[upward]` when no header is seen. Each `LHS = body` rule's body is either a function call `op(arg1, arg2)`, a single category (PROJECT), or `epsilon`. Layer-2 rows are tracked separately on `Grammar.rules_layer2` and stay out of `Grammar.rules` because they're post-hoc transformations, not productions.

Loading is gated via `<WordSpace><language><grammarCfg>` in the XML config — when set to a path (resolved against `ProjectPaths.PROJECT_DIR`), `Grammar._ensure_configured()` calls `Grammar.load_from_cfg(...)` and skips the legacy `<grammar>` block. When unset, the XML grammar wins (legacy path). Both paths populate `Grammar.reverse_rules` with the mechanically-derived Layer-2.5 reverse productions `(args_tuple, reverse_op_name, (lhs,))`.

Dispatch itself is unchanged: `SyntacticLayer.project` looks up `method_name` in `_RULE_METHODS` and calls the matching `<name>Forward` on the layer. The cfg path supplies the `method_name`; the layer methods supply the Basis / mask / subspace context. For callers that want to invoke an op stateless (no Basis context), `SyntacticLayer.dispatch_ops(op_name, *args, **kwargs)` resolves the op name through `_OPS_METHODS` and calls the bare `Ops` callable directly — useful for the explicit-op view from grammar.cfg without going through layer plumbing.

The reverse op name follows the convention `<op>Reverse` for everything except self-inverse ops (`not`) which keep their forward name; PROJECT rules (single-category RHS) emit the explicit `projectReverse` reverse name. Multi-return reverses `Ops.liftReverseAll(Y[, W])` and `Ops.lowerReverseAll(Y[, W])`

return `(X1, X2)` tuples consistent with the parent plan §Step 6 spec; the existing analytic single-arg forms `Ops.liftReverse(result, right)` and `Ops.lowerReverse(result, right)` are unchanged.

Thought-Free Mode (Shamatha Speech)

`Grammar.thought_free` is a boolean flag (default `False`) settable per-request by WikiOracle's `serve.py` before inference and reset in a `finally` block. Under the old three-tier grammar it suppressed C-level `not`; after the S-merge its filtering effect is minimal pending redesign. See Params.md for the XML configuration.

Ops Reference

The grammar dispatches each rule to an `Ops.*` callable defined in `bin/Layers.py` (class `Ops`). This section is the canonical reference for those callables: their current implementation, their position on the layer-anchor map, and their status against the target architecture (where every grammar op has a clean grammatical interpretation).

Layer-anchor map

Every grammar op should ultimately be implemented at one of four anchor sites, reflecting where its semantics actually live:

Anchor	Lives at	Description
Sym	SymbolicSpace	per-symbol algebra on `subspace.activation` (th
SigmaPi	`ConceptualSpace.pi` ↔ `SymbolicSpace.sigma` round trip	level-crossing operations. `lift` runs `forwardSigm`
Per	PerceptualSpace	percept-side operations (e.g. `chunk`). Currently o
Def	WordSpace	*definitional*: introduces a new symbol or asserts a

The xAI / HI precondition is that **every** op has an explicit grammatical interpretation, anchored at one of the four sites. The table below shows the mapping; the per-op subsections that follow describe the current implementation and what changes when the op migrates to its target anchor.

Target ↔ current ↔ status table

Op	Target	Current `Ops.*` body
`true`	Sym	`Ops.positive(x) = relu(x)`
`false`	Sym	`Ops.negative(x) = -relu(-x)` (or `relu(-x)` via `_OPS_METHODS`)
`non`	Sym	'1 -
`not`	Sym	`Ops.negation(x)` — bitonic sign flip; monotonic does paired-pole swap on bivector l
`conjunction`	Sym	`Ops.lower(x, y, mode='AND', kind=...)` (smooth: product; strict: min; radial: F
`disjunction`	Sym	`Ops.lift(x, y, mode='OR', kind=...)` (smooth: mean; strict: max; radial: RadM
`intersection`	SigmaPi	`Ops.lower(x, y, mode='AND')` (alias for `conjunction` in `_OPS_METHODS`)
`union`	SigmaPi	`Ops.lift(x, y, mode='OR')` (alias for `disjunction`)
`lift`	SigmaPi	`Ops.lift(X1, X2, mode, kind, inverse, monotonic)` — pure stateless dispatche
`lower`	SigmaPi	`Ops.lower(X1, X2, mode, kind, inverse, monotonic)` — pure stateless dispatch
`equal`	Def	`Ops.equal(x, y) = part(x, y) * part(y, x)` (mutual parthood, vector or scalar)
`part`	Def	`Ops.part(x, y) = x * (y / ‖y‖)` (vector) or clipped cosine in `[0, 1]` (scalar)
`what` / `where` / `when`	Sym	`Ops.{what,where,when}(x, nWhat, nWhere, nWhen)` — block selector zeroing the
`query`	Mereological	`Ops.query` doesn't exist as an `Ops.*` body — handled in `compose()` via norm-drop c
`swap`	Sym	Sinkhorn soft-permutation matrix P over 3 slots; $\mathrm{swap}(\ell, r) = (P \cdot [\ell, r, m]^\top)_0$
`chunk`	Per	`Basis.disjunction(`ℓ, r`, monotonic=True)` $\to \max(\ell, r)$ fallback; runs in the percep

The remainder of this section is the per-op specification: signature, math, current location, and a note on what migrating to the target anchor would change.

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 (e.g. a bitonic `Basis`), which supplies pointwise primitives `pos`, `negation`, `conjunction`, `disjunction`, `equal`, `part`. When $\mathcal{B}$ is absent the methods fall back to torch elementwise primitives. `mask` (optional concept-axis `Mask`) gates the output by zeroing non-selected dimensions; it is omitted below for clarity.

Ternary operators (true / false / non) — Sym

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

$$\begin{aligned}\text{true}(x) &= \mathcal{B}.\text{pos}(x) = \max(0,\ x)\\ \text{false}(x) &= \mathcal{B}.\text{pos}(-x) = \max(0,\ -x)\\ \text{non}(x) &= 1 - |x|\end{aligned}$$

These three operators partition unity: $\mathrm{true} + \mathrm{false}

- \mathrm{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 (no inverse registered).

Negation (not) — Sym

$$\text{not}(x) = \mathcal{B}.\text{negation}(x) = -x, \qquad \text{not}^{-1} = -x$$

Pure antipodal flip on the bitonic axis. Self-inverse.

Conjunction / disjunction — Sym

Both route through the unified `lift` / `lower` dispatcher (Step 1–2 of plans/2026-04-24-lift-lower-bivector-refactor.md). At the S tier the explicit-op grammar form `S = conjunction(S, S)` resolves to `Ops.lower(S1, S2, mode='AND')` and `S = disjunction(S, S)` resolves to `Ops.lift(S1, S2, mode='OR')`:

$$\text{conjunction}(\ell, r) = \mathcal{B}.\text{lower}(\ell, r,\ \text{mode='AND'}) \xrightarrow[\text{kind='strict'}]{} \min(\ell, r)$$

$$\text{disjunction}(\ell, r) = \mathcal{B}.\text{lift}(\ell, r,\ \text{mode='OR'}) \xrightarrow[\text{kind='strict'}]{} \max(\ell, r)$$

The `kind` parameter selects the forward point body: `'strict'` (default — lattice min/max), `'smooth'` (elementwise product / mean, the differentiable form used by `Pi/SigmaLayer.forward`), or `'radial'` (RadMin / RadMax — same-sign min/max magnitude with zero passthrough; equivalent to the historical `monotonic=False` body of `Basis.conjunction` / `Basis.disjunction`). The legacy thin-forwarder methods `Basis.conjunction(x, y, monotonic=...)` / `Basis.disjunction(x, y, monotonic=...)` are preserved as deprecated aliases and route through the new dispatcher (`monotonic=True` → `kind='strict'`; `monotonic=False` → `kind='radial'`) so existing callers see bit-exact output.

Sentence-level fuzzy $\wedge$ and $\vee$ on bitonic activations. Both are lossy at the sentence level (no inverse registered for the S-tier usage), though the same primitives at C-tier subspace level retain Basis-level inverses via `Ops.conjunctionReverse` / `Ops.disjunctionReverse` (codebook-search recovery).

Intersection / union — SigmaPi (currently aliases `lower/lift`)

`intersection` and `union` are aliases for `conjunction` / `disjunction` under the explicit-op grammar form, dispatching the same unified call:

$$\text{intersection}(\ell, r) = \mathcal{B}.\text{lower}(\ell, r,\ \text{mode='AND'}) \qquad \text{union}(\ell, r) = \mathcal{B}.\text{lift}(\ell, r,\ \text{mode='OR'})$$

Inverses are the codebook-search recoveries on r (supplied by `Basis` which carries the codebook W):

$$\text{intersection}^{-1}(y, r) = \mathcal{B}.\text{lower}(y, r,\ \text{mode='AND'},\ \text{inverse=True}) \qquad \text{union}^{-1}(y, r) = \mathcal{B}.\text{lift}(y, r,\ \text{mode='OR'},\ \text{inverse=True})$$

For `mode='NOT'` the inverse is identical to the forward (self-inverse pole flip / sign flip).

Part — Def (currently a tensor score)

Parthood score, weighting the right operand by its containment in the left:

$$\text{part}(\ell, r) = \mathcal{B}.\text{part}(\ell, r,\ \text{scalar=True}) \cdot r = \frac{\max(0,\ \ell \cdot r)}{|\ell|\,|r|} \cdot r$$

The scalar $s \in [0, 1]$ is broadcast to r's rank before multiplication. Empty-operand contract: if $|\ell|$ or $|r|$ is near zero, $s = 1$ (the empty set is part of everything). Lossy.

Equal — Def (currently a cosine-gated projection)

$$\text{equals}(\ell, r) = \mathcal{B}.\text{equal}(\ell, r,\ \text{scalar=True}) \cdot r$$

"Pass r upward only to the degree ℓ matches it." The equal score is mutual parthood $\mathcal{B}.\text{equal}(\ell, r) = \text{part}(\ell, r) \cdot \text{part}(r, \ell)$, scalar-reduced and broadcast to r.

Under a `mask`, equals degenerates to an L1-distance agreement score on the selected dims:

$$\text{equals}^{\text{mask}}(\ell, r) = \text{clip}\left(1 - \frac{\sum_d m_d\,|\ell_d - r_d|}{\sum_d m_d},\ 0,\ 1\right) \cdot r$$

When the owning `SymbolicSpace` supplies a `PiLayer` back-reference and both operands match the Pi output dim, `equalsForward` first reverse- projects operands to concept space and invokes the concept-basis `equal` on the bitonic subspace. Otherwise it uses the local basis. Lossy.

Slot selectors (what / where / when) — Sym

Given per-subspace column widths $(n_{\text{what}}, n_{\text{where}}, n_{\text{when}})$:

$$\text{what}(x)[\dots, i] = \begin{cases} x[\dots, i] & 0 \le i < n_{\text{what}} \\ 0 & \text{otherwise} \end{cases}$$

$$\text{where}(x)[\dots, i] = \begin{cases} x[\dots, i] & n_{\text{what}} \le i < n_{\text{what}} + n_{\text{where}} \\ 0 & \text{otherwise} \end{cases}$$

$$\text{when}(x)[\dots, i] = \begin{cases} x[\dots, i] & n_{\text{what}} + n_{\text{where}} \le i \\ 0 & \text{otherwise} \end{cases}$$

Parameter-free axis projections. In 2D (`[B, N]`) activation mode the block structure is unavailable, so selectors degenerate to identity – the axis semantics are carried by the subspace's modality tensors rather than the flat activation vector. Lossy.

Query — Mereological (currently accumulator preservation)

$$\text{query}(\ell, r) = \ell$$

The query marker is pushed onto WordSubSpace at the accumulation point by `compose()` when it detects norm-drop (see below), not by `queryForward` itself. When the parse tree is re-evaluated downstream, query returns the preserved accumulator (the left operand). Lossy.

Swap (Sinkhorn soft permutation) — Sym

Let $M \in \mathbb{R}^{3\times 3}$ be a learned logit matrix. Apply $k = 5$ Sinkhorn normalisation iterations to produce a doubly-stochastic soft permutation P:

$$M^{(0)} = M, \qquad M^{(t+1)} = M^{(t)} - \text{logsumexp}_{\text{cols}}(M^{(t)}) - \text{logsumexp}_{\text{rows}}(M^{(t)}), \qquad P = \exp(M^{(k)})$$

Given operands ℓ, r and a learned broadcast marker m, stack and contract along the three "slots":

$$\text{swap}(\ell, r) = \left(P \cdot [\ell,\, r,\, m]^{\top}\right)_0$$

i.e.\ the first row of the mixed-slot stack. The permutation is shared across batch / position / dim. Lossy.

Lift / lower — SigmaPi (mode-dispatch ops)

`Ops.lift` and `Ops.lower` are unified mode-dispatchers. Default modes are `lift` → `'OR'` (synthesis, $\vee$) and `lower` → `'AND'` (analysis, $\wedge$); each accepts `kind` ∈ `{'strict', 'smooth', 'radial'}` selecting the forward body, and `inverse=True` for the reverse direction.

Forward bodies (point):

	`mode='AND'` (`lower`)	`mode='OR'` (`lift`)
`kind='strict'`	$\min(X_1, X_2)$	$\max(X_1, X_2)$
`kind='smooth'`	$X_1 \cdot X_2$ (elementwise product)	$(X_1 + X_2)/2$ (mean)
`kind='radial'`	RadMin (same-sign min magnitude, zero-collapse)	RadMax (same-sign max magnitude, zero-passthroug

Region forms (operand is a tuple (ℓ, u)): `lower(mode='AND')` returns $(\max(\ell_1, \ell_2), \min(u_1, u_2))$; `lift(mode='OR')` returns $(\min(\ell_1, \ell_2), \max(u_1, u_2))$ — independent of `kind`.

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

Reverse / inverse:

- `inverse=True, mode='AND' or 'OR'` requires a codebook `W` and raises `NotImplementedError` if unsupplied — callers must use the `Basis.*` flavor, or `Ops.conjunctionReverse(result, y, W)` / `Ops.disjunctionReverse(result, y, W)` directly.
- `Ops.lowerReverseAll(Y, W, monotonic)` and `Ops.liftReverseAll(Y, W, monotonic)` return the multi-operand pair `(recovered_left, recovered_right)` per the parent plan §Step 6 Layer-2.5 grammar convention.

- The single-operand analytic forms `Ops.liftReverse(result, right)` = result/(right + ε) and `Ops.lowerReverse(result, right)` = $2 \cdot$ result − right are preserved as legacy callers' inverse of the old `Ops.lift` = $\odot$ and `Ops.lower = mean` bodies. They warn-only and are permanent.

Layer integration. The cfg-driven grammar dispatch invokes `Ops.lift` / `Ops.lower` directly (option (a) per the Step 6.5 closure: explicit two-operand calls). 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 between dispatch-time algebra and learnable layers is the `binary=True` flag on `PiLayer.forward` / `SigmaLayer.forward` (Step 5): it pre-applies `Ops.top2_select_ste` to hard-select the top-2 input operands before the layer body runs.

Chunk — Per (invertible max-union; pre-pass)

$$\text{chunk}(\ell, r) = \mathcal{B}.\text{disjunction}(\ell, r,\ \text{monotonic=True}) \xrightarrow[\text{fallback}]{} \max(\ell, r)$$

Monotonic max-union. Inverse via $\mathcal{B}$.disjunction_inverse(y, r). Not registered in the default S-tier rule list; used by `ChunkingLayer` in the perceptual pre-pass. This is the only Per-anchored op currently shipping.

Reverse / inverse ops

These are the analytic and codebook-search inverses paired with the forward ops above. None are dispatched by the cfg grammar directly — they are invoked by `Basis.*` methods or by the parent plan's Layer- 2.5 grammar reverse convention.

Op	Behavior
`Ops.negationReverse(x, monotonic)`	self-inverse — calls `Ops.negation(x, monotonic)` again
`Ops.conjunctionReverse(result, y, W, monotonic)`	codebook-search recovery — find cb_i such that conjunction(cb_i, c
`Ops.disjunctionReverse(result, y, W, monotonic)`	dual of `conjunctionReverse`; recovers left operand of disjunctio
`Ops.liftReverse(result, right)`	analytic single-operand inverse of the legacy `Ops.lift` = $\odot$ body
`Ops.liftReverseAll(Y, W, monotonic)`	multi-return Step-6 form: (recovered_left, Y) via codebook searc
`Ops.lowerReverse(result, right)`	analytic inverse of legacy `Ops.lower = mean`: $2 \cdot$ result − right.
`Ops.lowerReverseAll(Y, W, monotonic)`	dual of `liftReverseAll` — multi-return form for the Step 6 gra

The `_binary_op_inverse_impl(result, W, op, monotonic)` private helper backs both `conjunctionReverse` and `disjunctionReverse`. It searches the codebook W for the pair (cb_i, cb_j) whose forward *op* best matches `result`, returning cb_i.

Mereological suite — see Mereology.md

The mereology suite (`part`, `whole`, `equal`, `overlap`, `underlap`, `boundary`, `copart`) lives in `Ops.*` but the canonical specification of its semantics is Mereology.md. Each member has a vector form (default) and a scalar form (`scalar=True`):

Op	Vector form	Scalar form
`part(x, y)`	$x \cdot (y/\lvert y\rvert)$	clipped cosine $\max(0, x \cdot y)/(\lvert x\rvert\lvert y\rvert) \in [0, 1]$
`whole(x, y)`	$(1 - x) \cdot (y/\lvert y\rvert)$	part$(y, x$, scalar=True)
`equal(x, y)`	part$(x, y) \cdot$ part(y, x)	mutual parthood $\in [0, 1]$ — 0 disjoint, $\in (0, 1)$ overlap, 1 identity
`overlap(x, y)`	min(part(x, y), part(y, x))	boolean: $0 <$ equal(...) < 1
`underlap(x, y)`	min(whole(x, y), whole(y, x))	boolean: equal(...) $= 0$
`boundary(x, y)`	$\lvert$part$(x, y) -$ part$(y, x)\rvert$	always 0 under the symmetric clipped-cosine score
`copart(x, y)`	$y - x$	$1 -$ part$(x, y,$ scalar=True)

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

`part` and `equal` are also surfaced through the grammar dispatch table; the rest of the suite is utility-level (called by `Basis.*` and the TruthLayer derivation pipeline). The target architecture moves `part` and `equal` from tensor scoring to **Def** anchors that introduce a recorded relation in WordSpace; see the table at the top of this section.

Pointwise primitives

These are the small utility ops with no current grammatical role — they are called by the layer / loss / encoding code paths and have no dispatch entry. Listed for completeness so the inventory is closed.

Constants: `Ops.true() = 1`, `Ops.false() = -1`, `Ops.unknown() = 0`.

Element-wise:

Op	Body
`Ops.positive(x)`	$\mathrm{relu}(x)$
`Ops.negative(x)`	$-\mathrm{relu}(-x)$
`Ops.neutral(x)`	$1 - \lvert x \rvert$
`Ops.pos(x)`	$\mathrm{relu}(x)$ — alias used by the bivector path
`Ops.sign(v)`	signum with $\mathrm{sgn}(0) = 1$ (not 0)
`Ops.saturate(x)`	$\mathrm{clamp}(x, -1, 1)$, NaN $\to 0$
`Ops.threshold(x, t)`	zero entries with $\lvert x \rvert < t$
`Ops.complement(x)`	$\mathrm{sign}(x) - x$
`Ops.convertSensation(x)`	$2x - 1$ — $[0, 1] \to [-1, 1]$ remap
`Ops.minMag(x_1, x_2)`	x_i with smaller $\lvert x_i \rvert$
`Ops.maxMag(x_1, x_2)`	x_i with larger $\lvert x_i \rvert$
`Ops.error(x_1, x_2)`	$\lvert x_1 - x_2 \rvert_2$ — scalar L2
`Ops.norm(x)`	last-dim L2 norm
`Ops.distance(x, y, monotonic, dim)`	$(1 - \cos(x, y))/2$ (bitonic) or volume-weighted L2 (monotonic)

Predicates (return Boolean):

Op	Body
`Ops.isActive(x, t)`	$\lvert x \rvert \geq t$
`Ops.isEqual(x_1, x_2)`	`torch.equal` strict equality
`Ops.isReducer(x_1, x_2)`	$\lvert x_2 - x_1 \rvert_1 < \lvert x_2 \rvert_1$ — scalar Bool

Selection (Step-5 STE bridge): `Ops.top2_select_ste(x, dim=-1)` selects the top-2 entries of x by $\lvert x \rvert$, zeroing the rest. Backward is straight-through. Used by `PiLayer.forward(x, binary=True)` and `SigmaLayer.forward(x, binary=True)` to specialize the layer body to per-rule binary application.

Private helpers (referenced by the public ops above): `_radmin(x, y)`, `_radmax(x, y)` — the same-sign min / max magnitude bodies used by `kind='radial'`; `_binary_op_inverse_impl(result, W, op, monotonic)` — the codebook-search backbone for `conjunctionReverse` / `disjunctionReverse`; `_as_region(x)` — promotes a point to a degenerate region $(\ell, u) = (x, x)$ for the region-form bodies.

Rule Prediction (`forward`)

`SyntacticLayer.forward(x)` 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_{deriv}, W_{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, ι = grammar.interpretation, t is the index of the transition rule (if present), and τ is the Gumbel-softmax temperature (annealed from 1.0 toward 0.1 via `set_tau`).

Output: `{rule_logits, rule_probs, predicted_rules, words}`.

Differentiable rule selection

- **Training:** `gumbel_softmax(logits, tau)` – soft one-hot, gradients flow through all candidate rules proportional to probability.
- **Eval:** `softmax` + `argmax` – discrete rule selection.
- **Annealing:** τ starts at 1.0 (diffuse) and decreases toward 0.1 (near-discrete) over training.

Projection: `project(grammar, rule_id, left, right)`

Rule dispatch flows through `SyntacticLayer.project()`. It looks up `(forward_name, reverse_name, is_binary)` in `_RULE_METHODS[method_name]` and calls the named `<name>Forward`. Inputs are `[B, N]` activations (2D mode) or `[B, N, D]` vectors (3D mode). (The "bivector" name is reserved in the code for the 2-component `[..., :2]` slice of a symbol codebook row, not the full `D`-wide vector.)

`reverse_project` is the dual, used during `decompose` to invert the registered reversible rules.

Composition: `compose(data, subspace, grammar, target_count=None)`

`SyntacticLayer.compose()` is the unified two-phase driver. It dispatches on input rank:

- `data.ndim == 2` –> `_compose_activation` (2D activation mode)
- `data.ndim == 3` –> `_compose_vector` (3D vector mode, with optional pairwise reduction via `_compose_to_target`)

Returns `(composed, svo_or_None)`. `svo` is a `(subject, verb, object)` tuple of `[B, 1, D]` tensors when the chart-compose path (see below) finds a canonical `S -> S VO` firing and its matching `VO -> V O`; otherwise `None`.

Phase 1: deterministic `not`

Before soft superposition (3D path only), apply a single deterministic `not` to the top-of-stack when that vector has a negative mean:

$$\text{for each batch } b \text{ with top-of-stack position } p_b: \qquad \text{if } \overline{\text{data}[b, p_b]} < 0 \implies \text{data}[b, p_b] \leftarrow \text{not}(\text{data}[b, p_b]),$$

and a `not` word is pushed for b at position p_b. This strips any leading negation from the accumulator before it mixes with siblings.

Phase 2: soft-weighted cascading composition

Let $v_i^{(b)} \in \mathbb{R}^D$ be the i-th active leaf vector in batch b, and let $L_b = |\{\text{active positions in } b\}|,\ L_{\max} = \max_b L_b$. Build leaf slabs $v[\cdot, i, \cdot, \cdot]$ of shape $[B, L_{\max}, N, D]$ by masking. Initialize the accumulator:

$$a^{(0)} = v[\,\cdot\,,\ 0\,,\ \cdot\,,\ \cdot\,]$$

Let $\mathcal{R}^\star$ be the *composable* rule set: the grammar S-rules minus `not` (already applied in Phase 1). Let $p^{(d)} \in \Delta^{|\mathcal{R}^\star|}$ be the rule-prediction probabilities for depth d renormalised over $\mathcal{R}^\star$.

Iterate depth $d = 0, 1, \ldots$ until $d \geq D_{\max}$ or the next leaf index exceeds $L_{\max}$:

$$\ell = a^{(d)}, \qquad r = v[\cdot,\, d+1,\, \cdot,\, \cdot]$$

$$\text{for each } \rho \in \mathcal{R}^\star : \qquad u_\rho^{(d)} = \begin{cases} \rho(\ell, r) & \text{arity}(\rho) = 2 \\ \rho(\ell) & \text{arity}(\rho) = 1 \end{cases}$$

Soft candidate (training) vs. argmax candidate (eval):

$$c^{(d)} = \begin{cases} \sum_\rho p_\rho^{(d)}\, u_\rho^{(d)} & \text{training} \\ u_{\rho^\star}^{(d)}, \quad \rho^\star = \arg\max p^{(d)} & \text{eval} \end{cases}$$

Query / norm-drop detection Detect symbolic contradiction at the accumulation point by comparing the pre- and post-composition Frobenius norms per batch row:

$$\nu_\ell = \|\ell\|_F\,, \qquad \nu_c = \left\|c^{(d)}\right\|_F\,, \qquad \text{qmask}_b = (\nu_{\ell,b} > 10^{-6}) \wedge (\nu_{c,b} < \kappa\, \nu_{\ell,b})$$

with $\kappa = 0.1$ (`_QUERY_NORM_DROP_RATIO`). If qmask_b fires, the new accumulator preserves the prior state and a query word is pushed onto WordSubSpace with leaves $(\text{last_rule}_b,\ \arg\max_\rho p_b^{(d)},\ -1)$; otherwise accept the candidate:

$$a_b^{(d+1)} = \begin{cases} \ell_b & \text{qmask}_b = 1 \\ c_b^{(d)} & \text{qmask}_b = 0 \end{cases}$$

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

Advance $d \leftarrow d + 1$ and $\text{leaf_idx} \leftarrow \text{leaf_idx} + 1$ (or $+2$ for arity-3 rules when a third leaf is available).

Leaf ledger (codebook indices)

Before Phase 1 begins, codebook indices of the original leaves are snapshotted (so the ledger reflects the pre-`not` input exactly):

$$\text{cb_idx}[b, i] = \arg\max_k \left(\text{data}[b, i] \cdot \text{cb}[k]^\top\right)$$

and each active leaf emits a transition word with `order = -1` and `leaf1 = cb_idx[b, pos]`. This ledger is what `decompose` uses to reconstruct the pre-compose tensor exactly.

2D activation path (`_compose_activation`)

The 2D path runs the same depth-wise mixture but on `[B, N]` scalar activations:

$$a^{(d+1)} = \sum_{\rho} p_{\rho}^{(d)} \cdot \begin{cases} \rho(a^{(d)}, v[\,\cdot\,,\, d+1]) & \text{arity}(\rho) = 2 \\ \rho(a^{(d)}) & \text{arity}(\rho) = 1 \end{cases}$$

with the same query/norm-drop logic (norm over the N-axis instead of the $[N, D]$-plane). This path is used when the caller forwards norms directly (legacy S-tier driver).

Pairwise reduction: `_compose_to_target(data, subspace, grammar, target_count)`

When `target_count` is supplied, the 3D path reduces the active-token count down to that value via iterated pairwise reductions. Composable rules in this path are the arity-2 rules minus `not`.

At each outer iteration d:

1. Count active positions per batch. Stop if $\max_b |A_b| \leq$ target_count.
2. For each batch, walk pairs (p_{2i}, p_{2i+1}) while reduction would still exceed the target. For each pair, compute all composable rule outputs, take either the soft mixture (training) or argmax (eval), write the result to the left position, zero the right, and append the left position to the new active list.
3. Record the chosen rule as a word at `left_pos` with the current outer-iteration order.
4. Carry forward any leftover unpaired positions.

Finally, extract the non-zero subset of `data` at the surviving positions, zero-fill the rest, and return.

Chart-like Compose (typed SVO)

Selected by `<WordSpace><chartCompose>true</chartCompose></WordSpace>`. The cascade-based `_compose_vector` above builds a strictly left- associative tree `(((leaf0, leaf1), leaf2), ...)`; it cannot produce `(S, (V, O))` from an `N V N` sentence because step 1 is forced to merge `leaf[0]` with `leaf[1]` (an `N V` pair) and no rule in a typed grammar fires on that shape. Chart compose replaces the cascade with per-step pair selection, so the model can reduce `V O` first and then `S VO` second.

Pair-scorer + per-step argmax

At each depth step:

1. A pair-scorer MLP takes the current rule-prediction hidden state plus `(left, right)` feature slices and produces a softmax over the `N_alive - 1` adjacent live-leaf pairs. Input width is `hidden_dim + 2 * feature_dim`; `feature_dim` is a first-class `SyntacticLayer.__init__` kwarg so the pair-scorer lines up with the actual leaf dim at compose time (not `n_slots`, which tests happen to set equal to `D`). Note: until `MentalModel` pipes the rule-prediction hidden through, `_compose_vector_chart` passes an all-zero `[B, hidden_dim]` tensor (see `Language.py:1354`); the pair scorer therefore conditions only on `(left, right)` features today.
2. The rule head emits the usual per-depth softmax over composable rules (upward rules only; downward is emit-time, not chart-time).
3. A **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 `O` (`'?'`) is a wildcard, so unseeded leaves can match any typed rule during warmup.
4. The joint `pair_probs * rule_probs * compat` is renormalised. In training the merge output is the soft mixture; at eval the argmax pair / rule pick the concrete merge site.

5. The merged vector replaces the left slot in an out-of-place `torch.where`-style update (inplace mutation on the live tensor would break autograd because `live = data.clone()` carries `grad_fn`). The right slot is marked dead via the alive mask; the merged slot's category becomes the rule's LHS.

The reduction runs for at most `N - 1` steps, so `N` live leaves collapse to one in `N - 1` merges.

Category tensor + POS seeding

Alongside `data: [B, N, D]`, the chart path carries `category: [B, N]` of category IDs (the `Grammar.rules`-derived table is stored on `SyntacticLayer._category_names` / `_category_index`, with `'?'` at index 0 as the wildcard). `_seed_category` lets upstream code pre-populate slot categories.

Category seeding is currently a no-op stub; the wildcard `?` keeps the compat mask permissive. Upstream code may call `_seed_category` directly to bias the chart toward specific categories when a domain-specific tagger is available.

Derivation trace

`SyntacticLayer._derivation_trace` is a per-batch list (one list per row in the batch) of 5-tuples `(rule_id, left_slot, right_slot, merged_slot, merged_category)` appended on every chart merge. The trace is the only surface that downstream readers (SVO extractor, future reconstruct templates) rely on; the `live` tensor is already compacted by the time compose returns.

Grammar-derived SVO

`SyntacticLayer._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` so `truth_modulated_loss` treats it as "no score". There is no positional fallback — the earlier heuristic that labelled positions `0/1/2` as `S/V/O` is gone. Consumers read `SyntacticLayer.last_svo` exactly as before.

Downward Head Emission (`S -> C`)

Selected by `<WordSpace><downwardGeneration>true</downwardGeneration></WordSpace>`. After the upward parse reduces `N` leaves to a single root vector, `MentalModel.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.

`SyntacticLayer.emit_head(state, codebook) -> (best_idx, contained, residual)`

Projection-coefficient scoring, not cosine:

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

The per-row argmax is `best_idx`. Because `state` is *not* L2- normalised (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 on a `None` codebook.

Returns `{heads: list[int] (len B), contained: [B, D], residual: [B, D], state: [B, D]}`. `MentalModel.forward` stashes `heads` as `self._predicted_head` — a per-batch list of vocabulary indices — so a training loss can compare against a supervised head token, and so `bin/interact_head.py` (interactive smoke test) can print the decoded word after each forward pass.

Stable hooks already wired

- `SyntacticLayer.last_svo: Optional[Tuple[Tensor, Tensor, Tensor]]`
- `SyntacticLayer.lifting_layer: LiftingLayer` (created in `init_lifting`, called from `WordSpace._build_syntacti`
- `MentalModel.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.
- `MentalModel._predicted_head: Optional[list[int]]` set by the downward pass above.

The historical design narrative (why chart compose was needed, why a cheaper hybrid doesn't work, curriculum plan for training) lives in the "Design History: Learned SVO Plan" appendix at the end of this document, and the implementation tracker at `doc/plans/2026-04-20-LearnedSVO-integration.md`.

Decomposition: `decompose(data, subspace, grammar)`

Reconstruct the pre-compose tensor from the symbolic word record.

- If a codebook is available and shapes match, look up each terminal word (those with `order = -1`) and place `cb[leaf1]` at the recorded position, producing the exact pre-compose tensor.
- Otherwise, fall through to the degraded rule-reversal pass: walk words in reverse and invert the registered reversible rules (`not`, `intersection`, `union`, `lift`, `lower`). Lossy rules (`equals`, `part`, `true`, `false`, `non`, `what`/`where`/`when`, `query`, `swap`, `conjunction`, `disjunction`) pass through.

Two-Layer Logic

The logic system operates at two levels:

Subsymbolic Layer (Vectors)

Objects are vector sets (B, N, D) interpreted as RBF / luminosity fields.

Operator	Implementation	Meaning
union	$\max(l, r)$	strongest affirmation
intersection	$\min(l, r)$	shared commitment
negation (not)	$-x$	antipodal opposition on hypersphere
non	$\alpha x,\ \alpha \in [0, 1)$	contraction toward zero (withdrawal of assertion)
parthood	fuzzy max-coverage $\in [-1, 1]$	signed containment

Symbolization

Map vectors -> scalar truth strength:

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

Interpretation: +1 -> strong presence, 0 -> neutral, -1 -> absence.

Symbolic Layer (Scalars in [-1,1])

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

Operator	Formula	Meaning
neg	$-a$	oppositional negation
non	αa	withdrawal / neutrality
union	$\max(a, b)$	strongest affirmation
intersection	$\min(a, b)$	shared commitment
part	$\text{clamp}(b - a, -1, 1)$	signed containment

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

Word Encoding

Each word is a 3-tuple `(batch, vector, rule)`:

- **batch** - index into the batch dimension $[0, B)$
- **vector** - index into the activation vector $[0, N)$
- **rule** - grammar rule ID from `TheGrammar` $[0, K)$

Words are stored as a Python list of tuples on the SubSpace. The list is a **derivation tree** in pre-order: the first entry is the root, and binary rules expand left-first. Terminal words carry `order = -1` and a codebook index in the `leaf1` slot; the leaf ledger used by `decompose` is built from these entries.

WordSpace and SymbolicSpace wiring

Under `MentalModel.xml`, a single `WordSpace` hosts one `syntacticLayer` attribute, built by `_build_syntactic_layer` for the SymbolicSpace. The driver methods are:

- `WordSpace.forwardSymbols(data, subspace)` - runs `compose` on the SymbolicSpace subspace and returns `(composed, svo)`.
- `WordSpace.reverseSymbols(data, subspace)` - runs `decompose`.
- `WordSpace.predict_rule()` / `predict_rule_hard()` - stack-oriented shift/reduce helpers.

`SymbolicSpace.forward` calls `WordSpace.forwardSymbols` as part of its composition pipeline; the conceptual and perceptual spaces no longer run their own syntactic layers.

SymbolicSpace

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 + vectors.
4. `WordSpace.forwardSymbols(...)` runs the unified `SyntacticLayer` and returns the composed symbolic tensor.

Reverse Path

1. Extract symbol activation $[B, n_{\text{Symbols}}]$.
2. Exact inverse of the PiLayer recovers $[B, n_{\text{Concepts}}]$.
3. `WordSpace.reverseSymbols(...)` invokes `decompose` to reconstruct the pre-compose tensor from the word record.

Key Properties

- Symbols are **zero-dimensional** – pure activation scalars, not vectors.
- The PiLayer allows $n_{\text{Concepts}} \neq n_{\text{Symbols}}$.
- `composeSyntax()` runs the unified SyntacticLayer and executes the soft superposition over all active S-tier rules.

AssociationLayer (EQUALS Implementation)

The `AssociationLayer` is a cross-symbol associative memory used by the EQUALS rule. Two modes are available:

- **`type="symmetric"`** – Hopfield-like: learns projection A, computes association scores $A^\top A$, softmax-retrieves the associated pattern. Associations are symmetric ($A \equiv B \Leftrightarrow B \equiv A$).
- **`type="hopfield"`** – Modern Hopfield: separate query/key projections, 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.

How Syntax Is Trained

The Intended Differentiable Mechanism

The syntax code is written to be differentiable:

- `SyntacticLayer.forward()` produces soft `rule_probs`
- `compose()` uses soft superposition over all candidate rules
- `_compose_to_target()` is soft in training, argmax in eval
- The Phase 1 `not` pass is deterministic (non-differentiable) but only fires when the top-of-stack mean is negative

A downstream loss can backpropagate into rule probabilities, rule- prediction weights, depth embeddings, and rule-execution parameters (e.g. the Sinkhorn logits for `swap`).

The Actual Losses in `runBatch()`

`BasicModel.runBatch()` computes three losses:

1. output loss
2. optional reconstruction loss
3. 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.

Current Integration Status

Under `MentalModel.xml`:

- syntax is enabled and the unified syntactic layer is instantiated
- syntax rules are predicted, syntax states and word tuples are produced
- `MentalModel.forward()` goes through `SigmaLayer` rather than invoking the syntax pipeline directly
- syntax is active as computation and analysis, but not strongly coupled to the task loss

The syntax system is a real, parameterized, differentiable grammar module – configured by XML rule subsets and implemented with a single `SyntacticLayer` class – but not yet placed on a strong loss path in the stock `MentalModel.xml` loop.

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 (full reconstruction information)

This pairs generative rules with their compositional inverses so that 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 below was authored before the chart-compose implementation landed; Phases A-D have since shipped (see the "Chart-like Compose (typed SVO)" section earlier in this document and the tracker at `doc/plans/2026-04-20-LearnedSVO-integration.md`). Kept here for archaeology and as a reference for any future extension of the same approach.

Learned SVO Role Identification from the Grammar

This section specifies what it would take to drive `TruthLayer.universality` from grammar-derived Subject/Verb/Object roles rather than the positional heuristic previously in use.

`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 is where those three tensors come from.

Current state (at the time of writing)

SVO was produced by a positional tap inside `SyntacticLayer._compose_vector` in `bin/Language.py`: the first three active leaf positions of a batch row were labelled subject/verb/object when every row had at least three active leaves. For the canonical three-token transitive corpus in `test/test_universality.py` (`"the teacher helped the student"` without determiners) this produced the right roles. It was wrong for realistic inputs: determiners shift the roles, adjectives insert modifiers, word-order variation breaks it entirely, and embedded clauses are invisible.

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

- `SyntacticLayer.last_svo: Optional[Tuple[Tensor, Tensor, Tensor]]`
- `SyntacticLayer.lifting_layer: LiftingLayer` (instantiated in `init_lifting`, called from `WordSpace._build_syntactic_layer`)
- `MentalModel.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.

Target grammar shape

The binary grammar should be able to express:

```
S  -> S VO
VO -> V O
V  -> v(V)      -- terminal category verb
O  -> n(O)      -- terminal category noun
```

so that a parse of a three-content-word sentence produces a derivation tree whose outer rule is `S -> S VO`, and whose `VO` child comes from `VO -> V O`. The subject vector is arg-0 of the outer rule; the verb vector is arg-0 of the inner rule; the object vector is arg-1 of the inner rule.

Longer inputs extend naturally:

```
S   -> NP VP
NP  -> DET N | N
VP  -> V NP
NP' -> DET N | ADJ N | DET ADJ N
```

and an SVO walker reads (subject = arg-0 of S, verb = head of VP, object = head of VP's NP).

Phase A - Grammar metadata (low risk, ~60 LoC)

A.1 Extend `RuleDef`

```
RuleDef = namedtuple(
    'RuleDef',
    ['tier', 'canonical', 'arity', 'method_name', 'lhs', 'rhs_symbols'],
)
```

- `lhs`: nonterminal name this rule reduces to (`'S'`, `'VO'`, `'NP'`, …).
- `rhs_symbols`: tuple of nonterminal / terminal category names for each RHS slot (e.g. `('V', 'O')` for `VO -> V O`).

Existing rules default `lhs='S'`, `rhs_symbols=('S', ...)` for backwards compatibility with the current single-tier S grammar.

A.2 Multi-LHS `configure()` `Grammar.configure()` hardcodes `for lhs in ('S',)` (see `bin/Language.py` around the `configure` method). Replace with iteration over every key in the XML `<grammar>` block, so each key becomes a distinct nonterminal with its own rule list.

A.3 RHS parser changes `_parse_rule` currently parses function-call syntax (`lift(S, S)`, `not(S)`). Add a bare-symbol-sequence form for typed rules:

```
<S>S VO</S>           -- rhs_symbols=('S', 'VO'), method='merge'
<VO>V O</VO>          -- rhs_symbols=('V', 'O'), method='merge'
```

Branch on whether the RHS contains `( / )`. The function-call form stays for existing ops; the bare-symbol form is the new typed form.

A.4 XML/XSD schema `MentalModel.xml` and its XSD must allow non-S nonterminals as child elements of `<grammar>`. Add each new nonterminal tag as an optional element in the XSD. No migration needed for existing configs because `<S>...</S>` stays valid.

Phase B - Compose restructure (high risk, ~120 LoC)

B.1 Problem: left-associative cascade can't produce `(S, (V, O))` `_compose_vector` in `bin/Language.py` is a strict left-associative cascade:

```
composed = leaf[0]
for d:
    composed = rule_d(composed, leaf[d+1])
```

This bracketing is `(((leaf0, leaf1), leaf2), ...)`. To realise `S -> S VO` where `VO` is itself `V O`, we need `(leaf1, leaf2)` to merge first into VO, then `(leaf0, VO)` into S. Left-associative cascade cannot express this. `_compose_to_target` has the same limitation - it pairs (0,1), (2,3), etc. leftmost-first.

B.2 Chart-like pair selection Replace the cascade with per-step pair selection:

- At each composition step the rule head picks, differentiably, WHICH two adjacent leaves to merge plus WHICH rule to apply.
- A pair-scorer MLP takes the current hidden state plus a pair context `(left, right)` and outputs a softmax over `(N-1)` adjacent pairs.
- The selected pair is merged; the resulting slot replaces the pair in the live-leaf array. Total steps: `N-1`.
- Maintain a per-batch live-leaf count and an alive-mask `[B, N_max]`.

B.3 Risks

- O(N^2) per depth step at train time (soft pair mixture over pairs x rules) vs O(1) in the current code. Acceptable for short inputs (N <= 16); may need chunking for longer.
- Hard argmax at eval, soft at train; convergence needs care.
- `decompose()` needs a matching inverse that reads the pair-selection trace.
- `subspace.add_word()` currently records a single position per rule firing. Pair-selection records `(pos_left, pos_right)` and a merged-slot position; `WordEncoding.encode(...)` already has `leaf1/leaf2/leaf3` fields for this.

B.4 Why a cheaper hybrid doesn't work One might keep the left-assoc cascade and let the rule head pick rules whose LHS is only reachable after the inner merge has happened. For `N V N`:

- Step 1 wants `V N -> VO` (inner first).
- Step 2 wants `N VO -> S` (outer second).

The left-assoc cascade forces step 1 to merge `leaf[0]` with `leaf[1]` (i.e. `N V` first), which is `N V` - not a rule in the target grammar. No rule fires, compose stalls. Left-assoc is unsalvageable for standard SVO. Phase B is required.

Phase C - Category tensor alongside activation (~60 LoC)

C.1 Per-slot category Augment the data tensor `[B, N, D]` with a category tensor `category: [B, N]` where `category[b, n]` is the nonterminal / terminal category ID at that slot (or `-1` for padding). This rides alongside activation through compose.

C.2 Initialisation from POS Two options for seeding category at input time:

- **(a) External tagger.** Run an external POS tagger at data-prep time, map tags to our lexical categories (`N`, `V`, `ADJ`, `ADV`, `DET`, …), store category indices alongside token indices. Clean; requires plumbing through `InputSpace` and pulling a tagger dependency back in.
- **(b) Codebook-derived POS.** K-means on the `WordVectors` codebook to produce K lexical clusters; at runtime, look up each leaf's cluster as its category. Cheaper, co-trained, no external dependency. Quality depends on embedding geometry.

(b) is now the preferred path — NLTK has been removed from the project.

C.3 Rule-compatibility masking The rule head emits `rule_probs: [B, max_depth, num_rules]`. Extend to a compatibility mask:

- For each candidate rule `r`, its `rhs_symbols` specifies the nonterminals the RHS must hold.
- At each merge step, compute `compat[b, r] = 1` iff the categories of the two candidate children match `r.rhs_symbols`, else `0`.
- Multiply into `rule_probs`, then renormalise. Incompatible rules drop out.
- On merge, set the merged slot's category to `r.lhs`.

C.4 Gradient / training dynamics A hard compat mask kills gradient through incompatible rules - semantically correct (those rules cannot fire) but brittle early in training if category assignment is wrong. Mitigations:

- Small epsilon on incompatible rules early; anneal to zero over training.
- Gumbel-softmax on the merged-slot category assignment to keep gradients flowing through category propagation; anneal temperature high-to-low.
- A designated always-compatible fallback rule (e.g. `chunk`) so compose never stalls on unknown pairs.

Phase D - SVO extraction hook (~30 LoC)

Once C is working, SVO falls out of the derivation trace.

D.1 Record the trace In `_compose_vector`, each rule firing produces a tuple `(rule_id, left_slot, right_slot, merged_slot)`. Accumulate per-batch into `self._derivation_trace: List[List[Tuple[...]]]` over depth.

D.2 Walk the trace to SVO After compose completes, for each batch row:

1. Find the outermost `S -> S VO` firing (argmax rule at the final step, constrained to `S`-LHS rules).
2. Read its left arg as subject (slot pointing to an `S`-category leaf or subtree - the canonical S subject is a noun phrase whose head is the subject noun).
3. Find the `VO -> V O` firing that produced the right arg of step 1.
4. Read its left arg as verb, its right arg as object.

Each of S/V/O is the concept vector at the respective slot in `data` (or the aggregate of the subtree slots if we want phrase heads rather than single-word heads).

D.3 Batch-level variability Per-batch traces are natural (rule firings are already per-batch). SVO extraction runs per batch and produces a per-batch mask of "has SVO"; rows without the canonical shape leave their `last_svo` entry as None. The `MentalModel` hook already handles a global `None`; extending to a batch mask means `truth_layer.universality(s, v, o, ..., mask=...)` filters out rows without valid SVO before computing luminosity deltas.

Phase E - Training dynamics

E.1 Soft-mixture / hard-gate interaction Phase B's soft pair mixture and Phase C's hard category gate interact:

- Rule output is a soft mixture over compatible rules.
- Merge output is a soft mixture over pairs.
- Merged-slot category is a categorical argmax of rule LHS (hard).

If hard category assignment is wrong, subsequent compat masks are wrong. Gumbel-softmax on the category assignment lets gradients backpropagate through the "wrong" choice and correct it.

E.2 Curriculum

- Start with short canonical inputs (3-content-word SVO, matching the test corpus) where the correct parse is unambiguous.
- Extend to 5-token `DET N V DET N` once short inputs produce stable SVO.
- Add modifiers (`ADJ`, `ADV`) and embedded clauses last.

E.3 Universality loss weighting `architecture.UniversalityWeight` currently defaults to 0.1. Once SVO is learned rather than heuristic, raise toward 0.3-0.5 - the Golden Rule signal is only stable when SVO extraction is reliable, and only then should it dominate loss.

Phase F - Test suite

F.1 New unit tests

- `test_ruledef_typed.py`: `RuleDef` stores `lhs` / `rhs_symbols`; grammar exposes them; XML round-trip preserves them.
- `test_compose_chart.py`: chart-like compose reduces `N` leaves to `1` in `N-1` steps, preserves tensor dims, handles alive-mask.
- `test_category_propagation.py`: merging `(N, V)` under `VP -> V N` yields a slot with category `VP`; incompatible pairs zero the rule prob.

F.2 Update `test/test_universality.py` `_get_svo_and_luminosity` currently returns `(None, None)` with a comment noting that the C-tier ternary-lift path is gone. Once Phase D is in place, replace it with a derivation-trace walk that reads SVO from the model and passes through `truth_layer.universality(...)`. The `xfail` markers stay: untrained models will not score `kind > unkind`.

Cost estimate

Phase	LoC	Risk	Depends on
A: Grammar metadata	~60	low	-
B: Chart-like compose	~120	**high**	A
C: Category tensor	~60	medium	A, B
D: SVO hook	~30	low	C
E: Training dynamics	tuning	medium	D
F: Tests	~80	low	A, B, C, D
Total	~350		

Phase B was the load-bearing risk. Allow at least a week for the compose restructure; everything else layers cleanly once B is stable.

Interim: positional tap (the pre-chart stopgap)

Until the learned approach landed, `SyntacticLayer._compose_vector` used a positional heuristic: any batch row with at least three active leaves labels positions 0/1/2 as subject/verb/object. This fired for the `test_universality.py` canonical corpus and silently skipped longer inputs (`last_svo` stayed `None`, universality stayed `None`, the loss term contributed zero).

Known limits of the positional tap:

- Wrong SVO for five-token sentences with determiners (picks the determiner as "subject").
- No role fidelity for non-canonical word order.
- No support for embedded clauses (only the top-level SVO is visible).

All three are addressed by Phases A-D.

Appendix - hooks that Phase D builds on (already wired)

- `SyntacticLayer.last_svo: Optional[Tuple[Tensor, Tensor, Tensor]]` (set in `compose`, populated in `_compose_vector`).
- `SyntacticLayer.lifting_layer: LiftingLayer` (created in `init_lifting`, called from `WordSpace._build_syntacti`
- `MentalModel.forward` reads both, invokes `truth_layer.universality(s, v, o, lifting_layer, symbolicSpace)`, and stores `self._universality_score`.
- `truth_modulated_loss(universality_score=..., universality_weight=...)` integrates the score into the training loss.

These four interfaces stay stable when learned SVO replaces positional SVO. The only change is how `last_svo` is produced.

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.

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 or `passThrough`, 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	τ: part score above this is "full part"
`disjointThreshold`	0.1	ϵ: 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.

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 SymbolicSpace where `monotonic=True`) and **bitonic** (sign-aware, the default). The monotonic forms are listed here; the bitonic forms (RadMin, RadMax) are in Section 7 Radial Operators.

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

Negation (affirming)

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

Non (non-affirming)

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

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 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; ± 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):

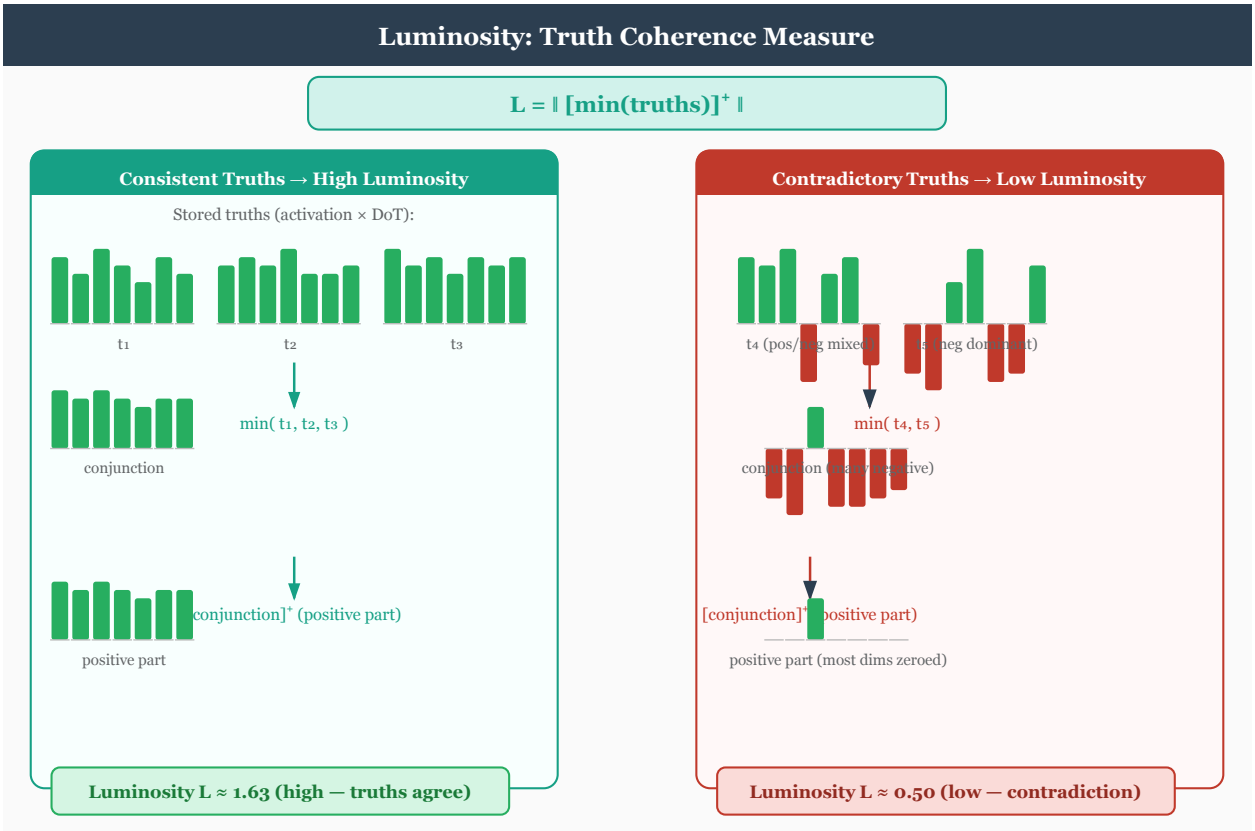
Input	Output
+1	0
0	0
-1	0

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 `MentalModel._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

`MentalModel.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

`MentalModel.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 Quadrants: useButterflies × useGrammar

Two orthogonal flags – `<useButterflies>` (under `<architecture>`) and `<useGrammar>` (under `<WordSpace>`) – select among three valid Sigma-Pi architectures. Butterflies and grammar are mutually exclusive: butterfly permutations fight the constituency structure that grammar composition requires, so the (true, true) quadrant is rejected at load time.

	useGrammar=false	**useGrammar=true**
useButterflies=false	Flat shared sigma (default)	Grammar-directed composition
useButterflies=true	Pairwise butterfly mixing	[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=false)

ButterflyStage wrappers around per-level Sigma and Pi layers 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 ButterflyStage (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=true)

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
`<useGrammar>`	`<WordSpace>`	false	Grammar-directed composition with progressive bottlene
`truthMinMagnitude`	`<SymbolicSpace>`	0.3	Minimum activation norm for truth storage
`truthMinNovelty`	`<SymbolicSpace>`	0.5	Minimum novelty for truth storage
`truthMaxInconsistency`	`<SymbolicSpace>`	0.3	Maximum inconsistency for truth storage

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;

When `thought_free` mode is active, the grammar already enforces the one-pointedness that `Contiguous()` characterizes (see Language.md Section Thought-Free Mode).

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
    $\rightarrow$ Pass 1: stream documents, lex + parse, count words, build vocabulary
    $\rightarrow$ Pass 2: stream documents again, train SBOW per sentence, discard examples
    $\rightarrow$ 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

Variable	Default	Description
`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 $\times$ 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
`MLM`	Zero word i	No	Embedding of word i	Bidirectional (BERT-style)
`ARLM`	Zero word i	Yes (j > i zeroed)	Embedding of word i	Autoregressive (GPT-style)
`ARUS`	Same as ARLM	Yes	Zero vector	Unsupervised (loss suppressed)
`RARLM`	Zero word i	Reverse (j < pos zeroed)	Embedding of word i	Right-to-left autoregressive

`<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
`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 λ 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 λ 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 (`ARLM` / `ARUS` / `ARIR`):

1. `MentalModel.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`:
   - `ARLM`: 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
`ARLM`	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 posit
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 α 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 α	Gradient Energy Sensor	Measures loss-surface curvature to set exploration level

The key insight: α **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_{eff}, while W stores the learned structure that increasingly shapes that local distribution as α 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 α:

$$\text{bias} = \alpha, \qquad \text{var} = 1 - \alpha$$

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 α

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 ε 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 α.

---

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 α)
Low $\hat{v}_t$	Loss surface is flat	Exploit learned weights (high α)

3.4 The Alpha Update Rule

$$\alpha_t = \frac{1}{1 + \tau \cdot \sqrt{\hat{v}_t}}$$

where τ (`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
- τ controls the transition speed: large τ means more exploration at equivalent gradient energy

After computing α_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 κ:

- `sigma=1`: encourage exploration (low κ, responsive to gradient variance)
- `sigma=0`: suppress exploration (high κ, var ≈ 0)

---

4. Comparison: Adam vs. Gradient Energy Sensor

Property	Adam (for W)	Gradient Energy Sensor (for α)
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 ε 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 α. 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 α_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 τ (`global_temp`) scales how responsive α is to gradient energy:

$$\alpha = \frac{1}{1 + \tau \cdot \sqrt{\hat{v}}}$$

τ	Effect
$\tau = 0$	$\alpha = 1$ always — pure exploitation, no exploration
$\tau \ll 1$	Aggressive exploitation — quickly converges to $\alpha \approx 1$
$\tau = 1$	Balanced — α tracks gradient energy on a natural scale
$\tau \gg 1$	Conservative — maintains high exploration even with moderate gradients

In practice, τ 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 α upward
- As W learns useful structure, gradient energy decreases and α 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 ε in W_{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_{eff} is materialised as a dense matrix and `pinv(W_eff)` is used for the reverse path.

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_{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 α 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 $\cdot$ noise_raw_L)
U_eff = I + strict_upper(raw_U + t $\cdot$ noise_raw_U)
d_eff = b $\cdot$ d_effective + t $\cdot$ 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 $\cdot$ d_clamped + t $\cdot$ 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 μ and σ 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 μ and σ 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 α:

$$err = (y - \hat{y})^2$$

$$\triangle_W = \alpha(y - \hat{y})\nabla_F x$$

$$err = (y - \hat{y})^2$$

$$e = y - \hat{y}$$

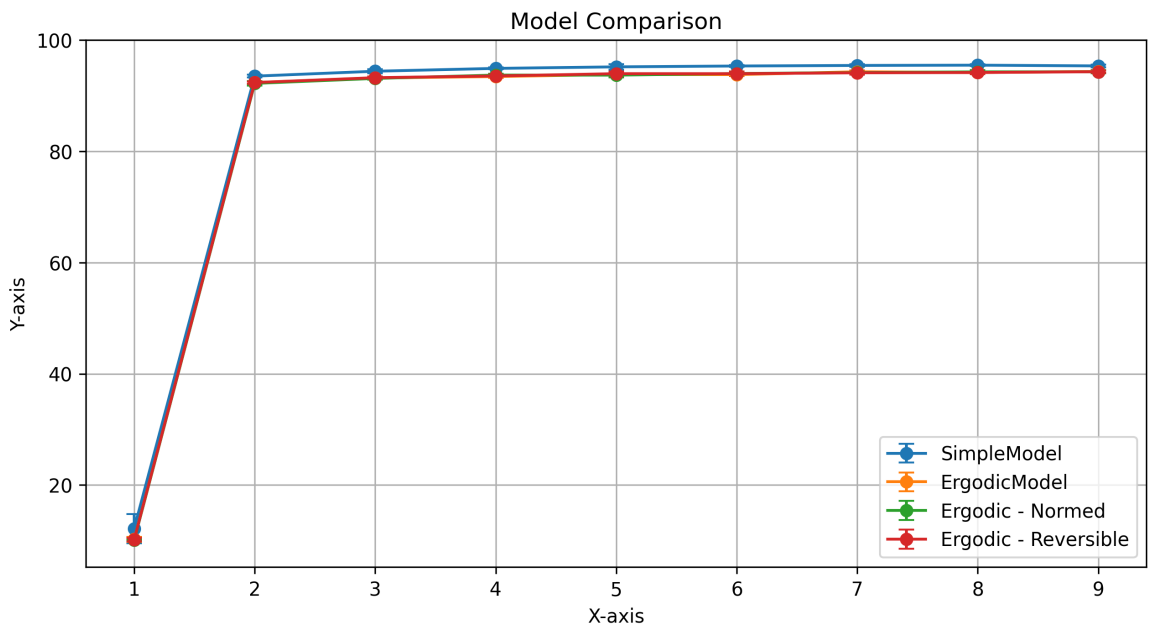
$$c = \alpha\hat{y}^2 + (1 - \alpha)$$

$$\triangle_W = (ce - \alpha\hat{y}e^2)x$$

This equation demonstrates that the network is simultaneously trying to minimize error and "incorrect certainty".

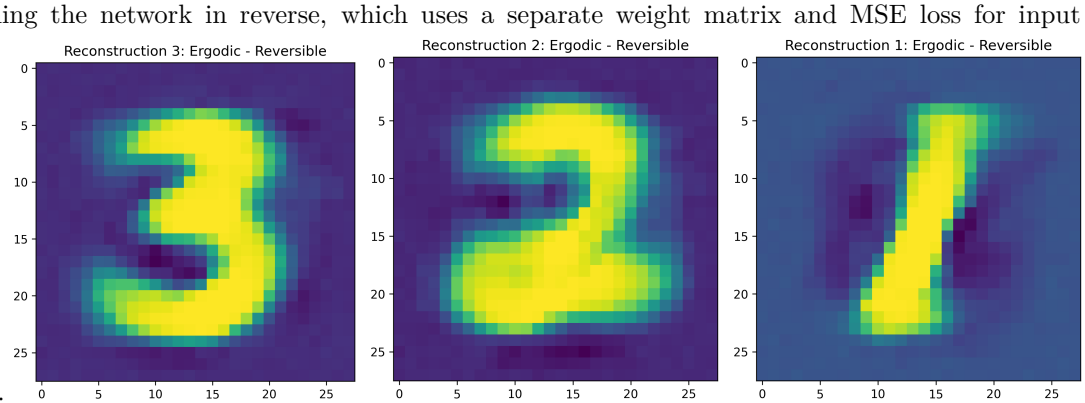
Results

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, MLM, ARLM, ARUS, RARLM. 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 λ 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 λ of the embedding co-occurrence loss in JOINT mode. Higher values bi

---

MentalModel 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 ButterflyStage (N-hal
`monotonic`	boolean	`false`	`<architecture>`	When true, invertible SigmaLayers use W>=0 (NonNegati
`useGrammar`	boolean	`false`	`<WordSpace>`	Enable grammar-directed per-level composition (progressive

```
<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>
```

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). Sh
`nDim`	int	`1`	Dimensionality of each perceptual vector. Set on TheObjectEncoding; not passed to the
`nVectors`	int	`0`	Codebook size (total vectors in the space). When > 0, enables vector quantization with
`invertible`	bool	`false`	Use a single invertible Pi layer instead of separate forward/reverse layers. When `true`, o
`hasAttention`	bool	`true`	Enable attention mechanism in this space.
`passThrough`	bool	`false`	Skip perceptual processing entirely; pass input through unchanged.

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, `nOutputSymbol`
`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.
`passThrough`	bool	`false`	Pass concepts through as symbols unchanged. Typically `true` for simple models.
`codebook`	bool	`false`	Enable codebook quantization. When `true`, the forward path produces a one-hot activ

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>
    <passThrough>true</passThrough>
  </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>ARLM</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