Self-Grading Networks and Living Patterns

K. Fowler & Claude

February 2026

Synthetic gradients, neural cellular automata, and WebGL deployment

Abstract

This report analyzes the design, implementation, and deployment of two neural architectures that replace global learning signals with locally learned approximations. In *Decoupled Neural Interfaces* (DNI), hidden layers learn to predict their own gradients rather than waiting for backpropagation. In *Neural Cellular Automata* (NCA), a single learned update rule applied uniformly across a grid produces emergent, self-repairing visual patterns. Both systems were built on an M2 Max using PyTorch's MPS backend, trained through architecture search, and shipped to the browser as a real-time WebGL2 application. The report focuses on failure modes, stabilizing constraints, and the engineering decisions that determined whether each system trained or diverged.

Contents

Abstract 1

1 Introduction 3

1.1 Hardware Constraints 3

2 Phase 1: Decoupled Neural Interfaces 3

2.1 Mechanism 3

2.2 Implementation 4

2.3 Stability Conditions 4

2.4 Results 5

3 Phase 2: Neural Cellular Automata 5

3.1 From Layers to Cells 5

3.2 Architecture 6

3.3 Training 7

3.4 Architecture Search 7

3.5 Results 8

4 WebGL2 Deployment 8

4.1 Texture Packing 9

4.2 Weight Export and Column Permutation 9

4.3 Boundary-Condition Drift 9

4.4 Multi-Architecture Shader Generation 9

5 Failure Taxonomy 10

6 Design Heuristics 10

7 Deployment and Access 10

8 Conclusion 11

A Model Catalog 13

A.1 Spiral 13

A.2 Voronoi 13

A.3 Checkerboard 13

A.4 Braid 13

A.5 Circle 14

A.6 Mandelbrot 14

A.7 Topo 14

A.8 Cross 14

A.9 Rings 15

Introduction

Backpropagation imposes global synchronization across layers: the gradient at layer l depends on every layer from $l{+}1$ to L, forcing every layer to wait for downstream computation. This design makes training an inherently sequential, tightly coupled process.

Two architectures eliminate this dependency along different axes. *Decoupled Neural Interfaces* [4] replace the true gradient at each layer with a locally learned prediction. *Neural Cellular Automata* [7] replace a centrally specified target pattern with a local update rule that produces the pattern through emergent computation. One decouples layers in a deep network; the other decouples cells in a spatial field.

This report examines a single hypothesis: **global coordination in learning systems can often be replaced by locally trained predictors.** DNI tests this hypothesis across depth; NCA tests it across space.

Both architectures were implemented on an Apple M2 Max, trained through hyperparameter search, and deployed as a real-time WebGL2 viewer. The work was carried out over approximately two weeks, emphasizing engineering pragmatism over paper reproduction.

	DNI	**NCA**
Decoupling axis	depth (layers)	space (cells)
Local predictor	gradient MLP	update rule CNN
Global signal replaced	backpropagation	target image
Neutral init output	$\hat{g} \approx 0$	$\Delta \approx 0$

Table 1 Structural parallel between DNI and NCA. Both replace a global signal with a locally learned approximation that must be initialized to a neutral output.

Hardware Constraints

All training ran on an Apple M2 Max (32 GB unified memory) using PyTorch 2.10's MPS backend [8]. The MPS backend imposes three constraints that materially affect training design:

- `num_workers=0` for DataLoaders (MPS tensors cannot cross process boundaries).
- `torch.mps.synchronize()` required before timing measurements (MPS operations are asynchronous).
- No CPU-to-GPU transfer during the training hot loop.

Every architectural choice described later follows from these constraints.

Phase 1: Decoupled Neural Interfaces

Mechanism

In standard backpropagation, the gradient arriving at layer l depends on every layer from $l{+}1$ to L. Jaderberg et al. proposed replacing this chain with a local function—a *synthetic gradient module*—that predicts the true gradient given only the current activation [4]. If the prediction is accurate, the layer can update without waiting for downstream computation.

Architecturally, DNI transforms training from a synchronized pipeline into independent modules. Each layer trains itself using locally available information. In distributed settings, this enables pipelined parallelism without the bubble overhead of micro-batching.

Implementation

Each synthetic gradient module is a two-layer MLP (`dim`×2 hidden width) that takes an activation vector and predicts the gradient that backpropagation would deliver. The module attaches via `register_hook` on the activation tensor.

The controlling mechanism is a *blend factor* $\beta \in [0, 1]$:

$$g_{\text{blended}} = (1 - \beta) \cdot g_{\text{true}} + \beta \cdot g_{\text{synthetic}}$$

During the forward pass, the module detaches and clones the activation, predicts a gradient with `torch.no_grad()`, and registers a backward hook that blends this prediction with the true gradient. After each step, the module's own optimizer trains the predictor against the true gradient using MSE loss.

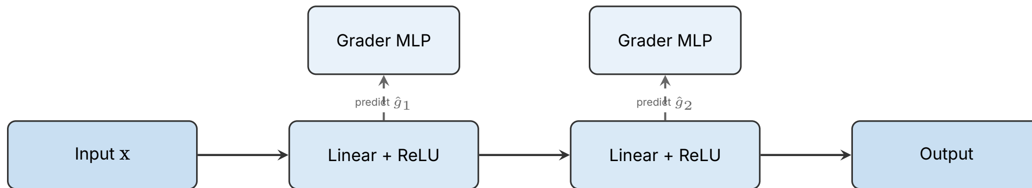


Figure 1 DNI architecture: each hidden layer has a grader MLP that predicts the gradient signal locally.

Stability Conditions

Primary stability condition: synthetic gradients must be introduced gradually.

At initialization, the grader MLPs have zero-initialized output layers, so they predict $\hat{g} \approx 0$. If β is set to 0.8 from epoch 1, the network receives near-zero gradients and diverges within a few iterations. The solution is a linear warm-up:

$$\beta(t) = \min\left(\beta_{\max}, \frac{t}{T_{\text{warm}}}\right)$$

We ramped β from 0 to 0.8 over 5 epochs ($T_{\text{warm}} = 5$). During this period, the graders train on true gradients and learn to approximate them before taking over.

Three measures form the complete stability triad:

1. Warm-up schedule: β ramps linearly from 0 to $\beta_{\max}$ over T_{warm} epochs.
2. Zero-initialized output: The grader MLP's final linear layer has weights and biases set to zero, so the initial prediction is $\hat{g} \approx 0$.
3. Gradient clipping: Predictions are clamped to $[-1, 1]$ element-wise; grader parameter gradients are clipped to norm 1.0.

Removing any of these three measures caused divergence in all test runs. With all three, training converged in every run.

Grader Convergence

The graders converge rapidly. With zero-initialized outputs, the initial prediction error is determined by the magnitude of true gradients (small early in training). Grader MSE drops below 10^{-5} within 20 training steps and remains near zero thereafter. This fast convergence is why the warm-up period can be short: 5 epochs suffice because the graders learn the gradient distribution within the first few hundred batches.

Results

We trained both a standard baseline and a DNI variant on MNIST [6] using a feed-forward network (784→256→128→10).

Model	Test Accuracy	Synthetic Ratio
Baseline (backprop)	98.27%	0%
DNI ($\beta = 0.8$)	96.33%	80%

Table 2 MNIST classification accuracy. DNI accepts a 1.94-point gap for decoupled layer updates.

The 1.94-point gap matches previously reported small-network results [4]. The objective is not accuracy parity—it is architectural decoupling.

Performance Characteristics

On a single device, DNI adds overhead rather than reducing it. Benchmarking a larger 3-layer network (784→512→256→128→10) on MPS:

Model	ms/step	Overhead
Baseline	2.5	—
DNI ($\beta = 0.8$)	8.4	$3.3\times$

Table 3 Wall-clock cost per training step on MPS. DNI is slower on a single device due to grader forward/backward passes. The value is in multi-device pipelining, where layers update asynchronously across GPUs.

The $3.3\times$ overhead comes from three grader MLP forward passes, three grader backward passes, and three grader optimizer steps per training step. On a single device, this is pure cost. The payoff is in distributed settings: each layer can compute its update independently, eliminating pipeline bubbles in model-parallel training. This tradeoff—local overhead for global parallelism—is the core DNI proposition.

Phase 2: Neural Cellular Automata

From Layers to Cells

DNI decouples layers within a network; Neural Cellular Automata decouple cells within a spatial grid. An NCA is a grid of cells, each carrying a state vector of C channels. At every time step, every cell reads its local neighborhood, runs a shared update rule, and adds a delta to its state. From a single seed pixel, complex patterns emerge through purely local computation [7].

This mirrors biological morphogenesis: Turing's reaction-diffusion systems [10] and von Neumann's self-reproducing automata [11] both demonstrate that local rules can produce global order. NCA renders this mechanism differentiable and trainable.

Architecture

Each cell stores 16 channels: 3 for visible RGB, 1 for alpha (alive/dead), and 12 hidden channels. The update pipeline has three stages:

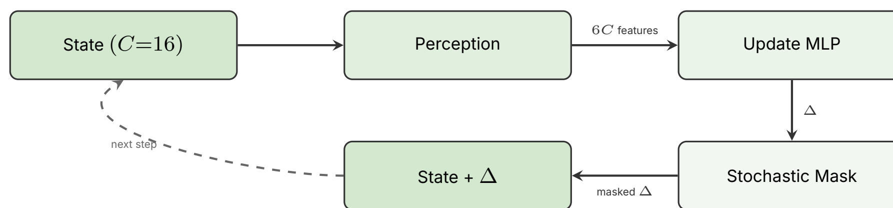


Figure 2 NCA update pipeline: perceive neighborhood, compute delta, apply stochastic mask, accumulate.

Perception

The perception layer reads each cell's local neighborhood. Four variants were evaluated:

- **Sobel (fixed):** Identity + Sobel-x + Sobel-y filters per channel, producing $3C$ features. Zero learnable parameters.
- **Learned:** Trainable 3×3 depthwise convolution, Sobel-initialized, producing $3C$ features. 432 extra parameters for C=16.
- **Dilated (fixed):** Standard Sobel plus a dilation-2 Sobel bank, producing $6C$ features. Effective receptive field: 5×5. Information propagates 2 pixels per step.
- **Learned + Dilated:** Trainable standard bank plus fixed dilated bank, producing $6C$ features.

Update Rule

The update rule is a pointwise MLP implemented as 1×1 convolutions, parameterized by depth, hidden width, residual connections, and an optional firing gate:

- **2-layer:** Perception → hidden → output.
- **3-layer with residual:** Skip connection from hidden layer 1 to hidden layer 2, improving gradient flow.
- **Gated firing:** Extra output channel interpreted as per-cell fire probability via sigmoid. Each cell decides whether to update based on its local state.

The output layer is zero-initialized: the initial delta is exactly zero.

> **Design Principle — Neutral Initialization:** Systems that must produce outputs before training must begin in a known safe state. In DNI, zero-init ensures the grader predicts near-zero gradients before it has learned. In NCA, zero-init ensures the update rule produces zero deltas before training. Both prevent initialization instability by starting from neutral and learning departures from it.

Training

NCA training differs from supervised learning: there is no labeled dataset. The system learns to grow a target pattern from a single seed pixel using a self-supervised objective:

$$\mathcal{L} = \underbrace{\|\hat{y} - y^*\|_2^2}_{\text{pattern}} + \frac{1}{2}\underbrace{\|\hat{y}_{\text{extra}} - y^*\|_2^2}_{\text{stability}} + \frac{1}{10}\underbrace{\text{overflow}(x)}_{\text{boundary}}$$

where $\hat{y}$ is the visible output after N NCA steps from seed, $\hat{y}_{\text{extra}}$ is the output after $N + 32$ more steps, and the overflow term penalizes states near the clamp boundary $|x| > 0.9$.

State Pool

Core training mechanism [7]: maintain a pool of P persistent states rather than always starting from seed. Each step samples a batch from the pool, runs the NCA forward, computes loss, and writes states back. One sample per batch is always reset to seed. This forces the NCA to maintain patterns over time, not grow them once.

We implemented prioritized replay: entries with higher loss are sampled more frequently, concentrating training on the hardest cases.

Training Stabilizers

Six features were incrementally developed:

1. **Gradient checkpointing** [2]: NCA training unrolls 48–96 forward steps. Checkpointing every 16 steps trades compute for $\sim 6\times$ memory reduction.
2. **Multi-scale loss:** MSE at scales $\{1, 2, 4, 8\}$ via average pooling, rewarding both fine detail and coarse structure.
3. **Perceptual loss:** VGG16 feature matching [5, 9] on RGB channels with weight 0.1.
4. **Resolution curriculum:** 64^2 (500 epochs, batch 8) $\rightarrow$ 128^2 (500 epochs, batch 4) $\rightarrow$ 256^2 (1000 epochs, batch 2). The fully convolutional update rule transfers across resolutions.
5. **Prioritized pool replay:** Loss-weighted sampling with softmax normalization.
6. **Truncated BPTT** [12]: Detach every $K{=}16$ steps, giving $3.3\times$ backward speedup with minimal accuracy impact.

Architecture Search

With four perception variants, two hidden widths (64, 128), two or three layers, optional residual connections, and optional gating, the search space contains 32 valid configurations. Optuna [1] with TPE sampling and median pruning searched this space with 20–30 trials per pattern.

Each trial trains a proxy model at 64×64 for 300 epochs ($\sim$2 minutes on MPS). The top architecture per pattern was promoted to full 256×256 training for 2000 epochs.

Observed trend: pattern complexity predicts architectural needs along three axes. First, *spatial frequency*: high-frequency patterns (spiral, cross) require 3-layer residual networks because the update

Pattern	Perception	Hidden	Layers	Features
spiral	learned + dilated	64	3	residual
mandelbrot	learned + dilated	128	2	—
voronoi	dilated	64	2	—
circle	dilated	64	3	residual
cross	learned + dilated	64	3	residual
rings	learned + dilated	128	3	residual
braid	learned + dilated	128	2	gated
topo	learned + dilated	128	2	—
checkerboard	dilated	128	2	gated

Table 4 Architecture per pattern selected by Optuna search and deployed to the web viewer.

rule must compute fine-grained deltas conditioned on subtle local differences; the residual connection preserves these details through depth. Second, *long-range structure*: patterns with large-scale spatial coherence (mandelbrot, topo, rings) converge only with 128-channel hidden layers, which provide the representational capacity to propagate information across many steps. Third, *geometric regularity*: braid and checkerboard—both characterized by strict, repeating geometry—converged only under gated firing, where each cell learns a binary "should I update?" decision. The fixed stochastic mask injects noise that blurs sharp boundaries; the learned gate preserves them.

Simpler patterns (voronoi, circle) converge with shallower, narrower, fixed-perception architectures because their targets have low spatial frequency and no fine geometric constraints.

Results

All nine patterns were trained at 256×256 with the full stabilizer set. Training: 3–4 hours per pattern on the M2 Max.

Pattern	Final Loss	Training Time
spiral	0.017	3.5h
voronoi	0.027	3.0h
checkerboard	0.033	3.5h
braid	0.045	4.0h
circle	0.048	3.0h
mandelbrot	0.050	3.5h
topo	0.069	3.5h
cross	0.070	3.5h
rings	0.146	4.0h

Table 5 Final evaluation loss (pattern + stability + overflow) for all nine NCA patterns, sorted by loss.

WebGL2 Deployment

Training and deployment expose different failure modes. The WebGL2 viewer runs all nine NCA patterns in real time at configurable resolution, with zero server-side computation.

Texture Packing

The 16 NCA channels are packed into four RGBA16F textures, double-buffered for ping-pong updates. The perception step reads from the "read" textures; the update step writes to the "write" textures. After each frame, they swap.

Weight Export and Column Permutation

The hardest deployment issue was the first convolutional layer's weight layout. PyTorch's perception output is ordered as (channel, filter_type)—all filters for channel 0, then channel 1. WebGL reads textures as (texture_group, filter_type, channel_within_group)—all channels packed into texture 0, then texture 1.

The first layer's weight columns must be permuted during export:

```
# PyTorch ordering: [ch0_id, ch0_sx, ch0_sy, ch1_id, ...]
# WebGL ordering:   [ch0_id, ch1_id, ch2_id, ch3_id, ch0_sx, ...]
for tex_group in range(4):
    for filter_type in range(n_filters):
        for ch_in_group in range(4):
            webgl_col = tex_group * n_filters * 4 \
                        + filter_type * 4 + ch_in_group
            pytorch_col = (tex_group * 4 + ch_in_group) \
                          * n_filters + filter_type
            permuted[:, webgl_col] = original[:, pytorch_col]
```

This failure signature: the NCA runs but produces smeared, channel-swapped output. The visual result is almost correct—which makes debugging harder than a clean failure.

Boundary-Condition Drift

Second failure class: representation mismatch at grid boundaries. In PyTorch, `conv2d(padding=1)` zero-pads the input. In WebGL, `CLAMP_TO_EDGE` repeats the boundary pixel. This difference—zero vs. repeat—causes the NCA to behave differently at boundaries. Because NCA patterns propagate information at 1–2 pixels per step, boundary artifacts creep inward and corrupt the entire grid within ~ 32 time steps.

Fix: explicitly sample boundary conditions in the shader and substitute zeros for any sample outside the grid.

> **Design Principle — Boundary Dominance:** Boundary conditions dominate long-horizon dynamics. When porting differentiable systems to GPU shaders, boundary handling is the most common source of divergence between training and inference behavior.

Multi-Architecture Shader Generation

The nine models span four distinct architectures. Rather than separate shaders, a dynamic generator reads model metadata and emits correct GLSL at load time:

- Standard vs. dilated perception (3 vs. 6 filter banks)
- 2-layer vs. 3-layer MLP (with optional residual connections)

- Standard stochastic masking vs. gated firing
- Correct uniform declarations for each weight tensor shape

Adding a new architecture variant requires only updating the generator.

Failure Taxonomy

Four failure classes were encountered across both phases:

1. Initialization instability: Learned components that produce outputs before training must start from a neutral state. Violated in DNI (random grader output) and NCA (random update deltas). Fix: zero-init output layers.
2. Synchronization assumptions: DNI diverges when synthetic gradients are trusted before the grader has learned. The warm-up schedule breaks this assumption gradually rather than abruptly.
3. Representation mismatch: PyTorch and WebGL order tensor dimensions differently. The weight permutation bug produced almost-correct output, delaying diagnosis.
4. Boundary-condition drift: Training and inference environments handle boundary padding differently. Small per-step errors compound over the NCA's long horizon.

Classes 1 and 2 are training failures. Classes 3 and 4 are deployment failures. Both share a common trait: the system runs, produces plausible output, and fails subtly.

Design Heuristics

The following rules generalized across both architectures:

1. Start from neutral. Systems that must produce outputs before training must begin in a known safe state. Zero-init output layers enforce this.
2. Trust must be earned. Local predictors must be trained on real signals before they are trusted. The warm-up schedule embodies this principle.
3. Boundaries dominate horizons. In iterative systems, boundary conditions compound per step. A 1-pixel error at the edge corrupts the interior within N steps for an N-pixel grid.
4. Almost-correct is the hardest failure. Representation mismatches produce output that looks plausible, delaying diagnosis. Binary pass/fail tests catch less than visual inspection.
5. Resolution is a free axis. Fully convolutional architectures amortize search cost: evaluate cheaply at low resolution, train expensively at high resolution, deploy at any resolution.

Deployment and Access

The system is deployed at `neural-petri-dish.pages.dev` as a static Cloudflare Pages site. All computation runs client-side in WebGL2. Users select among nine patterns, adjust resolution (64–512), and interact by drawing or erasing cells. Exported models range from 32 KB to 124 KB.

Source code, training pipeline, architecture search, web viewer, and weight export tooling are included in the project repository.

Conclusion

Both architectures demonstrate a single principle: global learning signals can be replaced with locally learned approximations when those approximations are trained against ground truth and constrained during early learning. DNI replaces backpropagation with local gradient prediction. NCA replaces a centrally specified target with local update rules that produce the target through emergent computation.

The engineering cost is real but tractable. DNI requires warm-up scheduling, zero-init, and gradient clipping. NCA requires pool training, multi-objective losses, and architecture search. WebGL deployment requires exact boundary-condition matching and data layout permutation. None of these are theoretically deep—they are craft problems, solved by understanding the system well enough to anticipate where it will fail [3].

Both systems show that global coordination can be decomposed into locally solvable subproblems whose solutions approximate the global objective. This suggests a general engineering heuristic: whenever a system requires centralized feedback, search for a learnable local proxy.

References

[1] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2623–2631, 2019.

[2] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[3] David Ha and Yujin Tang. Collective intelligence for deep learning: A survey of recent developments. *Collective Intelligence*, 1(1), 2022.

[4] Max Jaderberg, Wojciech Marian Czarnecki, Simon Osindero, Oriol Vinyals, Alex Graves, David Silver, and Koray Kavukcuoglu. Decoupled neural interfaces using synthetic gradients. *Proceedings of the 34th International Conference on Machine Learning*, pages 1627–1635, 2017.

[5] Justin Johnson, Alexandre Alahi, and Li Fei-Fei. Perceptual losses for real-time style transfer and super-resolution. *European Conference on Computer Vision*, pages 694–711, 2016.

[6] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[7] Alexander Mordvintsev, Ettore Randazzo, Eyvind Niklasson, and Michael Levin. Growing neural cellular automata. *Distill*, 5(2):e23, 2020.

[8] PyTorch Contributors. Pytorch mps backend. `https://pytorch.org/docs/stable/notes/mps.html`, 2024.

[9] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[10] Alan M. Turing. The chemical basis of morphogenesis. *Philosophical Transactions of the Royal Society of London. Series B, Biological Sciences*, 237(641):37–72, 1952.

[11] John von Neumann. Theory of self-reproducing automata. 1966. Edited and completed by Arthur W. Burks.

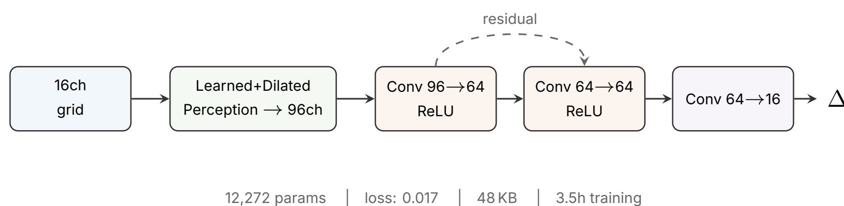
[12] Ronald J. Williams and Jing Peng. An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural Computation*, 2(4):490–501, 1990.

Model Catalog

Each of the nine deployed NCA models is described below with its target pattern, architecture diagram, and training statistics. All models use 16 state channels and produce perception features via Sobel-filtered and (optionally) dilated convolutions, yielding 96 input channels to the update rule. Diagrams read left to right: grid state → perception → hidden layers → output delta.

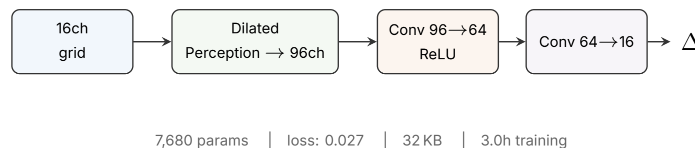
Spiral

A two-arm Archimedean spiral filling the plane, with hue cycling smoothly along the arms. The high spatial frequency of tightly wound arms requires depth (3 layers) and residual connections to compute fine-grained deltas conditioned on subtle local curvature.



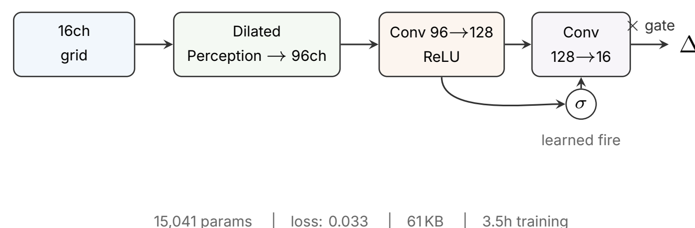
Voronoi

Random Voronoi tessellation with 16 seed cells, colored per-cell with edge-darkened boundaries. Low spatial frequency and no fine geometric constraints allow the simplest architecture: shallow (2 layers), narrow (64 hidden), and fixed dilated perception only.



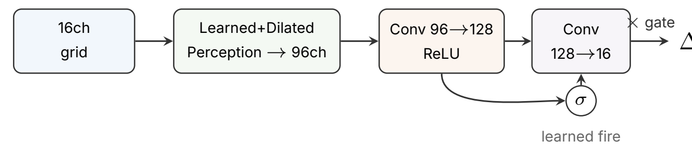
Checkerboard

Alternating magenta and chartreuse squares in an 8×8 grid. The strict repeating geometry requires gated firing: the learned gate preserves sharp cell boundaries that a stochastic mask would blur.



Braid

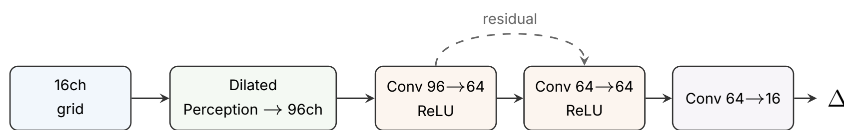
Three sinusoidal strands (red, green, blue) interwoven with depth-ordered occlusion. Like checkerboard, the precise interlocking geometry requires gated firing. The learned perception component captures the strand phase relationships.



15,473 params | loss: 0.045 | 61 KB | 4.0h training

Circle

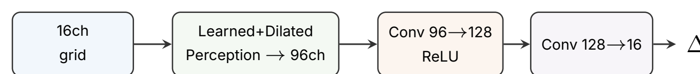
A bright filled circle centered on a dark background. Geometrically simple but requires a 3-layer residual network because the sharp edge must be synthesized precisely—the residual path preserves boundary detail through depth.



11,840 params | loss: 0.048 | 48 KB | 3.0h training

Mandelbrot

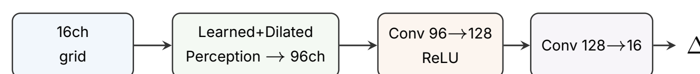
The classic Mandelbrot set boundary with smooth iteration-count coloring. Fractal detail at multiple scales demands wide hidden layers (128 channels) to represent the long-range spatial coherence. Learned perception helps the model detect the fine boundary gradients.



15,344 params | loss: 0.050 | 60 KB | 3.5h training

Topo

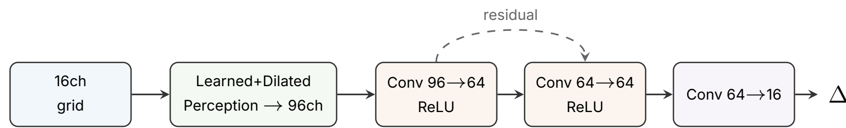
Blue topographic ridges generated by fractal noise with banding, rendered via a GLSL shader target. Like mandelbrot, the multi-scale ridge structure requires 128 hidden channels. The architecture is identical to mandelbrot's—both require wide representations to propagate long-range spatial coherence.



15,344 params | loss: 0.069 | 60 KB | 3.5h training

Cross

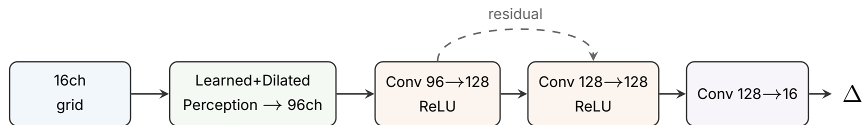
A white cross on dark background. The sharp right-angle geometry at each arm intersection requires 3-layer depth with residual connections, similar to spiral. Learned perception helps detect the orientation-dependent boundary structure.



12,272 params | loss: 0.070 | 48 KB | 3.5h training

Rings

Concentric colored rings with warm-to-cool tonal variation. The largest model: 128 hidden channels across 3 layers with residual connections. Rings require both fine radial precision (depth + residual) and wide representation (128 channels) to maintain coherence across the full radius.



31,856 params | loss: 0.146 | 124 KB | 4.0h training