Beyond Backpropagation:
An Evolutionary Framework for Neural Optimizers

Danil Kutny
danil.kutny@gmail.com

Abstract

Handcrafted learning rules and optimizers such as SGD and Adam remain the backbone of deep learning, despite being applied to models whose inner mechanisms are largely opaque. We propose a framework in which optimization itself is mediated by small auxiliary neural networks, evolved to shape gradient updates. After standard backpropagation, each auxiliary network receives local information and layer-wise statistics (inputs, outputs, gradients) and proposes corrections to the update, which are combined with Adam to form the final parameter step. Unlike gradient-trained meta-optimizers, these controllers are optimzed through evolutionary algorithms, enabling the discovery of non-differentiable update rules. In experiments on MNIST, our evolved optimizers consistently surpassed Adam, achieving peak accuracy of ~91.6% compared to ~89.6% for Adam under the same training regime. Importantly, the evolved update rules generalized to Fashion-MNIST without further evolution, yielding improved average accuracy over the Adam baseline. These results suggest that evolutionary discovery of gradient-shaping controllers can uncover novel and transferable training dynamics, complementing or even surpassing conventional optimization methods.

1. Introduction

Neural networks are trained with handcrafted optimization rules such as SGD [1] or Adam[2]. These rules are simple and interpretable to humans, yet they are applied to models whose internal mechanisms remain complex and often uninterpretable. This mismatch motivates an alternative: instead of fixing the training mechanism by design, we create a framework to evolve learning rules. In other words, black-box models should not be forced to adapt under rigid, human-crafted constraints.

Attempts to move beyond hand-designed optimizers have led to meta-learning [3], where update rules are parameterized or embedded in recurrent structures. However, these approaches remain limited: they are typically built on narrow functional forms and, paradoxically, are often trained using the very algorithms they aim to replace [4]. The result is a constrained search across a vast landscape of possible learning mechanisms.

In this work, we introduce a framework where optimization itself is mediated by small, trainable neural networks. After standard backpropagation computes gradients, auxiliary networks receive both global layer statistics and local variables (inputs, outputs, gradients) and propose corrections

to the gradients and updates. These artificial adjustments are combined with Adam's update to form the final parameter step. Crucially, the auxiliary networks are not trained with gradient descent, but evolved through evolutionary algorithms [5], enabling the discovery of non-intuitive update rules beyond human design. This new two-backward-pass mechanism—backprop followed by evolved gradient shaping—treats the optimizer itself as an open-ended object of search. Our results suggest that such evolved controllers can uncover novel training dynamics, complementing or even surpassing conventional methods.

2. Background and Related Work

Training deep neural networks relies on optimization algorithms that convert gradients into parameter updates. The most widely used optimizers are **hand-crafted rules** such as stochastic gradient descent (SGD) [1], and based on it – Adam [2]. These methods are simple and effective, but they are human-designed, with fixed functional forms that may not capture the full richness of possible learning dynamics.

A natural extension is to move toward **learned optimizers**. Andrychowicz et al. [4] first demonstrated that recurrent neural networks can be trained to output parameter updates, effectively "learning to learn by gradient descent." Wichrowska et al. (2017) [6] scaled this idea to larger models and showed some generalization across tasks

A different tradition arises from **evolutionary algorithms and neuroevolution** . Since the early work on genetic algorithms [5] and NEAT [7] evolutionary methods have been applied to architectures, learning rules, and policies. More recently, Salimans et al. (2017) [5] demonstrated that **evolution strategies** can scale to modern deep networks, while Fernando et al. (2016) [8] explored meta-learning through evolutionary adaptation (the Baldwin effect). Evolution has the advantage of not relying on differentiability, making it attractive for searching over update rules.

A particularly relevant inspiration for this work is the **Variable Shared Meta Learning (VSML)** framework of Kirsch & Schmidhuber (2020/2021) [9]. In VSML, network weights are replaced with small LSTMs, enabling the learning rule itself to generalize across tasks and architectures. The key idea is that tiny neural controllers can be reused flexibly, rather than being tied to a single fixed network structure. This insight motivated our design: in EvoGrad, we also insert small auxiliary neural networks into the training loop, but instead of replacing weights entirely, we let them **shape and adjust gradient updates**. Moreover, we evolve these controllers with population-based algorithms, avoiding the recursion of gradient-based meta-optimizers and ensuring that the controllers can scale across a wide variety of architectures.

3. Algorithm architecture

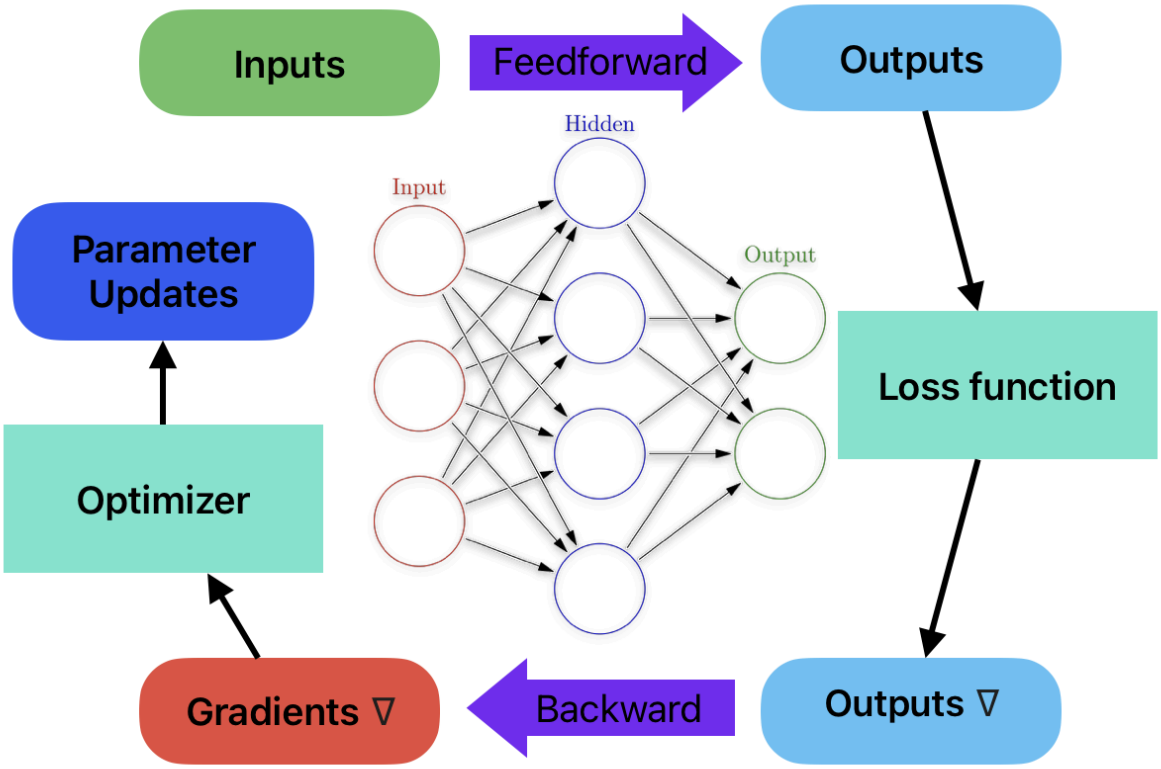


Figure 1. Standard training pipeline. *Forward pass, backward pass, and optimizer step form the classical deep learning workflow*

Figure 1 illustrates the classical deep learning workflow. During the forward pass, inputs are transformed through a network of parameters to produce outputs. A loss function compares outputs with targets. The backward pass then propagates derivatives of the loss through the network, yielding gradients with respect to each parameter [10]. Finally, an optimizer such as SGD or Adam maps these gradients into parameter updates. This three-step cycle (forward, backward, optimizer) constitutes standard training. Augmented Training Pipeline with EvoGrad

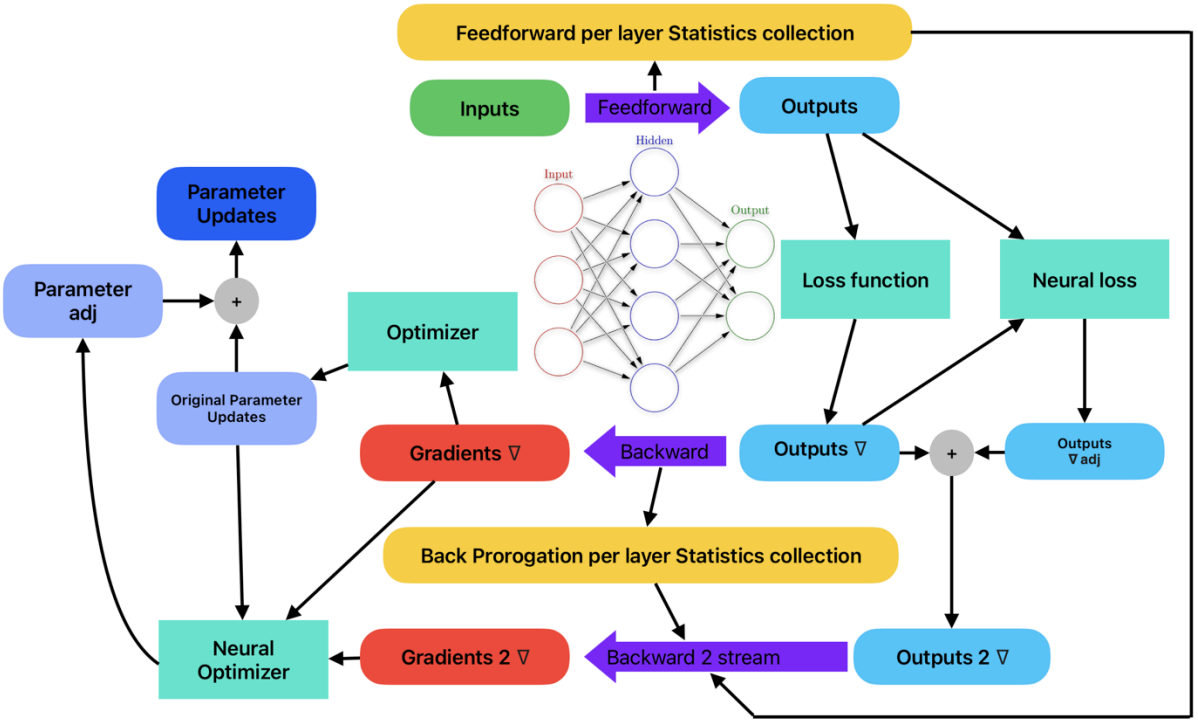


Figure 2. Augmented EvoGrad pipeline. *EvoGrad extends standard training with four components: per-layer statistics collection, Neural Loss, gradient correction networks, and a Neural Optimizer. This creates two losses, two backward streams, and two optimizers.*

Figure 2 shows our augmented framework EvoGrad (evolutionary gradients), which inserts four new mechanisms into the classical pipeline:

1. **Per-layer statistics collection**: during both forward and backward passes, mean, standard deviation, skewness, and kurtosis are calculated from the relevant vectors, such as inputs and outputs. This information about the whole layer is then processed, and features are extracted by a specialized neural network, to be used for gradient update guidance.
2. **Neural Loss** – generates loss signals for the second backpropagation stream.
3. **Neural learning rules** – produce gradient corrections (gradients²), which act as additional parameter updates.
4. **Neural Optimizer** – a stateful neural network (LSTM-based optimizer). It gathers the final information about the original gradient, the gradient adjustment signal, and the optimizer update step.

The process now consists of two losses, two backward streams, and two optimizers. At each step, small neural networks adjust learning signals, building on top of well known pipeline without interference, but can adjust learning dynamics in a non-intuitive and complex way.

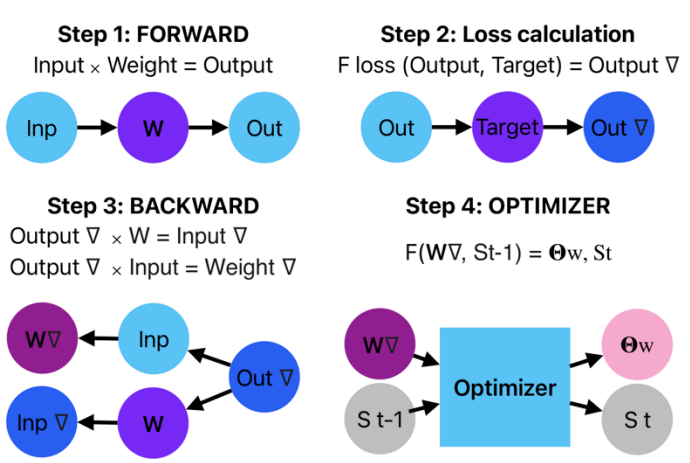
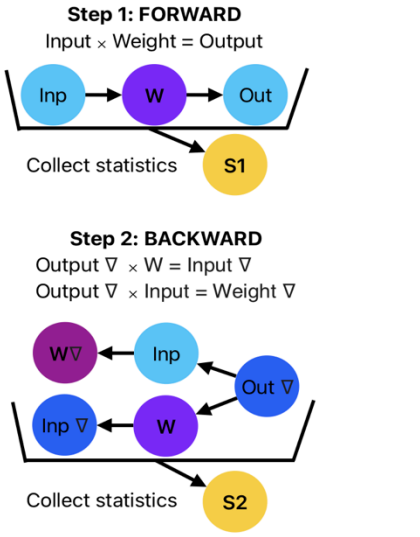


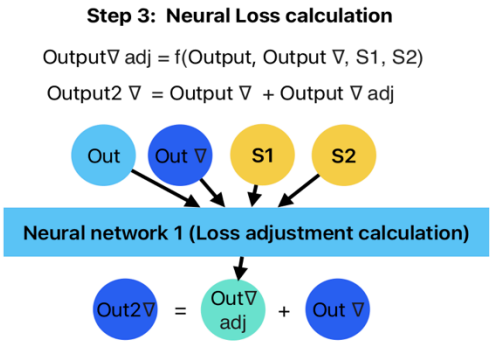
Figure . Matrix Multiplication Layer computation graph. Step-by-step breakdown of a single layer's forward and backward operations, used as the basis for inserting auxiliary networks.



Step-by-step Process

To make the workflow explicit, **Figure 3** contrasts the standard optimizer with our new pipeline, illustrated in **Figure 4**. The forward and backward passes, as well as loss calculation, remain unchanged. However, statistics are collected during these calculations.

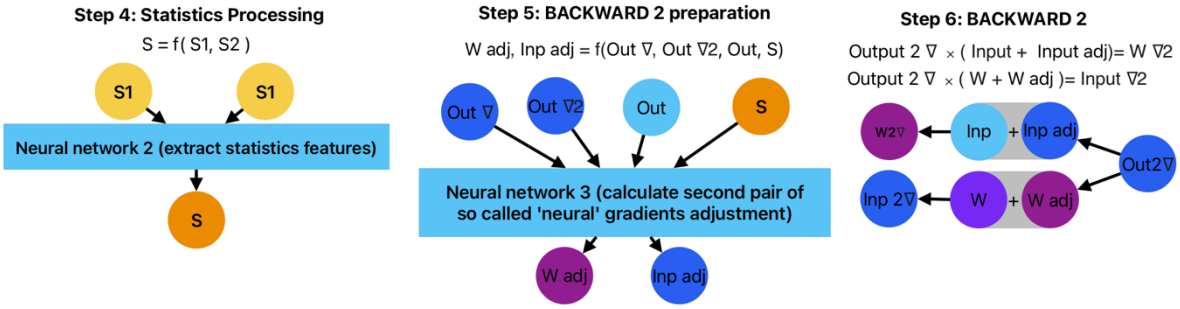
- **S1 and S2** represent mean, standard deviation, skewness, and kurtosis, collected from layer's vectors: inputs, outputs, and gradients.
- This information is then passed to the **Neural Loss function** – a lightweight feedforward neural network, which computes loss adjustments.
- The Neural Loss receives outputs, output gradients with respect to the original loss, as well as the statistical information, and produces an update. This update is used to adjust the output gradients (outputs² ∇). The parameters of the Neural Loss are themselves subject to optimization.



EvoGrad. *Illustration of how statistical features (S1, S2) and Neural Loss outputs are integrated into the training process.*

The second backward pass is illustrated in **Figure 5**. Two more neural networks are introduced at this stage:

1. **Statistical Processing Network (SPN)** – concatenates S1 and S2 tensors and calculates

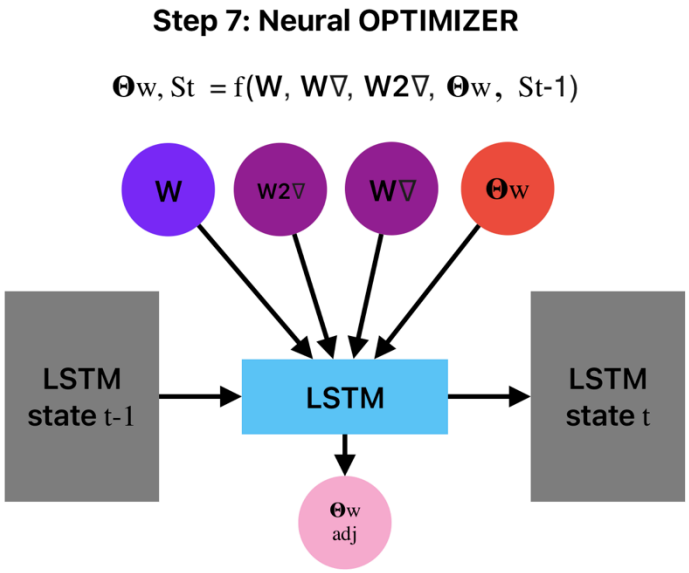


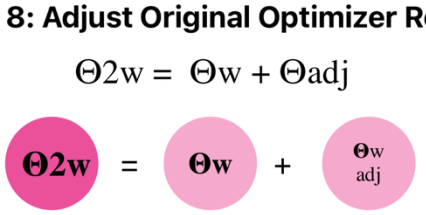
Network (SPN) generate additional gradient signals, producing a second error stream that complements the original backpropagation.

the features required for the next step.

2. **Gradient Processing Network (GPN)** – uses the SPN features, together with other relevant information, to calculate weight and input adjustments.

These adjustments are then used to calculate the second backward error and gradient corrections. This process represents the **second backpropagation stream**. The second backward error is used for further gradient calculations, while the neural gradients represent a second set of gradients for the main neural network. Importantly, this pipeline is independent of network size: SPN and GPN operate at the level of single signals, and can therefore be scaled to neural networks of arbitrary size.





With the help of statistical representations of entire layers, as well as access to already computed gradients, we expect this neural setup to capture important details during training. These networks can then be fine-tuned with evolutionary algorithms to discover better, more nuanced, and more effective learning rules.

To further improve the chances of convergence to a superior training algorithm, we added one more step. **Figure 6** illustrates this final stage – the **Neural Optimizer**. To capture advanced techniques for parameter updates, similar to Adam or RMSProp, we introduce an additional neural layer for optimization.

The Neural Optimizer is built with an LSTM architecture, since recurrent networks naturally represent the sequential process of continuous parameter updates, where each step depends on previous states. The optimizer takes as input Wi, ∇Wi, $\nabla 2Wi$, and Θwi. In this way, it integrates information from both backpropagation streams, as well as the already computed updates Θw, and produces a final adjustment Θw adj to the original update. This allows the Neural Optimizer to discover non-intuitive rules based on both current and past information.$\Theta w\ adj$

The proposed algorithm is **layer-specific** yet highly flexible. Once evolved for training a particular type of layer, it can be applied to layers of different sizes.

3. Evolutionary Optimization Setup

We do not optimize the parameters of controllers (NeuroLoss, SPNs, GPNs, NeuroOptimizer) with gradient descent. Instead, we treat them as evolvable entities and use a population-based evolutionary algorithm to search for effective learning rules [11].

This allows the framework to escape the recursion problem (where a learned optimizer must itself be trained with another optimizer) and explore a broader design space.

Population Encoding

- Each individual in the population represents a complete set of parameters for all controllers in the system.
- Concretely: for every unique layer type (Linear, ReLU, Loss), we store the parameters of its SPN (statistics processing network) and GPN (gradient processing network); globally, we store the weights of the NeuroOptimizer LSTM.
- In code, individuals are structured as nested dictionaries {layer → {SPN, GPN} → [tensor list]} where each tensor is cloned and detached from PyTorch modules.

Initialization

- Population is seeded by instantiating fresh NetCustom models and copying their controller weights.
- This ensures that all individuals begin with random but valid parameterizations.

Fitness Evaluation

Each individual is evaluated by training a *separate task network* (not the controllers themselves) for a fixed number of batches.

- Task network:
  - Randomly sampled feedforward MLP.
  - Hidden layer sizes are randomized from ranges (e.g. [32–1024], [16–512], etc.), with decreasing order to enforce pyramidal shape.
  - Learning rates are randomized in range 0.1 – 0.0001
  - Output dimension fixed to 10 for MNIST dataset.
- Training procedure:
  - Insert the individual's controller parameters into the task network.
  - Train for 2048 batches using both Adam and EvoGrad's dual update rule.
  - Record peak classification accuracy on training batches.
- Control baseline:
  - The same task network is trained with Adam only (no EvoGrad corrections).
  - This yields a control score for calibration.

Selection

- We apply elitism: the top 3 individuals by fitness are carried forward unchanged.
- The rest of the next generation is bred from parents chosen with rank-based sampling:
  - All individuals are sorted by fitness.
  - Sampling weight of individual at rank r is proportional to N−1, where N is population size.
  - Parents are drawn in pairs.

Crossover

- For each parent pair, we generate one offspring.
- Crossover is performed per-parameter tensor:
    - A binary mask (Bernoulli(0.5)) is sampled with the same shape as the tensor.
    - Child parameter = parent1 * mask + parent2 * (1 – mask).
- This ensures that both structure and scale of parameters are preserved.

Mutation

- After crossover, offspring are mutated to introduce variability.
- Mutation operates hierarchically:
    - Step 1: Select a layer with probability proportional to its parameter count.
    - Step 2: Select a sub-network (SPN or GPN) within that layer, weighted by size.
    - Step 3: Select a parameter tensor within that sub-network, again weighted by size.
- Once a tensor is selected:
    - With probability MUTATION_RATE per element, add Gaussian noise with variance sampled from MUTATION_STRENGTH = {1.0, 0.001}.
    - If the mask selects no elements, force-mutate one random position (ensures progress).

This "mask + Gaussian noise" procedure perturbs only a small subset of weights, allowing fine-tuning without destabilizing entire networks.

Evolution Loop

- Each generation consists of:
    1. Sample training batches (MNIST).
    2. Evaluate all individuals' fitness.
    3. Select elites + parents.
    4. Crossover parents into offspring.
    5. Mutate offspring.
    6. Form the next population (elites + offspring).
- We log best and mean fitness per generation, plus the Adam control score.
- Populations are checkpointed every 30 generations.

4. Results

We evaluate EvoGrad in two regimes:

1. Meta-Training (Evolution):
    - Evolutionary search was conducted *only* on the MNIST dataset.
    - The population of optimizers evolved exclusively against MNIST classification tasks, using networks of varying depth (1–3 hidden layers, 32–1024 units).
    - Evolution proceeded for up to one million generations with population size 30, elitism 3, mutation rate 0.02 for efficiency. A population size of 30 was chosen as

the smallest effective size for evolutionary search, balancing computational constraints (Eiben & Smith, 2015)

2. Generalization (Meta-Test):
    - After evolution, the best optimizers were frozen and tested on Fashion-MNIST.
    - Importantly, no evolutionary updates were applied during this stage.
    - This evaluates whether rules discovered on MNIST transfer to a different dataset.

4.2 Results on MNIST (Evolution Domain)

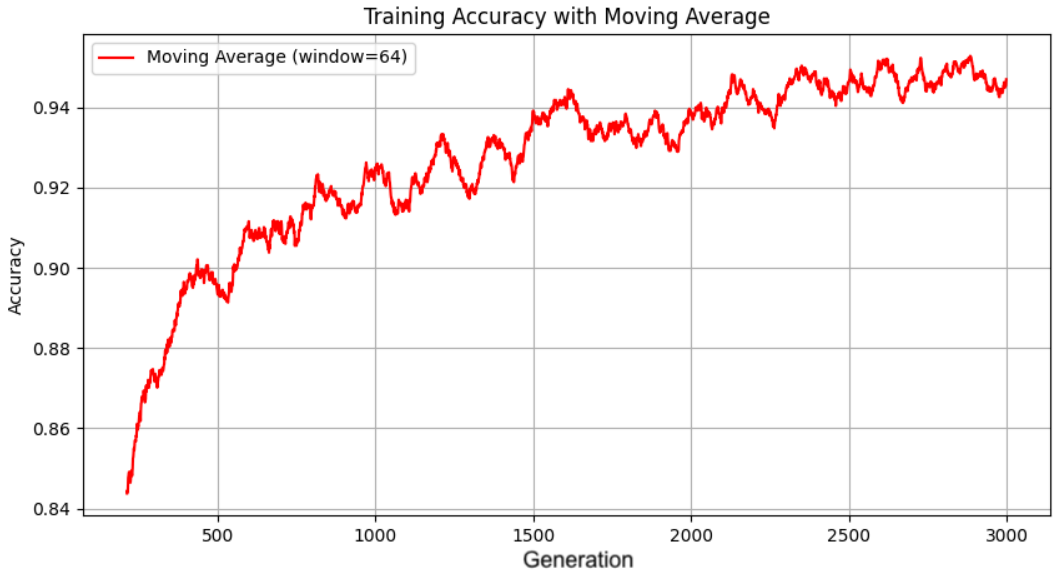
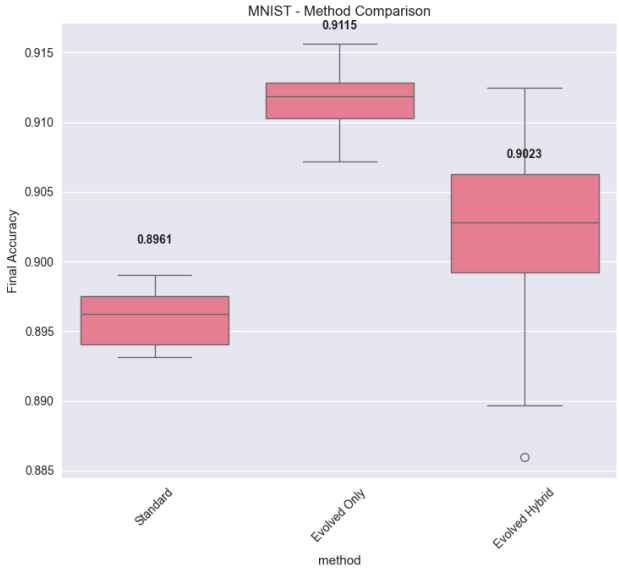
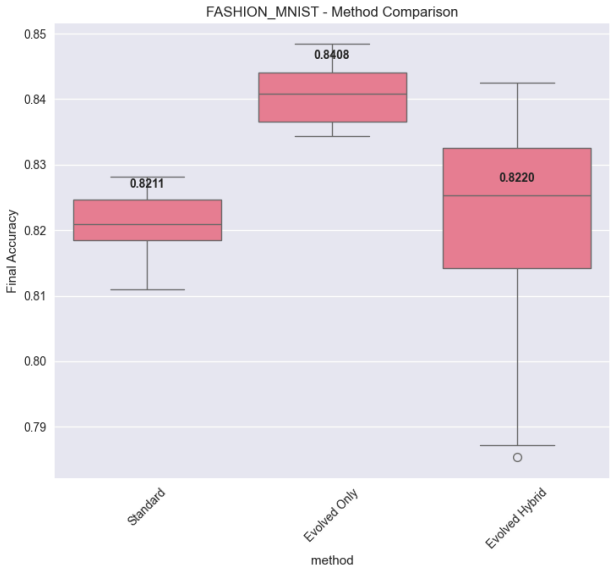


Figure 7. Evolutionary progress on MNIST. *Population accuracy steadily improves over generations*

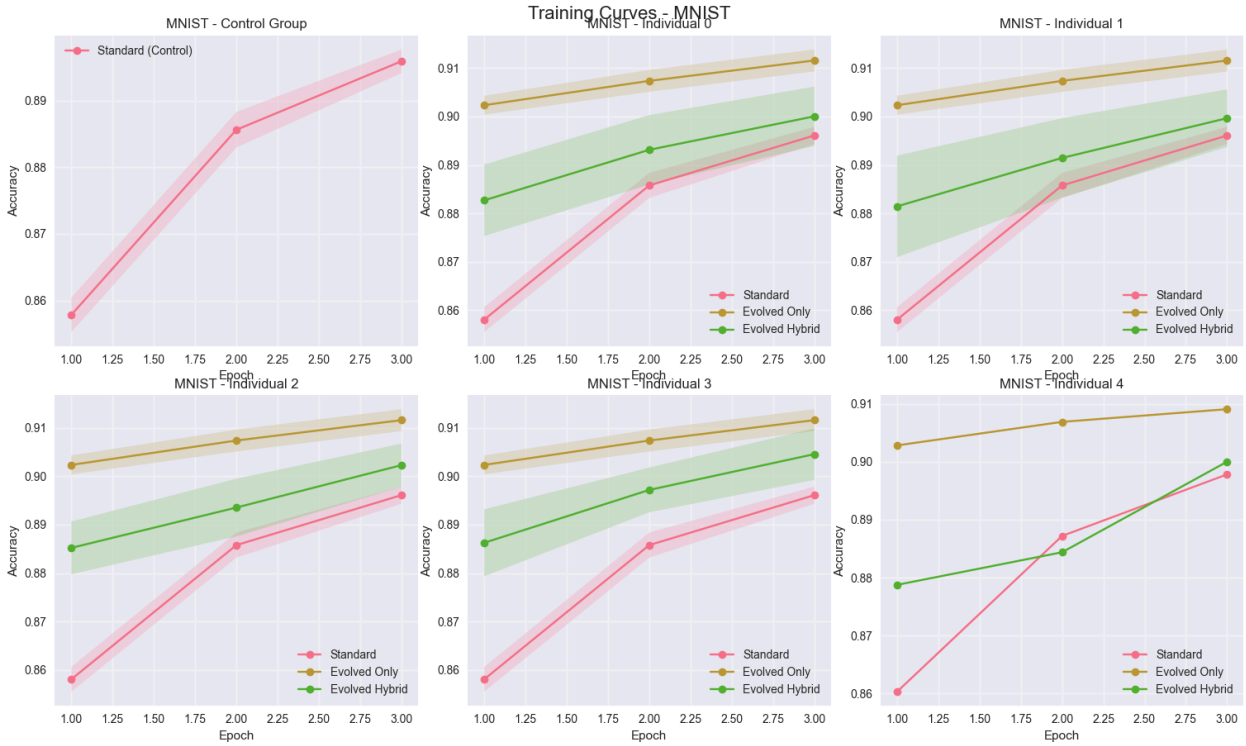
- During evolution, accuracy steadily improved. (figure 7)
- Peak evaluation accuracy: ~0.916 vs. ~0.896 for Adam (with 2000 training steps)
- Mean population accuracy: 0.905–0.913.


MNIST - Method Comparison
Final Accuracy
0.915
0.910
0.905
0.900
0.895
0.890
0.885
0.9115
0.9023
0.8961
Standard
Evolved Only
Evolved Hybrid
method


FASHION_MNIST - Method Comparison
Final Accuracy
0.85
0.84
0.83
0.82
0.81
0.80
0.79
0.8408
0.8220
0.8211
Standard
Evolved Only
Evolved Hybrid
method

4.3 Generalization to Fashion-MNIST (Unseen Domain)

- Despite being evolved exclusively on MNIST, EvoGrad transferred successfully.
- On Fashion-MNIST:
    - Mean population performance was consistently higher.
- This demonstrates that EvoGrad does not merely overfit to MNIST's specific structure, but learns transferable update rules.



5. Discussion

Our experiments demonstrate that evolutionary optimization of auxiliary networks can yield training dynamics that surpass standard optimizers such as Adam, even when evaluated outside the domain of evolution. The framework provides a proof of concept that optimization rules need not be manually designed or restricted to rigid gradient-based meta-learning.

At the same time, several limitations must be acknowledged:

- **Dataset scale:** Current evaluation is restricted to MNIST and Fashion-MNIST. These benchmarks are suitable for fast evolutionary experimentation, but it remains an open question whether the approach scales to more complex datasets and deeper architectures.
- **Computational cost during evolution:** Evolutionary training is resource-intensive, since each candidate optimizer must be evaluated through full training runs. Future work should investigate surrogate evaluation methods, partial training, or distributed search to reduce cost.
- **Computational cost during training:** training a task neural network with evograd mechanism is computationally more demanding. As we used small neural network for each signal output of a layer, and additionally small LSTM is attached to each single parameter, increasing computation cost.

- **Interpretability:** While evolved networks provide effective gradient corrections, their internal mechanisms were not analyzed in this work. Understanding which features and patterns are exploited could yield new theoretical insights into optimization.

By demonstrating that evolved neural optimizers can outperform Adam, EvoGrad opens the door to a new era of flexible, data-driven optimization strategies for deep learning.

6. Conclusion

We introduced **EvoGrad**, an evolutionary framework for neural optimization that augments standard backpropagation with auxiliary neural networks. These controllers—responsible for gradient shaping, loss adjustment, and parameter updates—are trained not by gradient descent but through evolutionary algorithms, enabling the discovery of non-differentiable and non-intuitive update rules.

On MNIST, EvoGrad consistently outperformed Adam under the same training regime, and crucially, the evolved rules generalized to Fashion-MNIST without further evolution. This demonstrates that optimization itself can be treated as an open-ended object of evolutionary search, capable of yielding transferable learning dynamics.

Looking forward, promising directions include scaling EvoGrad to larger datasets, exploring different neural architectures, and analyzing the internal behavior of evolved controllers. Together, these paths may help establish evolved optimizers as a viable complement to human-designed training algorithms.

References

[1] H. Robbins and S. Monro, “A Stochastic Approximation Method,” *Ann. Math. Stat.*, vol. 22, no. 3, pp. 400–407, Sept. 1951, doi: 10.1214/aoms/1177729586.

[2] D. P. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,” Jan. 30, 2017, *arXiv*: arXiv:1412.6980. doi: 10.48550/arXiv.1412.6980.

[3] J. Schmidhuber, “Learning to Control Fast-Weight Memories: An Alternative to Dynamic Recurrent Networks,” *Neural Comput.*, vol. 4, no. 1, pp. 131–139, Jan. 1992, doi: 10.1162/neco.1992.4.1.131.

[4] M. Andrychowicz *et al.*, “Learning to learn by gradient descent by gradient descent,” Nov. 30, 2016, *arXiv*: arXiv:1606.04474. doi: 10.48550/arXiv.1606.04474.

[5] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, “Evolution Strategies as a Scalable Alternative to Reinforcement Learning,” Sept. 07, 2017, *arXiv*: arXiv:1703.03864. doi: 10.48550/arXiv.1703.03864.

[6] O. Wichrowska *et al.*, “Learned Optimizers that Scale and Generalize,” Sept. 07, 2017, *arXiv*: arXiv:1703.04813. doi: 10.48550/arXiv.1703.04813.

[7] K. O. Stanley and R. Miikkulainen, “Evolving Neural Networks through Augmenting Topologies,” *Evol. Comput.*, vol. 10, no. 2, pp. 99–127, June 2002, doi: 10.1162/106365602320169811.

[8] C. T. Fernando *et al.*, “Meta-Learning by the Baldwin Effect,” in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, July 2018, pp. 109–110. doi: 10.1145/3205651.3205763.

[9] L. Kirsch and J. Schmidhuber, “Meta Learning Backpropagation And Improving It,” Mar. 13, 2022, *arXiv*: arXiv:2012.14905. doi: 10.48550/arXiv.2012.14905.

[10] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *Nature*, vol. 323, no. 6088, pp. 533–536, Oct. 1986, doi: 10.1038/323533a0.

[11] A. E. Eiben and J. E. Smith, *Introduction to Evolutionary Computing*. in Natural Computing Series. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015. doi: 10.1007/978-3-662-44874-8.