

Reactive Optimizers

Xingjian Bai, Jia-Hao Liang, Jayden Teoh, Vincent Sitzmann / October 2026

TL;DR We put an agent at the heart of the optimizer. It watches the training dynamics, reacts to them, and rewrites the optimizer’s source code as training goes. When no recipe has been tuned for a task, it matches or beats the best recipe one could pick off the shelf, across image and language tasks. Further, we demonstrate that it spontaneously comes up with sophisticated schedules and intricate strategies to prevent loss spikes and escape overfitting. Try it on your own training run with the [OPAL package](#).

A static optimizer

step 80,000

```
1  $\eta, \beta_1, \beta_2, \epsilon = 2e-4, 0.9, 0.95, 1e-8$ 
2 for  $t = 1 \dots 80K$ :
3    $g \leftarrow \nabla L(\theta)$ 
4    $m \leftarrow \beta_1 \cdot m + (1 - \beta_1) \cdot g$ 
5    $v \leftarrow \beta_2 \cdot v + (1 - \beta_2) \cdot g^2$ 
6    $\hat{m} \leftarrow m / (1 - \beta_1^t)$ 
7    $\hat{v} \leftarrow v / (1 - \beta_2^t)$ 
8    $\theta \leftarrow \theta - \eta \cdot \hat{m} / (\sqrt{\hat{v}} + \epsilon)$ 
```

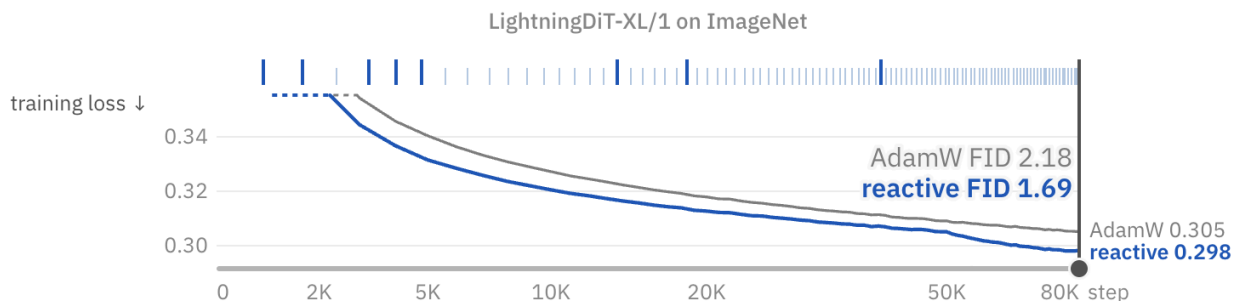
step 80K

done · program unchanged since step 39,201

A reactive optimizer

■ finished step 80,000

```
1 |  $\mu, s_W = 0.95, \text{rms}(\hat{m}/(\sqrt{\hat{v}}+\epsilon))$  at W's switch
2 |  $k_W = \{\text{attn}/\text{MLP } 1.5, \text{adaLN } 1.25\}$ 
3 |  $\eta, \beta_1, \beta_2, \epsilon = 2e-4, 0.9, 0.95, 1e-8$ 
4 | cooldown:  $1-\sqrt{\cdot}$  from 50K to  $0.03\times$  at 76K
5 for  $t = 1 \dots 80K$ :
6    $g \leftarrow \nabla L(\theta)$ 
7 |    $g_c \leftarrow g$  without t-emb layer 1
8 |   if  $\|g_c\| > 3 \cdot \text{EMA}\|g_c\|$ : cut this step's  $\eta$ 
9 |   for  $W$  in attn/MLP, adaLN matrices: # Muon
10 |      $M \leftarrow \mu \cdot M + g$ 
11 |      $W \leftarrow W - \eta \cdot k_W \cdot s_W \cdot \text{orth}(g + \mu \cdot M)$ 
12 |   for  $\theta$  in everything else: # AdamW
13 |      $m \leftarrow \beta_1 \cdot m + (1 - \beta_1) \cdot g$ 
14 |      $v \leftarrow \beta_2 \cdot v + (1 - \beta_2) \cdot g^2$ 
15 |      $\hat{m} \leftarrow m / (1 - \beta_1^t)$ 
16 |      $\hat{v} \leftarrow v / (1 - \beta_2^t)$ 
17 |      $\theta \leftarrow \theta - \eta \cdot \hat{m} / (\sqrt{\hat{v}} + \epsilon)$ 
```



Today, the recipe used to optimize a deep neural network is typically completely determined before training even begins. A great deal of human ingenuity has gone into priors regarding how training should progress: adaptive step sizes [1], preconditioning [2, 3], orthogonalized updates [4, 5], schedules tied to the training step [6, 7], and learning rates scaled with model size [8]. Automated discovery [9, 10, 11, 12] searches across many runs for better optimizers. Whether hand-crafted or automatically discovered, the optimizer then runs unchanged through the end of training, adapting, if at all, according to rules written in advance. This scope leaves the following questions unanswered: How should an optimizer respond to an unexpected loss spike or stall? What if one module's gradients start to oscillate while the others keep improving smoothly? And when a new type of architecture trains poorly with Adam or Muon, is the architecture worse, or has it not yet met an optimizer suited to it?

In this post, we attempt to expand the scope of conventional optimizers and make them reactive to the training itself, leading to an ever-adapting optimizer inside a single training run. This reactive optimizer reasons on the spot and writes the weight update rule for each parameter group. Directly writing optimization code, it has the freedom to implement complicated schedules, respond to training accidents, and customize the optimizer for the single training episode in which it exists. Building on earlier attempts to adapt optimizers during training [12, 13, 14] and to let a model direct its own fine-tuning [15], the reactive optimizer is entirely zero-shot, with nothing learned or retained from repeated runs, and puts the optimizer's entire algorithm up for rewriting.

We start with a practical situation, with tasks from image generation and language modeling, in which a reactive optimizer can do much better than static ones. We then turn to the emergent properties of the reactive optimizer, including sophisticated scheduling, suppressing loss spikes, and escaping overfitting. Lastly, we ask what all this costs and how practical it is at scale. We expect the reactive optimizer to grow only stronger with each generation of agents, to be most useful exactly where no recipe exists yet (for a new architecture or a new dataset), and to be one step toward solving the puzzles of deep learning optimization.

Contents

1. Training without a recipe
2. Updating with spontaneous analysis
3. What is dynamically changing during optimization?
4. Real-world applicability
5. Conclusion

1. Training without a recipe

Suppose a PhD student has a new idea: a model of their own design, or a new dataset on which to train a transformer. They have one GPU for a single day only and a modest allotment of LLM API tokens, allowing for just one serious training run. What should the student do? The obvious course of action would be to pick the most standard training recipe, or one from a recent paper, and cope with the possibility that it might be unsuitable for their scenario, with spiking/stalling loss, etc.

The underlying question is broader: [when bounded by GPU time, researcher time, or wall-clock time, how can a deep learning model be trained to its best performance without knowing the optimal optimization recipe in advance?](#) New architectures, modalities, languages, and scientific data all fall into this setting, where few people can afford a sweep at scale.

A reactive optimizer offers another way to use that budget: it spends LLM tokens in place of GPU/human hours. With optimizers reacting to training incidents and adjusting themselves, much of the burden of recipe tuning can be reduced.

Results in brief

We start from the optimization track (Track 3) of modded-nanogpt [\[16\]](#), a language-model pre-training task (training a 162M GPT on FineWeb) with many recipes tuned specifically for this one setting. From this task, we create two slightly differing natural settings: (1) the same model trained on an alternate language dataset (we select Chinese, Thai, and Kyrgyz), and (2) the original setup with a quarter of the batch size. Neither comes with a recipe tuned for that setting, so we compare the

reactive optimizer with the natural options one would have if they wanted to one-shot the new training: adopting five leaderboard recipes with natural adjustments. In all these new settings, the reactive optimizer outperforms or matches the best baseline (Figure 1).

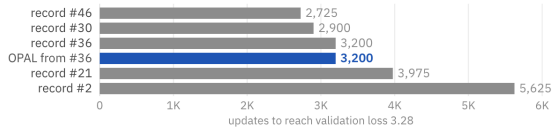
We also apply the reactive optimizer directly to Goldfish [17], which proposes a dataset of 350 languages and trains one small GPT per dataset with a standard recipe not tuned for each language. This makes it an ideal ground to test the optimizer’s adaptiveness in one shot. On the Amharic dataset, the official recipe ends at 83.48 bits per sentence on held-out FLORES-200 text (lower is better); the reactive optimizer, initialized with this same recipe, ends at 81.11 in one shot (on Swahili, we obtain a similar gain). Sweeping the learning rate of the standard recipe finds a static recipe that ends ahead of the reactive optimizer, at 80.89, but the reactive optimizer initialized from that swept recipe ends ahead again, at 80.73.

We also try the reactive optimizer on image generation, a direction where the literature usually does not sweep optimization hyperparameters, since diffusion-model design and optimizer design are viewed as mostly orthogonal¹. It ends ahead of the official recipe on LightningDiT-XL/1 ([18]; FID-50K 1.83 against 2.18 with guidance, 4.33 against 5.15 without), whose official recipe trains with AdamW, and on RAEv2-B (3.19 against 3.39), whose recipe trains with a Muon variant.

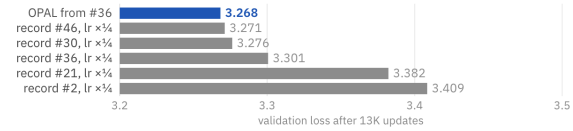
Figure 1 shows these runs, each with Claude Opus 5.5 powering the agent; we name our reactive optimizer OPAL (OPTimizer with an Agent in the Loop). Together, these results show that in one-shot settings, a reactive optimizer can lead to a better solution than adhering to a predetermined recipe.

Language modeling: modded-nanogpt Track 3

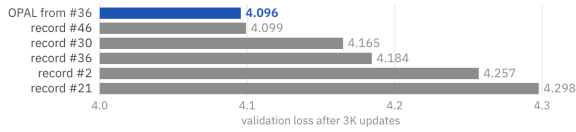
official NanoGPT



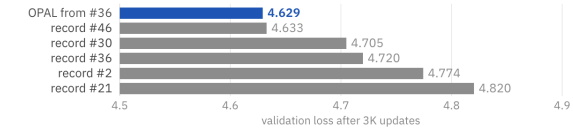
NanoGPT ¼ batch



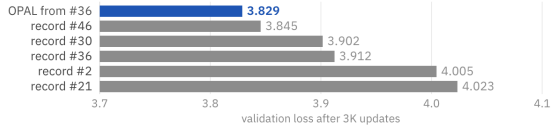
NanoGPT Chinese



NanoGPT Thai



NanoGPT Kyrgyz



Goldfish, Amharic

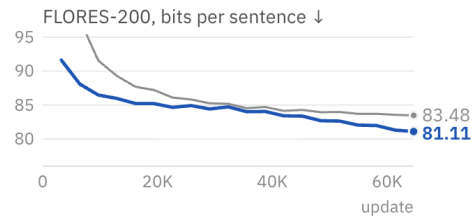
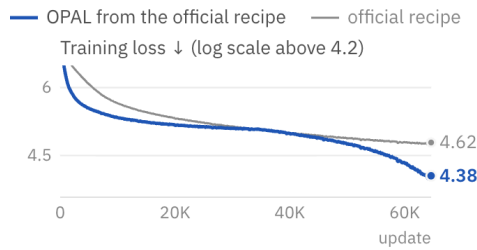
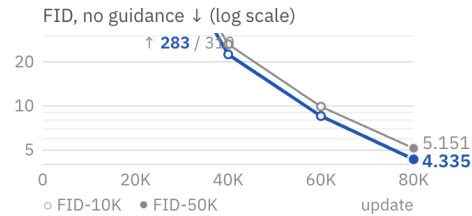
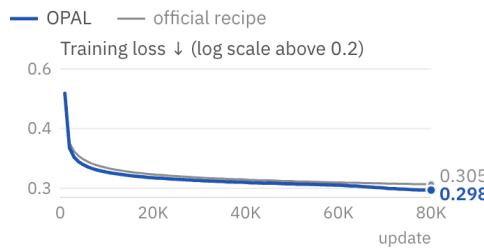


Image generation: LightningDiT-XL/1



RAEv2-B

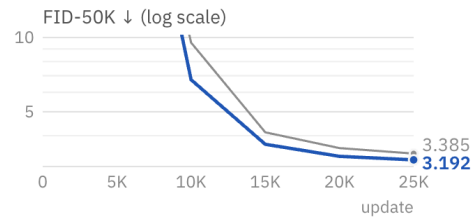
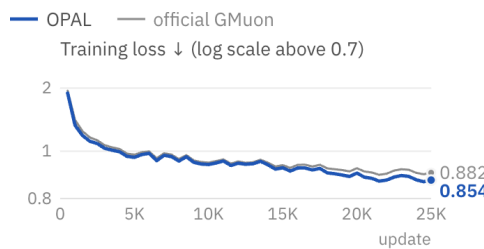


Figure 1. OPAL (Optimizer with an Agent in the Loop), our reactive optimizer, across different settings. In most settings where optimization hyperparameters are not heavily tuned for, reactive optimizers improve the training loss and test performance upon the canonical choices. In modded-nanogpt, #2 is AdamW, #21 is Shampoo, #36 is a tuned Muon, and #46 are more elaborate Muon-based recipes.

Beyond the practical benefit, we are intrigued by the spontaneous decisions made by the zero-shot agent, after observing and analyzing the training dynamics in its

own way. In the next two sections, we explore what the reactive optimizer chooses to watch and what it changes in response.

¹ For example, the standard recipes of DiT and SiT [19] train with AdamW at one constant learning rate. That has begun to change, with Muon and its variants taken up for diffusion transformers (CMuon [20]; Li et al. [21]; RAEv2’s GMuon [22]) and a recent controlled benchmark of optimizers for diffusion models [23].

2. Updating with spontaneous analysis

Figure 2 shows how an agent updates the optimizer code on the fly.

Concretely, we allow the agent to rewrite a file, `updater.py`, used in every step of optimization. We use code as the entire interface since this allows the agent to easily alter any part of the optimization. Yet, before the agent can make changes, it must observe, on its own, how training is progressing, leading to the second program it can rewrite,

`analysis.py`.

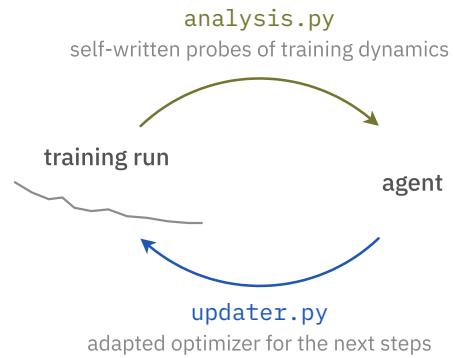


Figure 2. The two programs the agent writes and rewrites iteratively during training. With code as the entire interface, we give the agent maximal freedom to choose what to watch and what to change.

What information should the agent use to make its decisions? Raw training states would be impractical: the weights, activations, and gradients would swamp its context. Instead, the statistics and geometry of training dynamics could be helpful for decisions, such as the gradient norm of each parameter group, the size of an update next to the weights it changes, the cosine between consecutive gradients, and which tensors exhibit unusually large gradient spikes. Therefore, as with the optimizer, we give the agent the most freedom: it writes a program, `analysis.py`, to extract whatever it wants to know from the training dynamics. Its self-written probes run throughout training, taking from the state of training whatever the agent asks for and handing it to the agent for its next decision.

We complete the bidirectional loop. Through `analysis.py`, training reaches the agent in whatever form the agent chooses to see it, and through `updater.py`, the

agent’s decisions reach every step of training. The two programs let the agent devise new optimization algorithms along the way with maximum flexibility.

3. What is dynamically changing during optimization?

In this section, we look at what the reactive optimizer does throughout training, examining the decisions behind these results. As a summary, Figure 3 shows when, how, and why each agent edits its code.

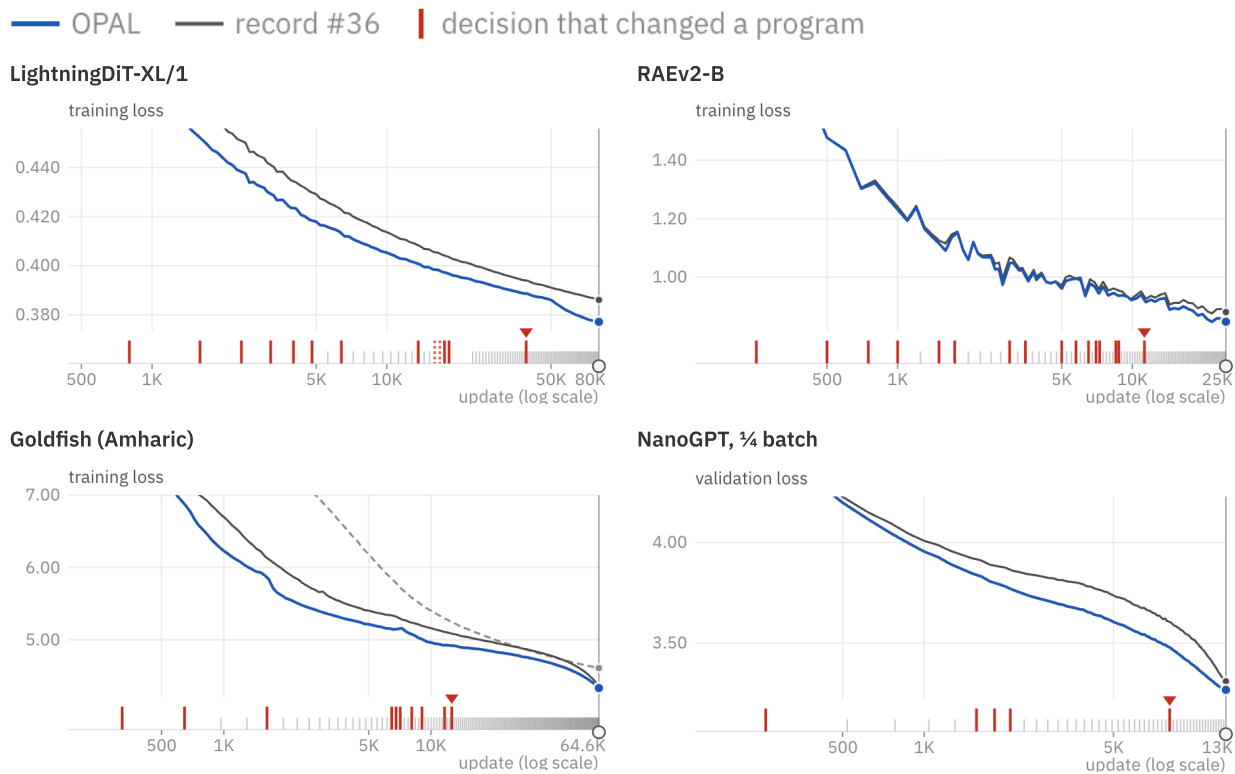


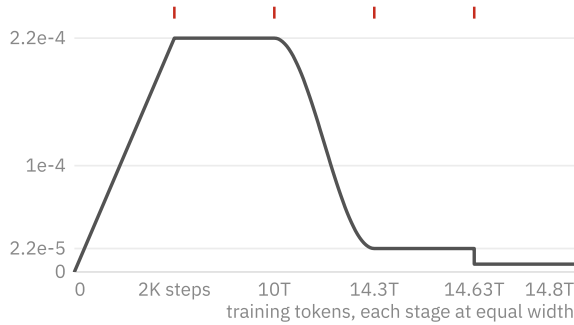
Figure 3. Overview of the reactive decisions of OPAL. Throughout training, OPAL builds its own audit signals and makes changes that are module-specific, triggered by its own observations, and timed to particular stretches of training.

The rest of this section examines these adaptations type by type.

3.1 Learning-rate schedule

Designed by hand

DeepSeek-V3

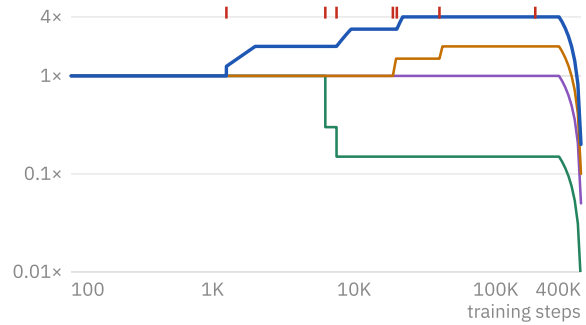


— learning rate
| a new stage

From the paper's Section 4.2 [24].

Set by OPAL

SiT-B/2



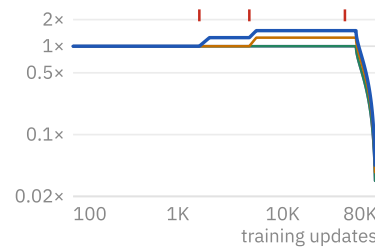
— block matrices
— adaLN modulation
— timestep embedder
— patch and class embedders

RAEv2-B



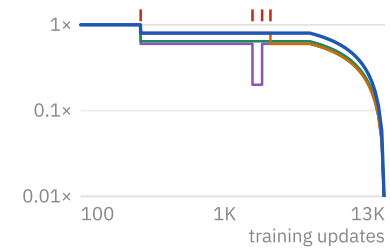
— backbone attention
— head attention
— vectors and timestep embedder

LightningDiT-XL/1



— attention and MLP matrices
— adaLN matrices
— timestep embedder, final adaLN
— everything else

NanoGPT, 1/4 batch



— q, k, projection and MLP matrices
— value matrices
— embedding
— final norm gain

Figure 4. DeepSeek-V3's learning rate, designed by hand (left), and learning rates OPAL set in four runs, as multiples of each recipe's rate.

Frontier training runs come with elaborate hyperparameter schedules, such as the five stages of DeepSeek-V3's learning rate ([24]; Figure 4, left), with such schedules sometimes being written by humans during the training process [25]. Engineers can make only so many decisions, each weighing only so many signals. A reactive optimizer can take over this work and has no such limit: it decides frequently, and at each decision, it can access the entire history of the run. As Figure 4 shows, the schedules our agents built are intricate, with separate multi-stage schedules for each parameter group, and differ from run to run. Several of them raise a learning rate mid-run, a move rarely seen in modern schedulers, yet naturally helpful in many scenarios [6, 26]. Moreover, the four examples below show the evidence they cited for their decisions.

We observe that the agents base these decisions on the diagnostics from `analysis.py`. Some examples are: the cosine between a module's consecutive gradients as a sign of overshooting (see "Deduced from clues" below); A/B testing between two learning rates every few hundred updates, comparing how fast the loss falls under each ("A-B-A test"); trial and error, with the learning rate raised, lowered, and raised again ("Trial and error"); and the trend of the loss, which makes agents start their cooldown earlier than planned ("Re-planned cooldown"). Each of these changes of schedule involves observation of the training dynamics and reasoning upon said observations.

▼ How the agents set the learning rate: four examples

Deduced from clues

The SiT agent's analysis program measured the cosine between consecutive gradients of each parameter group. Where it was negative, the agent lowered the learning rate: the timestep embedder's went to $0.3\times$ and then to $0.15\times$ of the official rate, with this stated reason:

Step 7,506, Decision 6

Timestep-embedder learning rate goes from $0.3\times$ to $0.15\times$ ($1.5e-5$) after update 7506, because its cosine is still around -0.22 .

Where the cosine stayed slightly positive, for the Muon matrices, it raised the learning rate.

A-B-A test

The RAEv2 agent alternated two learning rates over consecutive stretches of a few hundred updates (A-B-A) and compared the training loss under each. After one such test, it switched to $2\times$ from update 3,100, with this stated reason:

Update 3,000, Decision 12

The A-B-A test points the other way from my provisional read at update 2750: $2\times$ gives faster real progress than $1.5\times$.

Trial and error

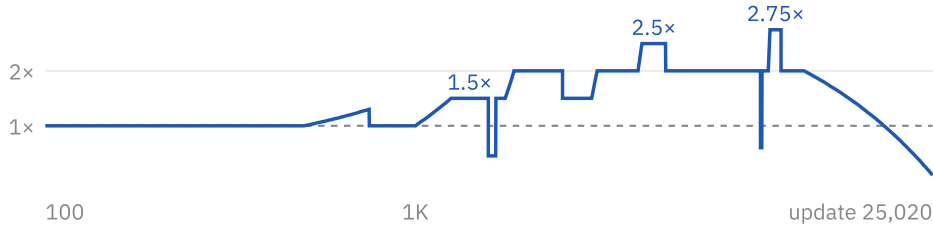
The same agent set every group's learning rate to $1.3\times$, returned the backbone's learning rate to $1\times$ when the backbone's gradients began to change sign at each step, and at the next decision raised every group's learning rate to $1.5\times$:

Update 1,000, Decision 4

My explanation from last time was wrong, so I'm raising the learning rate in every group to $1.5\times$ ($3e-4$).

It then ran learning rates of $2\times$, $2.5\times$ and $2.75\times$, each for about 650 updates, and kept $2\times$. Below, the learning rate of its backbone attention matrices, as a multiple of the official rate; the

two deep dips are 75-update reductions of the learning rate used as tests.



Re-planned cooldown

At step 190K, the SiT agent moved its planned cooldown from 320K to 280K and lowered its learning-rate floor from 0.1 to 0.05 of its peak (the bend in the SiT panel of Figure 4). Its stated reason:

Step 190,152, Decision 53

the constant-rate phase is now mostly holding the weights at a noise floor set by the learning rate, rather than making progress.

The RAEv2 agent started its cooldown at update 11,250, 45% of the run instead of the planned 75%, citing the trend of its loss and a paper on decaying the learning rate [27].

3.2 Optimizer design

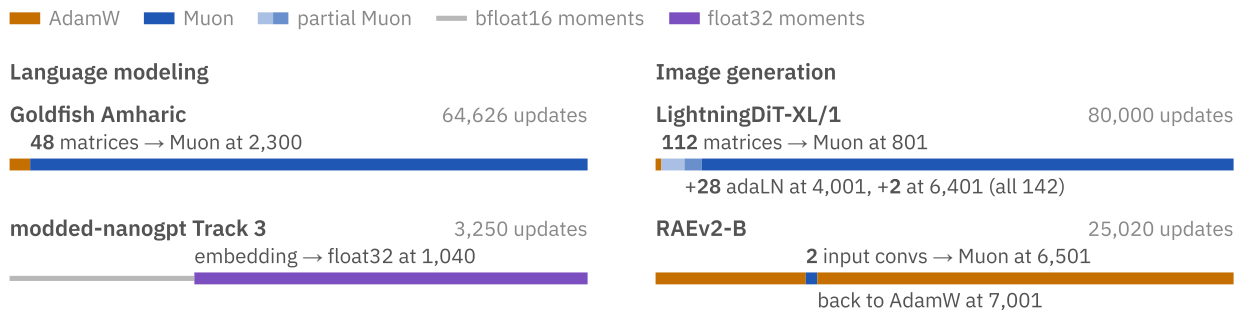


Figure 5. Changes of optimizer type and of precision in four OPAL runs. In some runs, the weights are simply moved to standard Muon, while in others, OPAL moves them gradually by module type and chooses high precision for a very small part of the weights, citing the signals it observed.

The reactive optimizers also change which rule updates which tensors. In our experiments, many switch from the initially given **AdamW** to the well-known hybrid, **Muon**, for the attention and MLP matrices while remaining with **AdamW** for the rest. Later, they make more specific moves (Figure 5).

The optimizer can also decide where precision matters. On modded-nanogpt Track 3, it measured that the entries of the token embedding are about 15 in size, where adjacent bfloat16 values are 0.0625 apart, so as the learning rate decayed, rounding would throw away a growing share of each update. From update 1,040, it kept that specific matrix’s Adam moments in **float32** and wrote its weights back with compensated rounding and left every other tensor as it was.

3.3 Momentum

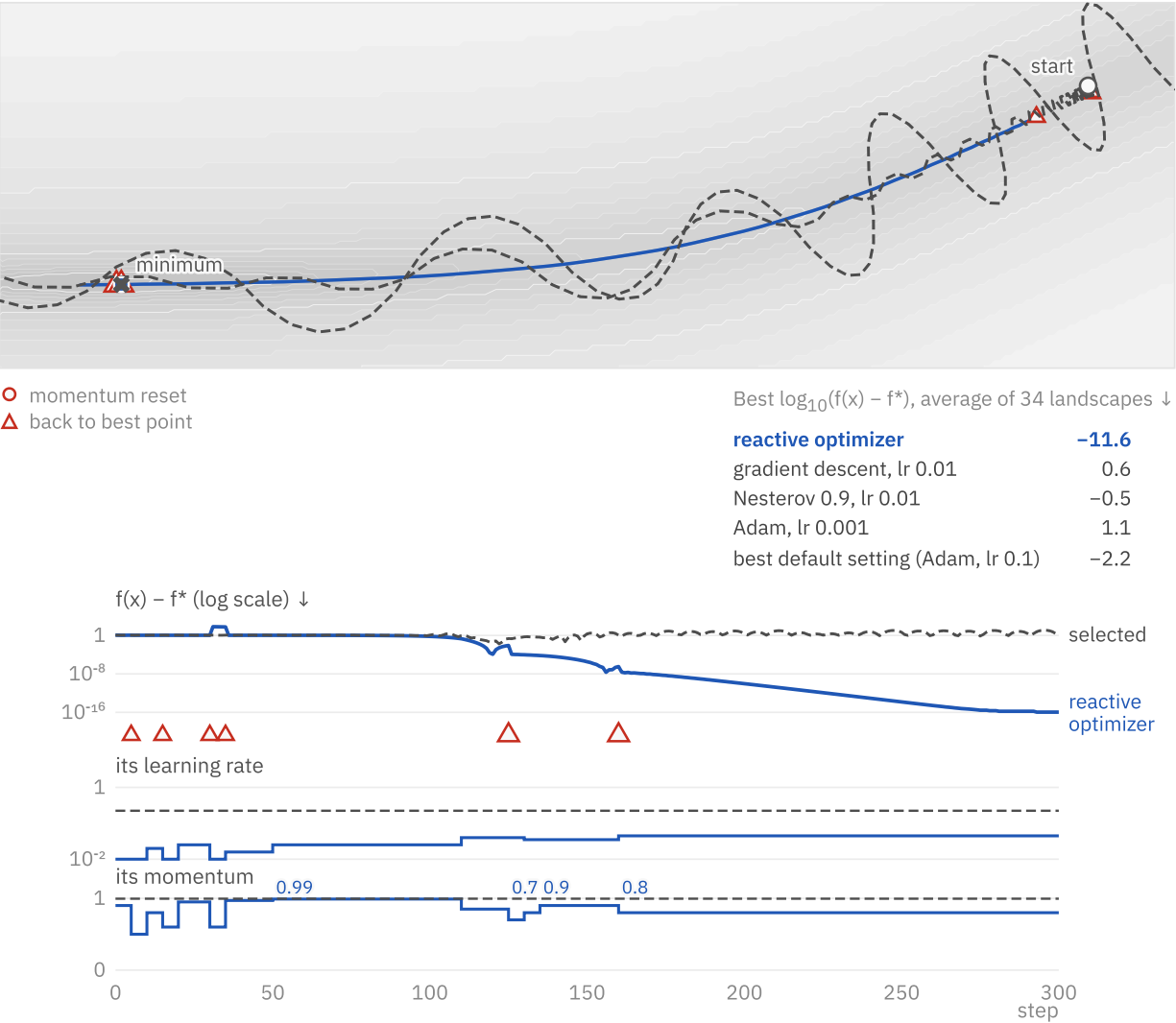


Figure 6. A variant of the tilted double-well potential, one of the 34 landscapes we test optimizers on. The dashed line is a static setting chosen with the sliders, the blue line is the reactive optimizer’s behavior. The reactive optimizer raises its momentum along the gentle slope, lowers it as the gradient falls, and clears it when it overshoots, reaching the optimum where common optimizers at their default settings stall.

We investigate the reactive optimizer’s ability to manipulate momentum on 2D toy landscapes, where we give the agent new abilities: resetting its momentum, returning to its best point so far, and jumping to a random nearby point. In these toy settings, standard OPAL would simply replace the default recipe with a second-order method within a few decisions and reach perfect performance very fast. We therefore restrict OPAL to the heavy-ball algorithm, while letting it control the hyperparameters. On the landscape in Figure 6, the start sits beside a nearly flat saddle where common optimizers stall. The reactive optimizer raises its momentum to cross the flat stretch, lowers it as the gradient falls near the minimum, and each time it overshoots, returns to its best point and restarts its momentum, reaching the minimum by the end of its 300-step budget. Over all 34 landscapes we ran, it achieves an average score of -11.6 ($\log_{10}(f - f^*)$, lower is better), while any single static setting does worse: -2.2 for the best of twelve common recipes, and -7.4 for the best of 382,080 swept recipes (Figure 6).

3.4 Loss spikes

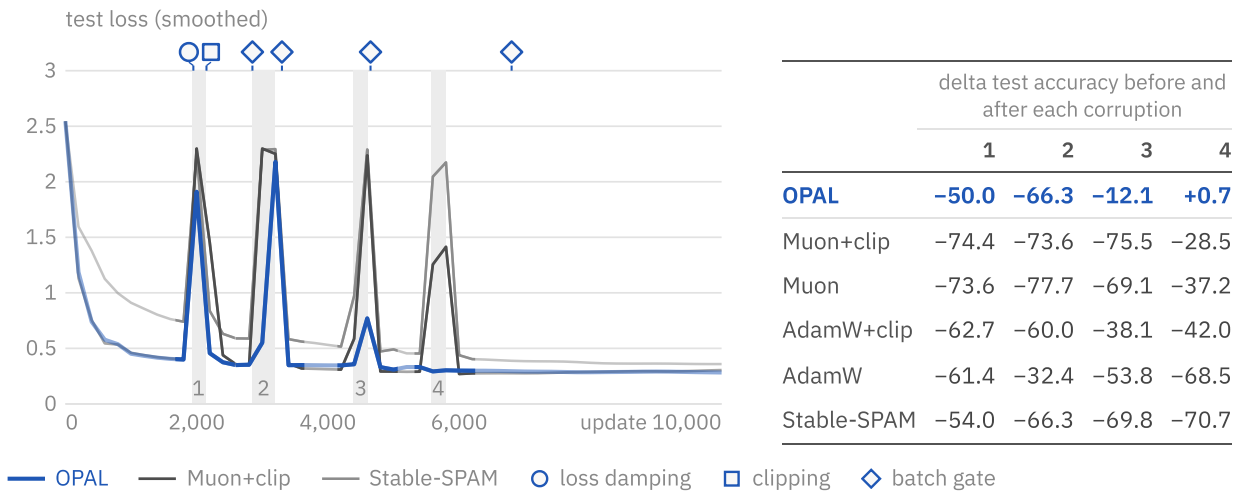


Figure 7. ResNet-18 on CIFAR-10 with four stretches of random images and labels, shaded. Left: test loss; hover a marker to read why OPAL changed its guard. Right: the change in test accuracy in each stretch. Reactive agents can defend against the designed type of loss spikes, induced by data corruption, after seeing it a few times.

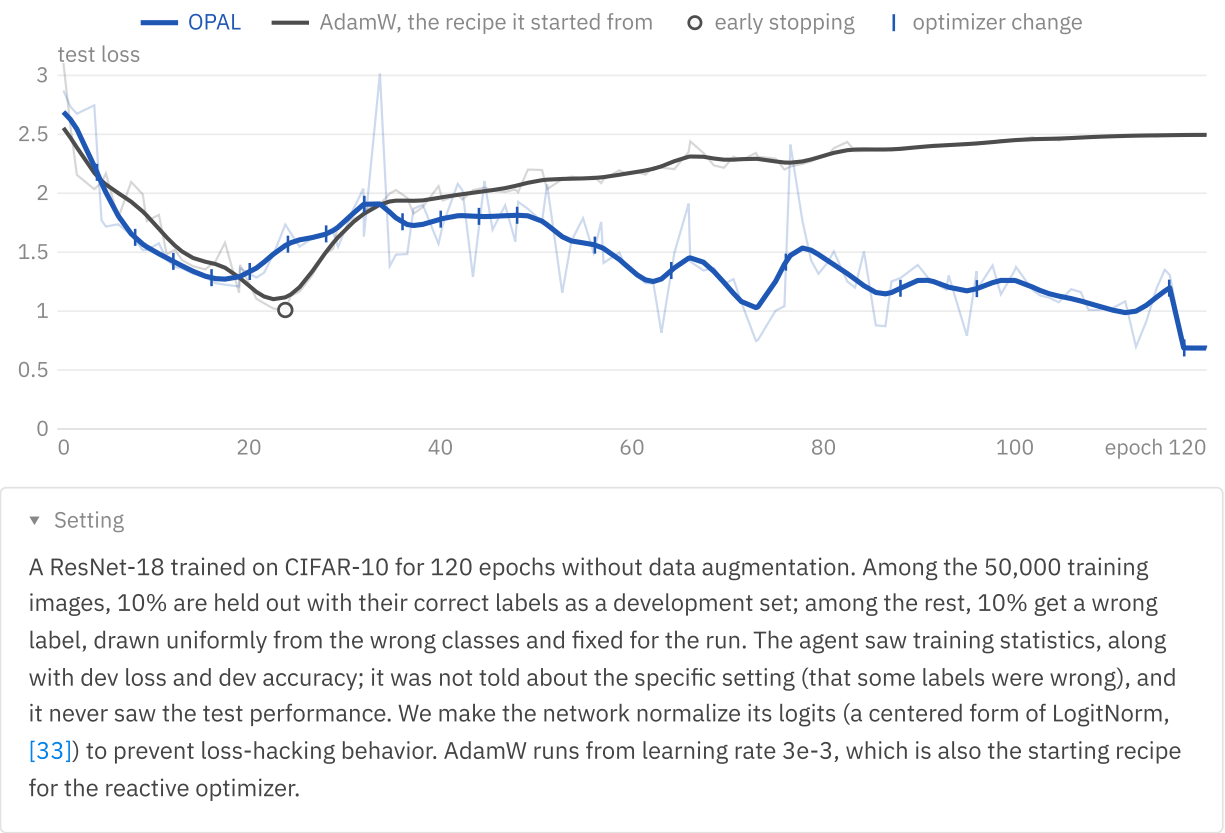
In our large-scale training, OPAL also limits the damage of gradient spikes with module-specific clipping: another SiT agent traced 17 of the 18 gradient spikes it logged to the timestep embedder, and added a clipping threshold for each tensor, set from that tensor’s own recent gradient norm, so that a spike in the embedder no

longer cuts the updates of every other tensor; its old global threshold stays only as a backstop. Neither move is standard practice, but both have precedents: clipping each tensor against its own history [28], and damping the one layer that spikes, as GLM-130B did for its embedding layer [29].

Spikes can also come from the data, through the interaction of particular batches with the model state [30], and clipping does not in general protect against noisy labels [31]. In a controlled experiment, we insert four stretches of random images and labels into CIFAR-10, and train a ResNet-18 with OPAL, which is not told about the corruption, and with static optimizers (Figure 7). The first two stretches cost OPAL 50 and 66 points of test accuracy, about as much as the baselines. In between, it added loss damping, replaced it with clipping, and then wrote a gate that holds back batches whose loss is far above recent clean ones. After it tuned that gate, the third stretch cost 12 points and the fourth none, while the clipped baselines and Stable-SPAM [32] still lost 28 to 76 points in each.

3.5 Overfitting

Test loss on CIFAR-10 with 10% mislabeled training images



	test loss	test accuracy	10% mislabeled training images, predicted as	
			true label ↑	assigned wrong label ↓
OPAL	0.685	78.6%	77.5%	3.7%
final weights: an average of its own weights from epochs 96–117, chosen on dev				
AdamW, the recipe it started from				
final weights	2.493	73.5%	0.0%	100.0%
dev-early-stopped	1.010	72.9%	59.7%	19.4%
EMA strategy adapted from OPAL	0.832	77.5%	70.0%	14.2%
a moving average of its weights, chosen on dev				

Figure 8. ResNet-18 trained for 120 epochs on 45,000 CIFAR-10 images, 10% with a wrong label. Hover the ticks at epochs 24 and 28: dev loss climbs and the agent reacts. OPAL counters the overfitting with progressively higher learning rates and a weight average it chooses on the development set.

Training on data with wrong labels is a standard way of testing overfitting [34]. We design such a setting (details under Figure 8) and ask if the optimizer can overcome overfitting by itself, reacting to signals from the training set and the development set. Trained straight through, the AdamW baseline ends at test loss 2.49 and

accuracy 73.5% (stopped early on the development set, at 1.01 and 72.9%; with its weights averaged and the average chosen there, at 0.83 and 77.5%). The reactive optimizer ends at loss 0.69 and accuracy 78.6%. As a direct measure of memorization, when asked to classify the 4,500 mislabeled training images, the baseline gives the wrong label it was trained on for all of them, its early-stopped version for 19% and its weight-averaged version for 14%. The reactive optimizer, on the other hand, gives that label for only 4%, and the true label for 78%. Its strategy combines progressively higher learning rates with weight averaging, and it chooses its final average on the development set, which its own analysis program calls “extended SWA” [35]. Hover the cursor on the ticks of Figure 8 to find out its exact decisions.



Summary of analysis

Above, we show that reactive optimizers can adapt their recipe to the dataset, to individual modules, and to the behavior of each run, a trait unseen in conventional optimizers. Such customization and self-adaptiveness allow them to craft sophisticated schedules and navigate complex training landscapes.

Still, tuning hyperparameters, suppressing loss spikes, and watching out for overfitting are things human experts can do, albeit less conveniently and at a higher cost. As agents get smarter, with hundreds of papers in their memory, thousands of tokens in their chain of thought, millions of training signals in their context, and every step in their hands, we are convinced that reactive optimizers will only become better, developing mechanisms during training more arcane than the ones demonstrated here, beyond what human experts could tune by hand. Such mechanisms may include tracking the second-order curvature of the loss landscape, auditing the variance of each data batch’s signal, or implementing never-before-seen optimization schemes.

4. Real-world applicability

Is a reactive optimizer practical? The very first concern is the time spent waiting for reasoning decisions. In its simplest design, training is stopped while the agent

reasons, inducing significant overhead. In our OPAL implementation, we instead provide the option to allow training to continue while the agent thinks, and swap in the agent’s edits at a later update (e.g., a median of 65 updates later on Goldfish). This should have little impact on performance, since most optimizer decisions are not especially time-critical. With this option, the runs in Figure 9 add only a little time on top of the static run (1.01× to 1.14×). At a larger scale, the current implementation still needs work to support sharded training and optimizer compilation; we leave these practical concerns to later work.

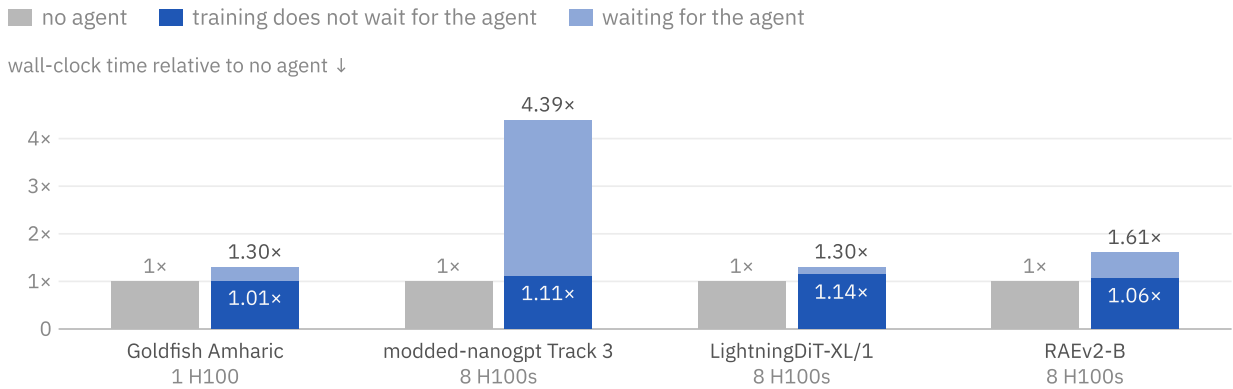


Figure 9. Wall-clock time relative to the same task trained without an agent. Training that waits for the agent at every decision takes much longer (1.30× to 4.39×), while training that does not wait adds only a little time on top of the static run (1.01× to 1.14×). That remaining overhead is the cost of running what the agent wrote: an update rule that costs more per step than the recipe’s (such as Muon’s orthogonalization, or keeping a tensor in float32) and the measurement cost of its analysis program.

When comparing API tokens to GPU hours, the reasoning of a reactive optimizer costs little compared with the training compute. At about 100 decisions in a single training run, the LightningDiT agent costs \$6.42 at Claude Opus 5.5 list price, on top of about 74 H100-hours of training (about \$510 at AWS’s current on-demand price); equipping it on larger training would only make this relative cost smaller. Admittedly, counted in FLOPs, any reactive optimizer would be impractical given the size of frontier agents. Here we simply want to point out that, for most researchers, GPU hours and token access to closed-source models are two different commodities and as the frontier continues to advance, the price of a given capability will continue to fall.

Another limitation is that the reactive optimizer needs room to observe and react; its gains shrink when a run is short and its recipe is fine-tuned. The modded-nanogpt track is such a setting: 3,250 updates of a recipe that many people have

tuned for nearly this exact task; our gains there are smaller than in the other tasks we tried, which train for about 8 to 120 times as many updates. Moreover, the reactive behavior introduces another source of uncertainty, potentially making training less reproducible. Most experiments in this post are run once, a few three times; reruns on the released package are in the [OPAL repository](#).

Fine-tuning platforms are bringing training to everyone. Tinker [36] and Prime Intellect’s Hosted Training [37] let users fine-tune open-weight models for their own applications without managing the GPUs behind them. On these platforms, however, the mental burden of a good training recipe still falls on people. A reactive optimizer fits naturally on top of these platforms, offloading that burden to agents and helping each user personalize a model to their own needs.

5. Conclusion

Until now, optimizers have been written before training. Agents, as a cheap form of reasoner with a long context window and expert knowledge, make something new possible: an optimizer that keeps deciding while the network trains in reaction to optimization events. We have demonstrated that such a reactive optimizer watched the training dynamics and rewrote its own code as training unfolded, for different groups of parameters, at the step it deemed proper. On toy and scaled-up experiments, most of these runs ended ahead of the recipes from which they started.

The analysis and update mechanism we design is meant to be as simple and general as possible. Wrapping a standard optimizer, it assumes nothing about the model, the data, or the training dynamics. It is useful today wherever no recipe exists yet. For example, a model on a biological or other scientific dataset, an architecture for which no recipe was written, or a language for which no one has tuned a recipe are all excellent use cases for such a reactive optimizer. This is because the optimizer’s agent takes the role of a human expert at countless training steps, where no expert has ever had the time to sit, making similar decisions in response to events in the run. Seen this way, a reactive optimizer is a natural thing to build, and its gains over static ones are to be expected. Furthermore, a training run holds far more information than humans can use, almost all of which is ignored today. A reactive optimizer has all this information

within reach. Imagine a world where the notion of a monolithic optimizer is gone; what replaces it is an optimization intelligence that adapts to every specific run, and it might not be gradient-based, but zeroth-order [38], language-based or more. Perhaps, in the future, we should delegate optimization decisions to agents, not humans.

In the past decade, deep learning models have come to possess unreasonably effective power through training. Yet the very heart of their making, the optimizer, is still mechanistic. Our post is an attempt to breathe the magic that training has given us into the core of training itself, forming a loop that we hope will create more capability in the future.



Code [repository](#)

```
pip install opal-optimizer
```

```
import torch
import opal

opt = torch.optim.AdamW(model.parameters(), lr=3e-4)
sched = torch.optim.lr_scheduler.CosineAnnealingLR(opt, T_max=steps)
opt = opal.wrap(opt, model, scheduler=sched, total_steps=steps,
                goal="...", max_cost_usd=20)

for step in range(steps):
    loss = criterion(model(x), y)
    loss.backward()
    opt.step(loss=loss)
    opt.zero_grad()
    sched.step()
```

Acknowledgements

We sincerely thank [George Cazenavette](#), [David Charatan](#), [Shivam Duggal](#), [Jyo Pari](#) and Qianwei Jia for their invaluable advice on this write-up. We are also grateful to [Phillip Isola](#) and the [Isola lab](#) for their insightful feedback on an earlier version of this post.

Citation

```
@misc{bai2026reactive,  
  title = {Reactive Optimizers},  
  author = {Bai, Xingjian and Liang, Jia-Hao and Teoh, Jayden and Sitzmann, Vincent},  
  year = {2026},  
  month = {oct},  
  url = {https://xingjianbai.com/reactive-optimizers/},  
  note = {Blog post}  
}
```

References

1. Kingma and Ba. Adam: A Method for Stochastic Optimization. ICLR 2015. [arXiv:1412.6980](#).
2. Gupta, Koren and Singer. Shampoo: Preconditioned Stochastic Tensor Optimization. ICML 2018. [arXiv:1802.09568](#).
3. Vyas et al. SOAP: Improving and Stabilizing Shampoo using Adam. [arXiv:2409.11321](#), 2024.
4. Jordan et al. Muon: An optimizer for hidden layers in neural networks. [Blog post](#), 2024.
5. Liu et al. Muon is Scalable for LLM Training. [arXiv:2502.16982](#), 2025.
6. Loshchilov and Hutter. SGDR: Stochastic Gradient Descent with Warm Restarts. ICLR 2017. [arXiv:1608.03983](#).
7. Hägele et al. Scaling Laws and Compute-Optimal Training Beyond Fixed Training Durations. NeurIPS 2024. [arXiv:2405.18392](#).
8. Yang et al. Tensor Programs V: Tuning Large Neural Networks via Zero-Shot Hyperparameter Transfer. [arXiv:2203.03466](#), 2022.
9. Lu et al. The AI Scientist: Towards Fully Automated Open-Ended Scientific Discovery. [arXiv:2408.06292](#), 2024.
10. Prime Intellect. Autonomous Speedrunning Experiment. [GitHub](#), 2026.
11. Chen et al. Symbolic Discovery of Optimization Algorithms. [arXiv:2302.06675](#), 2023.
12. Metz et al. VeLO: Training Versatile Learned Optimizers by Scaling Up. [arXiv:2211.09760](#), 2022.
13. Daniel, Taylor and Nowozin. Learning Step Size Controllers for Robust Neural Network Training. AAAI 2016. [doi:10.1609/aaai.v30i1.10187](#).
14. Jaderberg et al. Population Based Training of Neural Networks. [arXiv:1711.09846](#), 2017.
15. Zweiger, Pari, Guo, Akyürek, Kim and Agrawal. Self-Adapting Language Models. [arXiv:2506.10943](#), 2025.
16. Jordan. Modded-NanoGPT Optimization Benchmark. [GitHub](#), 2026.
17. Chang et al. Goldfish: Monolingual Language Models for 350 Languages. LREC 2026. [arXiv:2408.10441](#).
18. Yao, Yang and Wang. Reconstruction vs. Generation: Taming Optimization Dilemma in Latent Diffusion Models. [arXiv:2501.01423](#), 2025.

19. Ma et al. SiT: Exploring Flow and Diffusion-based Generative Models with Scalable Interpolant Transformers. [arXiv:2401.08740](#), 2024.
20. Chen, Sun and Yuan. CMuon: Accelerating and Stabilizing Diffusion Transformer Training via Chunked Momentum Orthogonalization. ECCV 2026. [arXiv:2608.02502](#).
21. Li et al. Scaling Muon for Diffusion Transformers. [arXiv:2608.20818](#), 2026.
22. Singh, Zheng, Wu, Zhang, Shechtman and Xie. Improved Baselines with Representation Autoencoders. [arXiv:2605.18324](#), 2026.
23. Bolatov et al. Optimizers for Diffusion Models: A Controlled Benchmark. [arXiv:2609.23055](#), 2026.
24. DeepSeek-AI. DeepSeek-V3 Technical Report. [arXiv:2412.19437](#), 2024.
25. Zhang et al. OPT: Open Pre-trained Transformer Language Models. [arXiv:2205.01068](#), 2022.
26. Smith. Cyclical Learning Rates for Training Neural Networks. WACV 2017. [arXiv:1506.01186](#).
27. Bergsma, Dey, Gosal, Gray, Soboleva and Hestness. Straight to Zero: Why Linearly Decaying the Learning Rate to Zero Works Best for LLMs. ICLR 2025. [arXiv:2502.15938](#).
28. Wang et al. AdaGC: Enhancing LLM Pretraining Stability via Adaptive Gradient Clipping. ICML 2026. [arXiv:2502.11034](#).
29. Zeng et al. GLM-130B: An Open Bilingual Pre-trained Model. [arXiv:2210.02414](#), 2022.
30. Chowdhery et al. PaLM: Scaling Language Modeling with Pathways. [arXiv:2204.02311](#), 2022.
31. Menon, Rawat, Reddi and Kumar. Can Gradient Clipping Mitigate Label Noise? ICLR 2020. [OpenReview](#), 2020.
32. Huang et al. Stable-SPAM: How to Train in 4-Bit More Stably than 16-Bit Adam. [arXiv:2502.17055v2](#), 2025.
33. Wei, Xie, Cheng, Feng, An and Li. Mitigating Neural Network Overconfidence with Logit Normalization. ICML 2022. [arXiv:2205.09310](#).
34. Nakkiran, Kaplun, Bansal, Yang, Barak and Sutskever. Deep Double Descent: Where Bigger Models and More Data Hurt. ICLR 2020. [arXiv:1912.02292](#).
35. Izmailov, Podoprikin, Garipov, Vetrov and Wilson. Averaging Weights Leads to Wider Optima and Better Generalization. UAI 2018. [arXiv:1803.05407](#).
36. Thinking Machines Lab. Announcing Tinker. [Blog post](#), 2025.
37. Prime Intellect. Getting Started (Hosted Training). [Documentation](#), accessed October 21, 2026.
38. Dahal, Mandal, Gülbahar and Vegesna. Dust: Pretraining Transformers Without Backpropagation. [Q Labs](#), 2026.