Yale University

Optimization for Deep Learning

S&DS 265 · Introductory Machine Learning · Lecture 12

Zhuoran Yang

Fall 2026

AI usage: figures were generated, and these slides polished, with the help of AI tools.

Outline

Learning Objectives

In this lecture you will learn:

1. How shuffling, minibatches, optimizer steps, and epochs define a training trajectory;
2. How a PyTorch loop connects the data stream, empirical risk, autodiff, and parameter update;
3. Why AdaGrad, RMSProp, Adam, and AdamW store different gradient histories;
4. How to choose and report a learning-rate schedule in the correct unit;
5. Why matrix-aware updates can help, and what state and systems costs they add.

Outline

Motivating Example: How Should Parameters Move?

Lecture 11 supplied the model, loss, initialization, and gradients. Now we must decide which examples define the next gradient, how that gradient becomes an update, and how the learning rate changes.

The data stream and update rule together define the training algorithm.

Example: training the MNIST MLP from Lecture 11.

The Training Dataset

Store n labeled observations as a map

$$i \longmapsto (x_i, y_i), \qquad i = 0, \ldots, n-1.$$

A PyTorch `Dataset` answers "what is observation i?" A `DataLoader` answers "which indices form the next batch, and how are their tensors assembled?"

These are separate from the model, loss, and optimizer.

Reading One Image Batch

For MNIST with batch size $B = 128$, one loader output has

$$X_{\mathcal{B}} : [128, 1, 28, 28], \qquad y_{\mathcal{B}} : [128].$$

The MLP flattens the images to $[128, 784]$ before the first linear layer.

One batch determines one gradient estimate

$$g_{\mathcal{B}}(\theta) = \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla_\theta \ell_i(\theta).$$

Problem Setup

Hold the data, MLP class, and empirical objective fixed:

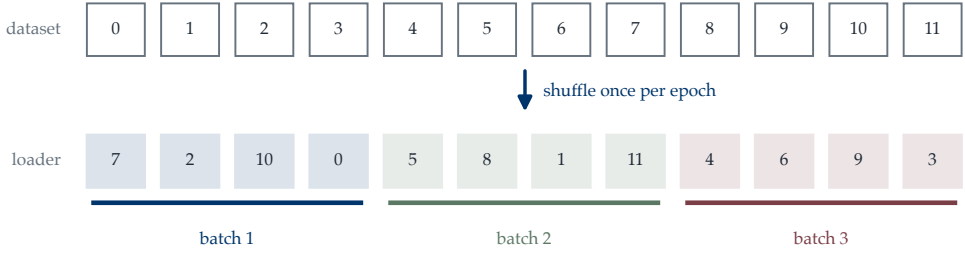
$$\widehat{R}_n(\theta) = \frac{1}{n}\sum_{i=1}^{n} \ell_i(\theta).$$

Today's fitted estimator is the output of a **training procedure**, not a new model class:

$$\hat{\theta}_T = \mathcal{A}_T(\theta_0; \mathcal{B}_{1:T},\ \eta_{1:T},\ \text{state}).$$

The loader supplies $\mathcal{B}_t$, autodiff estimates g_t, and the optimizer updates its state and θ_t. The objective is fixed; changing batch order, update geometry, or schedule can produce a different fitted parameter.

Shuffling Forms Batches Without Replacement



Every observation appears once; batch neighbors change after reshuffling.

Source: Declared permutation.

This Differs Slightly from the Lecture 5 Idealization

Lecture 5 analyzed the convenient with-replacement model

$$I_t \sim \text{Uniform}\{1, \dots, n\}$$

independently at every step. A common implementation instead draws one without-replacement permutation per epoch and walks through it in batches.

- Within an epoch, examples are sampled without replacement.
- Across epochs, a new permutation changes batch composition.
- Both produce useful stochastic gradients, but their dependence structures differ.

An Epoch Is One Pass through the Loader

If `drop_last=False`, the number of batches is

$$\boxed{\text{steps per epoch} = \left\lceil \frac{n}{B} \right\rceil}.$$

For MNIST, $n = 60{,}000$ and $B = 128$ give 469 optimizer steps per epoch, so E epochs process about En examples but make only $E\lceil n/B \rceil$ updates.

Always say whether a schedule is indexed by epochs, batches, or optimizer steps.

PyTorch Builds the Loader Explicitly

```
loader = DataLoader(
    train_dataset,
    batch_size=128,
    shuffle=True,
    num_workers=4,
    pin_memory=True,
    drop_last=False,
)
```

`shuffle=True` creates a new random order when iteration begins. Worker count and pinned memory affect throughput, not the mathematical loss.

The Last Batch and Reproducibility Are Decisions

If B does not divide n, the final batch is smaller.

- Keep it to use every observation.
- Drop it when fixed shapes or batch-dependent layers require a full batch.
- Seed the sampler and record worker seeding when exact replay matters.

In distributed training, a sampler partitions indices across devices; it must reshuffle consistently at each epoch.

Outline

The Optimizer Converts a Gradient into Parameter Motion

Backpropagation provides $g_t = \nabla_\theta L_{\mathcal{B}_t}(\theta_t)$. An optimizer maintains state s_t and applies an update rule

$$(\theta_{t+1}, s_{t+1}) = \mathcal{U}(\theta_t, g_t, s_t; \eta_t).$$

Different algorithms use the same gradients but retain different histories:

- Momentum smooths recent directions;
- AdaGrad and RMSProp estimate coordinatewise scale;
- Adam combines both; AdamW adds decoupled shrinkage;
- Shampoo and Muon use matrix structure for selected weights.

Here we compare update rules; residual connections and dropout belong to Lecture 11.

SGD and Momentum Are the Starting Point

Lecture 5 introduced

$$\theta_{t+1} = \theta_t - \eta_t g_t$$

and momentum

$$m_t = \beta m_{t-1} + (1-\beta) g_t, \qquad \theta_{t+1} = \theta_t - \eta_t m_t.$$

Momentum smooths directions across batches, but all coordinates still share one scalar learning rate. Adaptive methods change this geometry.

AdaGrad Accumulates Coordinatewise Gradient Scale

Keep a running sum of squared gradients, one number per coordinate, and divide the step by its square root:

$$v_t = v_{t-1} + g_t^2, \qquad \theta_{t+1} = \theta_t - \eta \frac{g_t}{\sqrt{v_t} + \varepsilon}.$$

AdaGrad Accumulates Coordinatewise Gradient Scale

Keep a running sum of squared gradients, one number per coordinate, and divide the step by its square root:

$$v_t = v_{t-1} + g_t^2, \qquad \theta_{t+1} = \theta_t - \eta \frac{g_t}{\sqrt{v_t} + \varepsilon}.$$

A coordinate with consistently large gradients accumulates a large v_t and therefore takes smaller steps; a flat coordinate keeps its own scale. One global η now serves both.

AdaGrad Accumulates Coordinatewise Gradient Scale

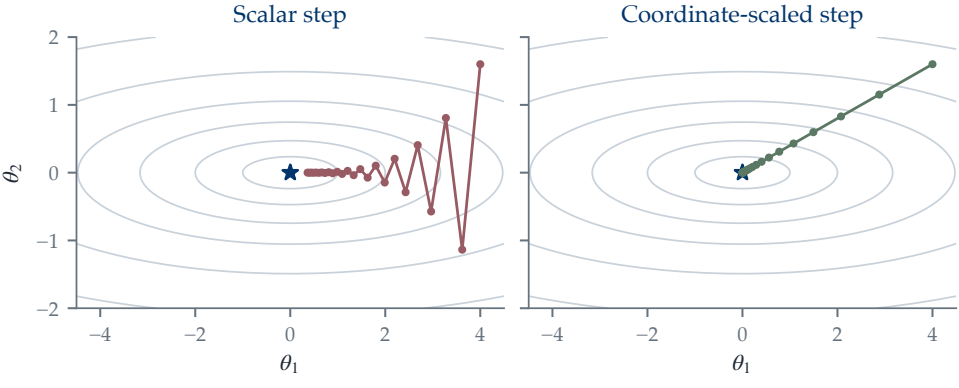
Keep a running sum of squared gradients, one number per coordinate, and divide the step by its square root:

$$v_t = v_{t-1} + g_t^2, \qquad \theta_{t+1} = \theta_t - \eta \frac{g_t}{\sqrt{v_t} + \varepsilon}.$$

A coordinate with consistently large gradients accumulates a large v_t and therefore takes smaller steps; a flat coordinate keeps its own scale. One global η now serves both.

v_t only grows, so every effective rate decays through training — which is what RMSProp fixes next.

One Rate per Coordinate Straightens the Path



The same ill-conditioned quadratic under both rules. A single scalar step must be small enough for the steep coordinate, so it zig-zags and crawls along the flat one; scaling each coordinate by its own accumulated gradient walks almost straight to the minimum.

Course calculation on a declared ill-conditioned quadratic.

RMSProp Forgets Distant Squared Gradients

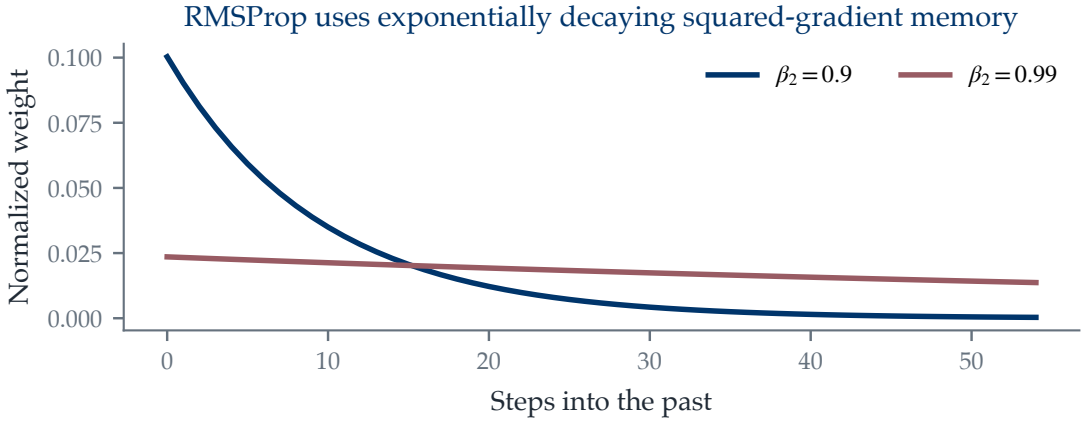
Replace AdaGrad's cumulative sum by an exponential moving average:

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2, \qquad \theta_{t+1} = \theta_t - \eta \frac{g_t}{\sqrt{v_t} + \varepsilon}.$$

The memory horizon is controlled by β_2; recent gradient scale matters more than distant history.

Source: D2L, Optimization Algorithms: RMSProp.

Exponential Memory Emphasizes Recent Steps



Course calculation of normalized exponential weights.

Adam Combines Momentum and RMSProp

$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t, \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2, \\
\widehat{m}_t &= m_t / (1-\beta_1^t), \qquad \hat{v}_t = v_t / (1-\beta_2^t), \\
\theta_{t+1} &= \theta_t - \eta_t \frac{\widehat{m}_t}{\sqrt{\widehat{v}_t} + \varepsilon}.
\end{aligned}
$$

The first moment smooths direction; the second moment rescales coordinates.

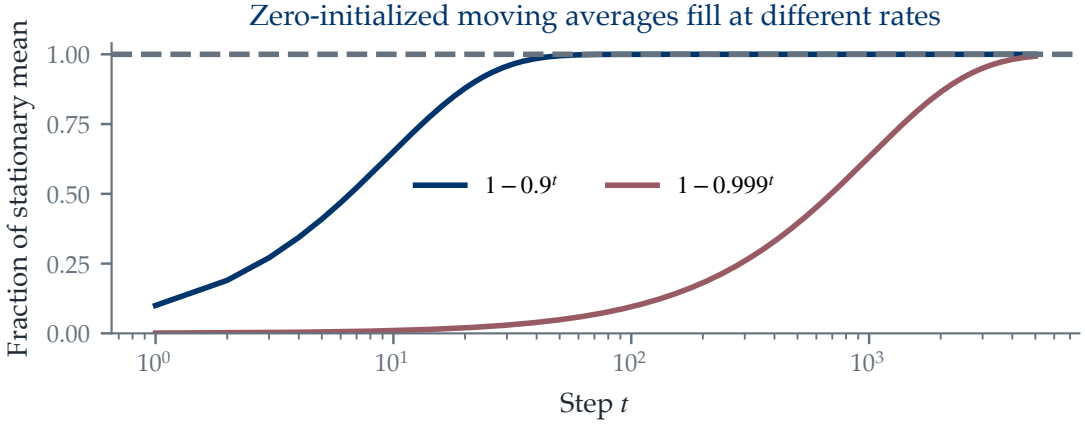
Source: Kingma and Ba, 2015.

Bias Correction Removes the Zero-Initialization Transient

Starting $m_0 = v_0 = 0$ pulls early moving averages toward zero. The denominators $1 - \beta_1^t$ and $1 - \beta_2^t$ correct that startup bias.

For a constant gradient, the corrected first and second moments recover g and g^2 at the first step.

Bias Correction Matters Most at the Beginning



Course calculation of moving-average fill-in.

One Adam Step Reveals the Coordinate Scaling

Suppose $g_1 = (2, 0.5)$ and $m_0 = v_0 = 0$. After bias correction,

$$\widehat{m}_1 = g_1, \qquad \hat{v}_1 = g_1^2.$$

Ignoring ε,

$$\Delta\theta_1 = -\eta \frac{(2, 0.5)}{(2, 0.5)} = -\eta(1, 1).$$

At later steps, different gradient histories produce different effective coordinate learning rates.

Why L2 in the Loss Is Not Weight Decay for Adam

Put the penalty in the loss and its gradient $\lambda\theta$ joins g_t, so it passes through the same normalization as everything else:

$$\theta_{t+1} = \theta_t - \eta \frac{\widehat{m}_t(g_t + \lambda\theta_t)}{\sqrt{\widehat{v}_t} + \varepsilon}.$$

Why L2 in the Loss Is Not Weight Decay for Adam

Put the penalty in the loss and its gradient $\lambda\theta$ joins g_t, so it passes through the same normalization as everything else:

$$\theta_{t+1} = \theta_t - \eta \frac{\widehat{m}_t(g_t + \lambda\theta_t)}{\sqrt{\widehat{v}_t} + \varepsilon}.$$

The shrinkage a coordinate receives is therefore divided by its own gradient history: a coordinate with small $\hat{v}_t$ is decayed far harder than one with large $\hat{v}_t$, though λ is the same for both.

The intended reading of λ — shrink every weight by the same factor per step — is simply not what happens.

AdamW Decouples the Decay

Take the penalty out of the loss and apply it directly to the parameter, outside the adaptive update:

$$\theta_{t+1} = \underbrace{(1 - \eta_t \lambda)\theta_t}_{\text{uniform shrink}} - \eta_t \frac{\widehat{m}_t}{\sqrt{\widehat{v}_t} + \varepsilon}$$

AdamW Decouples the Decay

Take the penalty out of the loss and apply it directly to the parameter, outside the adaptive update:

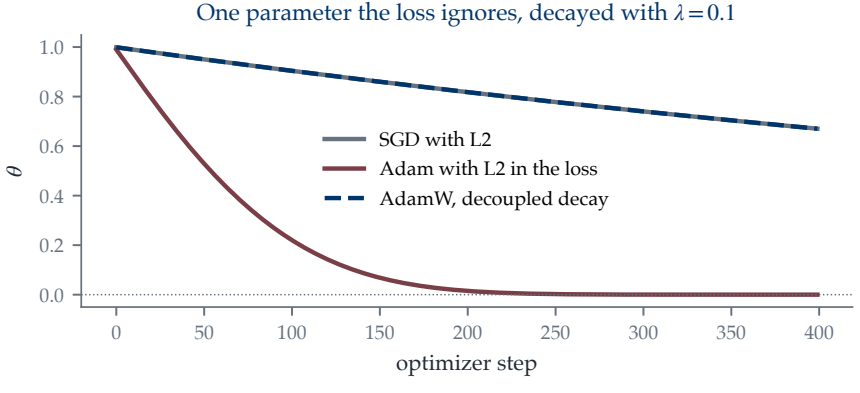
$$\theta_{t+1} = \underbrace{(1 - \eta_t \lambda)\theta_t}_{\text{uniform shrink}} - \eta_t \frac{\widehat{m}_t}{\sqrt{\widehat{v}_t} + \varepsilon}$$

Now every coordinate shrinks by the same factor $1 - \eta_t \lambda$ per step, whatever its gradient history, and λ means what it says.

The deep-learning default, usually with separate parameter groups so that biases and normalization scales are not decayed.

Source: Loshchilov and Hutter (2019).

The Two Rules on a Parameter the Loss Ignores



With zero gradient, decay is all that acts. SGD and AdamW shrink geometrically by $1 - \eta\lambda$, reaching 0.67 after 400 steps. Adam with L2 divides $\lambda\theta$ by its own running scale, so the step barely shrinks with θ and the weight is driven to zero by step 200.

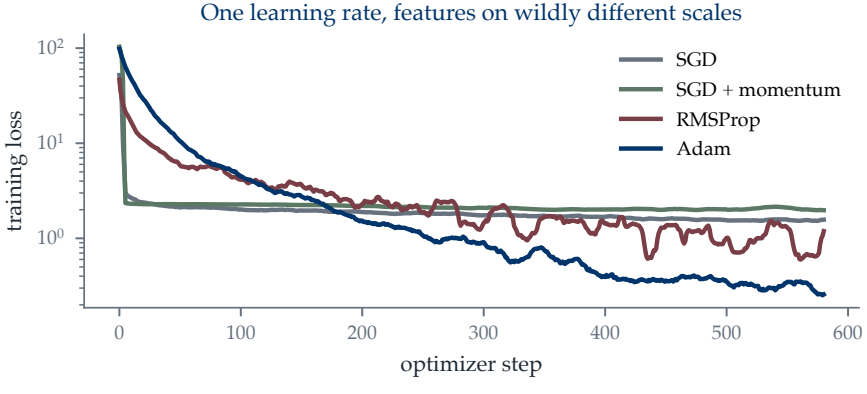
$\eta = 0.01$, $\lambda = 0.1$, PyTorch `Adam(weight_decay)` against `AdamW`.

The Optimizers Store Different Histories

method	state per parameter	effective geometry
SGD	0	scalar learning rate
momentum	1	smoothed direction
AdaGrad	1	cumulative diagonal scale
RMSProp	1	recent diagonal scale
Adam / AdamW	2	direction + recent diagonal scale

For a large model, state memory is a first-class algorithmic cost.

Where Adaptivity Actually Pays



A small network on digit pixels with one block of features stretched by 100, so the curvature is wildly unequal. At one shared learning rate, SGD reaches 1.57 and momentum 1.98 in 600 steps, RMSProp 1.19, and Adam 0.26: per-coordinate scaling is what makes a single η workable.

Identical initialization, batch 128, $\eta = 10^{-3}$ for every method.

Outline

One Epoch Connects Loader, Autodiff, and Optimizer

```
model.train()
for x, y in loader:
    x, y = x.to(device), y.to(device)
    optimizer.zero_grad(set_to_none=True)
    logits = model(x)
    loss = loss_fn(logits, y)
    loss.backward()
    optimizer.step()
```

Iterating through `loader` once completes one epoch. Each loop body normally completes one optimizer step.

The Operations Have Separate Responsibilities

1. Loader chooses and assembles the data;
2. Model computes logits and saves required activations;
3. Loss reduces predictions and labels to a scalar;
4. Backward fills parameter gradient buffers;
5. Optimizer reads gradients, updates state, then parameters.

A bug at any interface can leave the loop running while training the wrong procedure.

Gradient Accumulation Separates Microbatches from Steps

If one large batch does not fit in memory, accumulate K microbatch gradients:

$$g_t = \frac{1}{K} \sum_{k=1}^{K} g_{t,k},$$

then call `optimizer.step()` once.

- Loader batches processed: K;
- Backward calls: K;
- Optimizer steps: 1.

Divide each microbatch loss by K and do not zero gradients between the microbatches.

Accumulation in PyTorch

```
optimizer.zero_grad(set_to_none=True)
for k, (x, y) in enumerate(loader):
    loss = loss_fn(model(x), y) / accum_steps
    loss.backward()
    if (k + 1) % accum_steps == 0:
        optimizer.step()
        optimizer.zero_grad(set_to_none=True)
```

Batch-dependent layers can behave differently under microbatching even when the accumulated gradient is otherwise equivalent.

Learning Rate Is Indexed by Optimizer Step

A scheduler implements η_t after defining what counts as t.

- Warmup increases the rate during early optimizer steps;
- Cosine or piecewise decay reduces it later;
- An epoch-level scheduler changes more coarsely;
- Accumulation reduces the number of optimizer steps per data pass.

Report the peak rate, warmup length, total steps, and decay rule together.

Choose a Learning Rate by a Controlled Sweep

Hold model, data order, batch size, and budget fixed. Try a small logarithmic grid and inspect

$$\widehat{R}_{\text{train}}(t), \quad \widehat{R}_{\text{val}}(t), \quad \|g_t\|, \quad \frac{\|\Delta\theta_t\|}{\|\theta_t\| + \varepsilon}.$$

A rate is useful only if it is stable and reaches good validation quality within the available compute.

AdamW with Warmup and Cosine Decay

```
optimizer = torch.optim.AdamW(
    model.parameters(), lr=1e-3, weight_decay=1e-2)
scheduler = torch.optim.lr_scheduler.SequentialLR(
    optimizer, schedulers=[warmup, cosine],
    milestones=[warmup_steps])

# call after each optimizer.step()
scheduler.step()
```

Match the scheduler's step unit to the intended schedule.

Training and Evaluation Use Different Modes

Training

`model.train()`; gradients enabled; dropout active; batch statistics updated.

Evaluation

`model.eval()`; `torch.no_grad()`; dropout disabled; stored normalization statistics used.

Validation must not update parameters, optimizer state, or training-only layer statistics.

Outline

Why Look at a Weight as a Matrix

Lecture 5 diagnosed slow training as unequal curvature: a step safe in the steep direction crawls in the flat one. AdaGrad and Adam answered by rescaling each coordinate by its own gradient history.

Why Look at a Weight as a Matrix

Lecture 5 diagnosed slow training as unequal curvature: a step safe in the steep direction crawls in the flat one. AdaGrad and Adam answered by rescaling each coordinate by its own gradient history.

But a layer's weights are a matrix, and its update has directions that no single coordinate names. If the momentum matrix M_t has a few large singular values, the update moves almost entirely along those directions and barely at all along the rest.

Why Look at a Weight as a Matrix

Lecture 5 diagnosed slow training as unequal curvature: a step safe in the steep direction crawls in the flat one. AdaGrad and Adam answered by rescaling each coordinate by its own gradient history.

But a layer's weights are a matrix, and its update has directions that no single coordinate names. If the momentum matrix M_t has a few large singular values, the update moves almost entirely along those directions and barely at all along the rest.

That is the Lecture 5 problem again, one level up: unequal scale across directions rather than across coordinates.

Adam Treats a Weight Matrix Coordinatewise

For $W \in \mathbb{R}^{m \times n}$, Adam stores two $m \times n$ state matrices and rescales each entry independently:

$$\Delta W_{jk}^{\text{Adam}} \propto -\frac{\widehat{M}_{jk}}{\sqrt{\widehat{V}_{jk}} + \varepsilon}.$$

Neural-network weights have row and column structure that a diagonal preconditioner does not represent.

Shampoo Stores Row and Column Gradient Geometry

For gradients $G_t \in \mathbb{R}^{m \times n}$, Shampoo accumulates

$$L_t = \varepsilon I_m + \sum_{s \le t} G_s G_s^\top, \qquad R_t = \varepsilon I_n + \sum_{s \le t} G_s^\top G_s,$$

and uses

$$\boxed{\Delta W_t = -\eta L_t^{-1/4} G_t R_t^{-1/4}}.$$

State falls from an intractable full $(mn) \times (mn)$ matrix to $m^2 + n^2$ entries, but inverse roots and communication remain expensive.

Source: Gupta, Koren, and Singer, 2018.

SOAP Runs Adam Inside Shampoo's Basis

Shampoo's preconditioners change slowly, so their eigenvectors can be reused. **SOAP** keeps that rotation and runs plain Adam inside it:

- Eigendecompose L_t and R_t occasionally, not every step;
- Rotate the gradient into that basis, take an Adam step there, rotate back.

SOAP Runs Adam Inside Shampoo's Basis

Shampoo's preconditioners change slowly, so their eigenvectors can be reused. **SOAP** keeps that rotation and runs plain Adam inside it:

- Eigendecompose L_t and R_t occasionally, not every step;
- Rotate the gradient into that basis, take an Adam step there, rotate back.

The expensive part now happens every few hundred steps, and what runs each step is the optimizer everyone already knows how to tune.

Reported to reach a target loss in fewer steps than AdamW on language-model pretraining, at a modest wall-clock overhead.

Source: Vyas et al. (2024), "SOAP: Improving and Stabilizing Shampoo using Adam".

Muon Orthogonalizes a Momentum Matrix

If $M_t = U\Sigma V^\top$ is a momentum-like update, Muon approximately maps

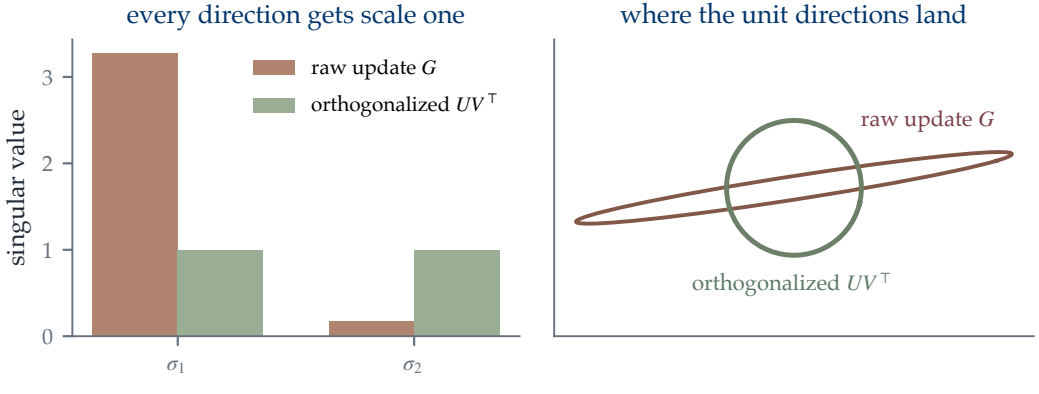
$$M_t \longmapsto UV^\top,$$

replacing unequal singular values by a common scale. Every singular direction then gets the same step length. This is matrix normalization along singular directions, rather than coordinates. Practical code uses a few Newton–Schulz iterations because only $UV^\top$ is needed.

Muon is applied only to selected two-dimensional hidden weights; embeddings, biases, gains and output heads use AdamW.

Source: Keller Jordan's Muon write-up; Liu et al., 2025.

Orthogonalization Equalizes the Step



Left: the update's singular values, 3.27 and 0.17, both replaced by one. Right: where the same unit directions land after one step. The raw update stretches one direction nearly twenty times more than the other; the orthogonalized update moves equally in all of them.

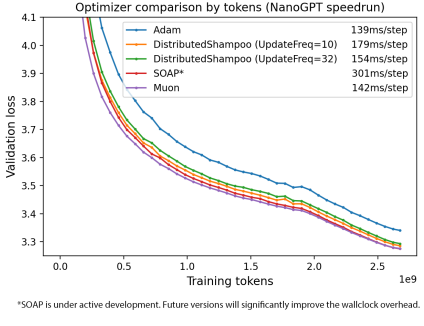
Course calculation on a declared 2×2 update matrix.

Structured Geometry Changes State and Computation

method	state for $m \times n$ weight	extra work
AdamW	$2mn$	elementwise operations
Shampoo	$m^2 + n^2$	matrix inverse roots
Muon	about mn	approximate orthogonalization

Block size, update frequency, numerical precision, and distributed sharding determine whether the theoretical geometry is practical.

Muon Has Promising but Conditional Evidence



Source: Keller Jordan, “Muon,” 2024; attributed classroom excerpt.

Optimizer Choice Remains an Empirical Procedure

Matrix-aware methods can improve time-to-quality in some transformer pretraining configurations, but the result depends on

- Parameter grouping and update scaling;
- Learning-rate and weight-decay tuning;
- Matrix shapes, batch size, and precision;
- Communication and kernel efficiency.

Use AdamW as a tuned baseline and compare under a declared compute and memory budget.

Outline

Training Memory Has Four Main Components

1. Parameters, possibly with full-precision master copies;
2. Gradients or communication buckets;
3. Optimizer state such as momentum and second moments;
4. Saved activations and temporary workspaces.

DataLoader workers consume host memory, while prefetched or pinned batches occupy additional host-side buffers.

Data and Optimizer Bottlenecks Look Different

symptom	likely data-side cause	likely optimizer-side cause
accelerator idle	slow workers / transfer	synchronization
loss spikes	corrupt batch / scale	rate or state instability
low throughput	decoding / collation	expensive preconditioner
memory overflow	prefetch / batch size	state / activations

Profile the pipeline before changing the mathematics.

A Fair Optimizer Comparison Fixes the Data Stream

Match

- Model initialization and batch order where possible;
- Number of processed examples or tokens;
- Tuned learning-rate schedules and parameter groups;
- Validation protocol and stopping rule;
- Hardware, precision, and distributed configuration.

Report time-to-quality, final quality, peak memory, and variability—not only loss after a convenient number of updates.

A Minimal Debugging Order

1. Overfit one tiny batch;
2. Verify loader shapes, labels, and shuffling;
3. Compare manual and autodiff gradients on a small case;
4. Inspect loss, gradient norms, and update ratios;
5. Add validation and a schedule;
6. Only then compare more elaborate optimizers.

A sophisticated optimizer cannot repair a mismatched target or broken data pipeline.

Summary: Data Stream and Steps

1. A DataLoader turns one shuffled permutation into minibatches; one pass is an epoch.
2. Steps, batches, epochs, and accumulated microbatches are different units.
3. A learning-rate schedule must name which of those units indexes it.

$$g_t = \frac{1}{|\mathcal{B}_t|} \sum_{i \in \mathcal{B}_t} \nabla \ell_i(\theta_t), \qquad \text{steps per epoch} = \left\lceil \frac{n}{B} \right\rceil.$$

Summary: Optimizer State and Geometry

1. AdaGrad, RMSProp, Adam, and AdamW differ through the gradient history they store and how it rescales updates.
2. Shampoo and Muon exploit matrix structure but add systems costs.
3. Optimizer evidence must control the data stream, schedule, memory, and compute budget.

$$\Delta\theta_t^{\mathrm{AdamW}} = -\eta_t \frac{\widehat{m}_t}{\sqrt{\widehat{v}_t} + \varepsilon} - \eta_t \lambda \theta_t.$$

Next Lecture

The data stream and update rule now specify how parameters move. For images, the remaining choice is architectural: should every pixel location have an unrelated weight, or should local evidence be reused across positions?

Lecture 13 — Convolutional Neural Networks encodes locality and weight sharing, derives the resulting tensor shapes, and builds a complete image classifier with pooling, residual blocks, and batch normalization.

Reading for this lecture: D2L Chapter 12, Optimization Algorithms; DL Chapter 8.