CSE 592: Foundations of AI — University of Michigan

Lecture 10: Optimization



Gabriel Poesia
Assistant Professor, CSE
University of Michigan, Ann Arbor
poesia@umich.edu
https://gpoesia.com

Plan

0. Administrivia
1. Recap
2. Backpropagation
3. Automatic differentiation
4. Momentum and adaptive steps
5. Adam and beyond

Administrivia

HW2 due this Wednesday

Two issues we'll fix today (thanks for reporting!)

(1) Autograder overcounting points, (2) no outline to mark pages on your PDF

Recap

Where we left off

Two-layer neural network: score $= w \cdot \sigma(V\phi(x))$.

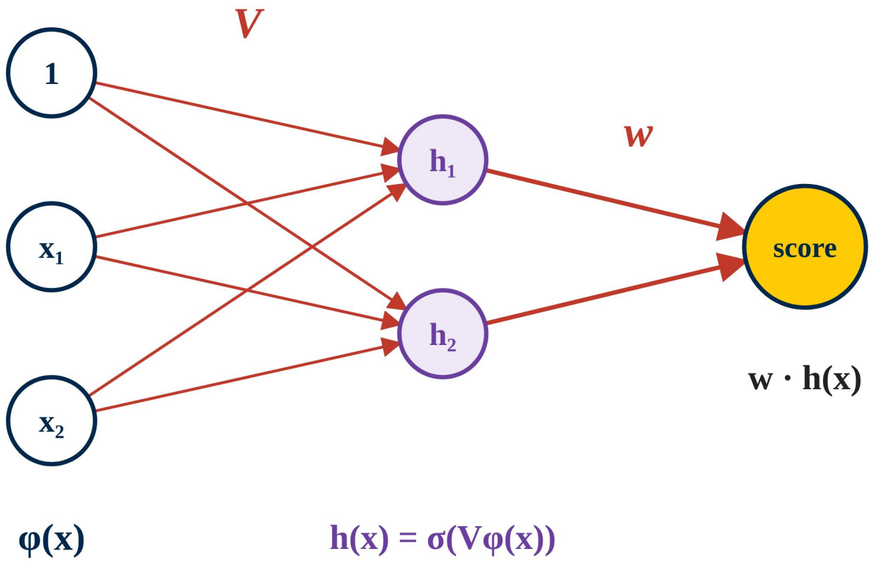
Features $h(x) = \sigma(V\phi(x))$ are **learned**, along with w.

Same losses as before. Trained with SGD.

SGD needs $\nabla_V \text{Loss}$ and $\nabla_w \text{Loss}$ for every example.

Q: How do we compute them?

Idea will generalize to deeper networks with many more layers.



Backpropagation

A deeper network

Suppose a 3-layer NN with ReLUs for regression, squared loss.

Recall: $\mathrm{ReLU}(v) = \max(0, v)$ (element-wise).

Given (x, y), we compute the loss L as:

$$L = \Big(w \cdot \mathrm{ReLU}\big(W_2\, \mathrm{ReLU}(W_1\, \phi(x))\big) - y\Big)^2$$

Gradient-based optimizers need $\nabla_{W_1} L$, $\nabla_{W_2} L$ and $\nabla_w L$.

Analytically, will get hairy and different for each new architecture.

Instead, we'll do **backpropagation**, with two passes:

- **Forward pass:** compute one layer at a time, up to the loss.
- **Backward pass:** go back from the loss towards each parameter, computing intermediate gradients with chain rule.

Result: loss (forward pass) + $\nabla_P L$ for each parameter P (w, W_1, etc)

Forward pass

Forward: break down computation and save intermediate values:

$$\text{Start: } \phi(x)$$
$$z_1 = W_1\,\phi(x)$$
$$h_1 = \mathrm{ReLU}(z_1)$$
$$z_2 = W_2\,h_1$$
$$h_2 = \mathrm{ReLU}(z_2)$$
$$s = w \cdot h_2$$
$$r = s - y$$
$$L = r^2$$

One simple operation in each step.

Each step only uses values from before.

Keep all of them for backward pass: z_1, h_1, z_2, h_2, s, r.

Backward pass: the chain rule

Forward pass:

$$z_1 = W_1\,\phi(x)$$
$$h_1 = \mathrm{ReLU}(z_1)$$
$$z_2 = W_2\,h_1$$
$$h_2 = \mathrm{ReLU}(z_2)$$
$$s = w \cdot h_2$$
$$r = s - y$$
$$L = r^2$$

Q: How do we go backwards now?

A: use chain rule for partial derivatives!

To simplify the example, treat each value as a scalar for now.

$$\frac{\partial L}{\partial r} = 2r$$
$$\frac{\partial L}{\partial s} = \frac{\partial L}{\partial r} \cdot \frac{\partial r}{\partial s} = 2r \cdot 1$$
$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial s} \cdot \frac{\partial s}{\partial w} = 2r \cdot h_2$$
$$\frac{\partial L}{\partial h_2} = \frac{\partial L}{\partial s} \cdot \frac{\partial s}{\partial h_2} = 2r \cdot w$$
$$\frac{\partial L}{\partial z_2} = \frac{\partial L}{\partial h_2} \cdot \frac{\partial h_2}{\partial z_2} = 2r\,w \cdot \mathbf{1}[z_2 > 0]$$
$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial z_2} \cdot \frac{\partial z_2}{\partial W_2} = 2r\,w\,\mathbf{1}[z_2 > 0] \cdot h_1$$
$$\frac{\partial L}{\partial h_1} = \frac{\partial L}{\partial z_2} \cdot \frac{\partial z_2}{\partial h_1} = 2r\,w\,\mathbf{1}[z_2 > 0] \cdot W_2$$
$$\frac{\partial L}{\partial z_1} = \frac{\partial L}{\partial h_1} \cdot \frac{\partial h_1}{\partial z_1} = 2r\,w\,\mathbf{1}[z_2 > 0]\,W_2 \cdot \mathbf{1}[z_1 > 0]$$
$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial z_1} \cdot \frac{\partial z_1}{\partial W_1} = 2r\,w\,\mathbf{1}[z_2 > 0]\,W_2\,\mathbf{1}[z_1 > 0] \cdot \phi(x)$$

In each step, only need gradient w.r.t. next layer times one local gradient.

Forward and backward

To simplify notation, we write $g_v = \partial L / \partial v$.

Summarizing computations:

Forward (top to bottom)	Backward (bottom to top)
$z_1 = W_1\,\phi(x)$	$g_{W_1} = g_{z_1} \cdot \phi(x)$
$h_1 = \mathrm{ReLU}(z_1)$	$g_{z_1} = g_{h_1} \cdot \mathbf{1}[z_1 > 0]$
$z_2 = W_2\,h_1$	$g_{h_1} = g_{z_2} \cdot W_2, \quad g_{W_2} = g_{z_2} \cdot h_1$
$h_2 = \mathrm{ReLU}(z_2)$	$g_{z_2} = g_{h_2} \cdot \mathbf{1}[z_2 > 0]$
$s = w \cdot h_2$	$g_{h_2} = g_s \cdot w, \quad g_w = g_s \cdot h_2$
$r = s - y$	$g_s = g_r \cdot 1$
$L = r^2$	$g_r = 2r$

Backward pass costs roughly as much as forward pass.

Note that **training** needs keeping intermediate activations, so needs much more memory!

For **inference**, forward pass only, we can potentially discard as we compute

For vectors and matrices: generalize from univariate $\rightarrow$ matrix calculus

Most formulas are very similar (with transposes, element-wise products, etc).

Can derive them thinking about each parameter individually

Computation graphs

Feed-forward neural net was just a chain of simple operations

In general, computation of the loss can form a graph (DAG).

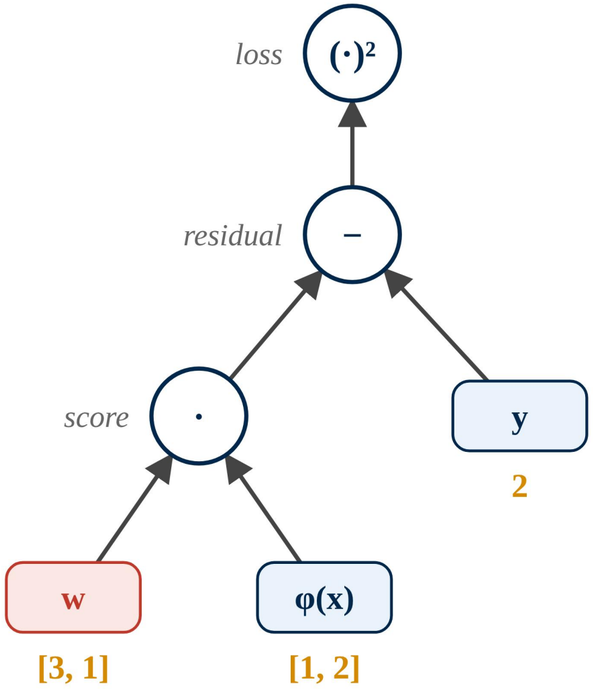
Each node computes one value from its inputs.

E.g., $\text{Loss}(x, y, w) = (w \cdot \phi(x) - y)^2$:

- score $= w \cdot \phi(x)$
- residual $=$ score $- y$
- loss $=$ residual^2

Each operation is simple, so we know its derivatives.

Unlike in a chain, a value can feed several nodes.



Building blocks

Q: What are the derivatives of each operation?

Node f	$\partial f/\partial a$	$\partial f/\partial b$
$a+b$	1	1
$a-b$	1	-1
$a \cdot b$	b	a
a^2	$2a$	
$\max(a,0)$	$\mathbf{1}[a>0]$	
$\sigma(a)=\frac{1}{1+e^{-a}}$	$\sigma(a)(1-\sigma(a))$	

Same for vectors: $f = a \cdot b$ with $a, b \in \mathbb{R}^d$ gives $\nabla_a f = b$.

Losses and complex layers are built out of these simple blocks.

The chain rule on a graph

Say the loss depends on a only through b:

$$\frac{\partial \text{loss}}{\partial a} = \frac{\partial \text{loss}}{\partial b} \cdot \frac{\partial b}{\partial a}$$

What if several nodes use a? Add up over all uses:

$$\frac{\partial \text{loss}}{\partial a} = \sum_{\text{edges } a \to b} \frac{\partial \text{loss}}{\partial b} \cdot \frac{\partial b}{\partial a}$$

Check: $f = a \cdot a$. Two edges from a, each with local derivative a.

Sum: $a + a = 2a$. Same as the derivative of a^2 ✓

So: if we know $\partial \text{loss} / \partial b$ for every b that uses a, we get $\partial \text{loss} / \partial a$.

Start at the loss ($\partial \text{loss} / \partial \text{loss} = 1$), and go backwards in reverse topological order.

Backpropagation

f_i: value of node i. $g_i = \partial \text{loss} / \partial f_i$.

```
function BACKPROP(graph, parameters):
    # Forward pass
    for each node i (topological order):
        f_i = compute value of node i using value of inputs
    # Backward pass
    g_loss = 1
    for each node i (reverse topological order):
        g_i = Σ over edges i → j of  g_j · ∂f_j/∂f_i
    return g_i for all parameters i
```

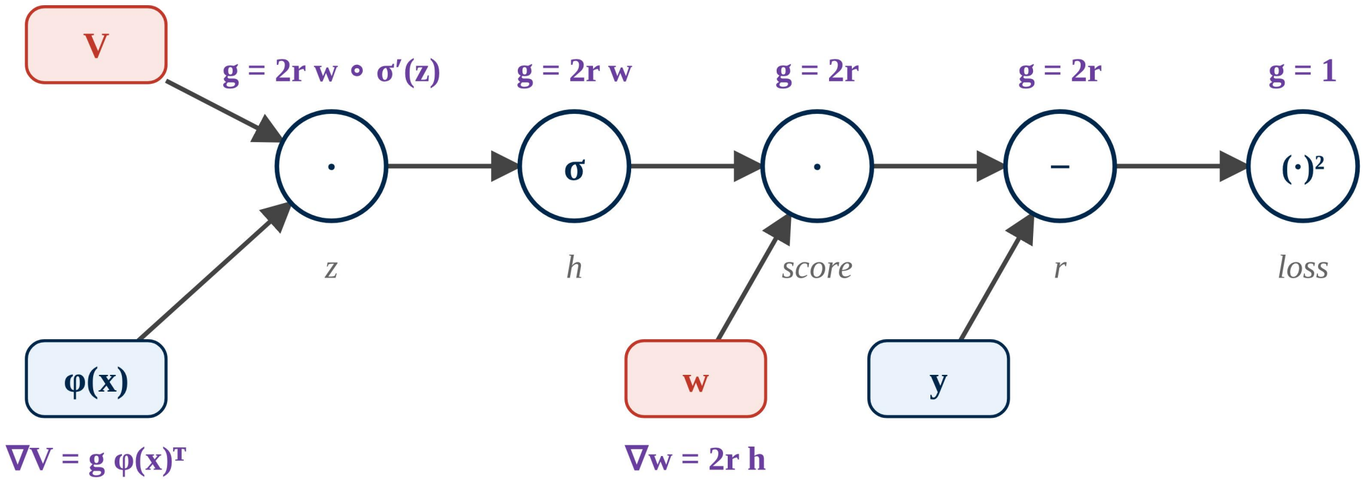
Forward pass: compute and save every node's value.

Backward pass: the local derivatives $\partial f_j / \partial f_i$ use the saved values.

Each edge is visited once in each direction.

Popularized for neural nets by Rumelhart, Hinton & Williams (1986).

Backpropagation on a two-layer network



Back from the loss, one block at a time (r is the residual):

- $(\cdot)^2$, then $-$: $g = 2r$ at the residual and at the score.
- $w \cdot h$: $\nabla_w = 2r\,h$, and $g = 2r\,w$ at h.
- σ, elementwise: multiply by $\sigma'(z)$.
- $V\phi(x)$: $\nabla_V = g_z\,\phi(x)^\top$.

Same idea generalizes to deep networks with many more layers.

Gradient checking

Another way to get a gradient: nudge each parameter, compute the loss and see how it changes.

Notation note: we sometimes imagine all parameters in a single vector θ

Suppose e_i is a "one-hot" vector for parameter at index i

Then, approximately:

$$\frac{\partial \text{Loss}}{\partial \theta_i} \approx \frac{\text{Loss}(\theta + \epsilon e_i) - \text{Loss}(\theta)}{\epsilon}$$

Q: What does this cost to compute all gradients?

A: One forward pass *per parameter* in θ

Too expensive for training, but easy to implement correctly

Useful for testing a backprop implementation (we sample some parameters)

Automatic differentiation

From backprop to autograd

Backprop needs the computation graph + local gradients

Can we get both automatically?

Autograd libraries (e.g., PyTorch, ...) let you write just the forward pass as a program

They then can both do backward pass and , and record it as the program runs.

How? Each operation on a "gradient-aware" value creates a node (w/ value, inputs and local derivatives)

`loss.backward()` runs the backward pass on the computation graph.

Loops, ifs, function calls all work fine: graph is built dynamically.

Non-differentiable points (ReLU at 0): just pick a value, e.g. 0. Extremely unlikely in practice.

Live coding

Let's build a tiny autograd library and train a ReLU neural net!

Inspired by Andrej Karpathy's awesome micrograd

Autograd in PyTorch

Same idea, on tensors (arrays), on GPUs:

```
1   model = nn.Sequential(nn.Linear(d, k), nn.ReLU(), nn.Linear(k, 1))
2   opt = torch.optim.SGD(model.parameters(), lr=0.1)
3   for x, y in batches:
4      loss = ((model(x) - y) ** 2).mean()
5      opt.zero_grad()     # .grad accumulates (+=), so reset it
6      loss.backward()     # backprop: fills p.grad for every parameter p
7      opt.step()          # θ ← θ − η ∇
```

`backward()` computes gradients.

`opt.step()` updates parameters.

Here using `torch.optim.SGD`.

Let's see other better optimizers for training deep neural networks.

Momentum and adaptive steps

Optimizing neural networks is harder

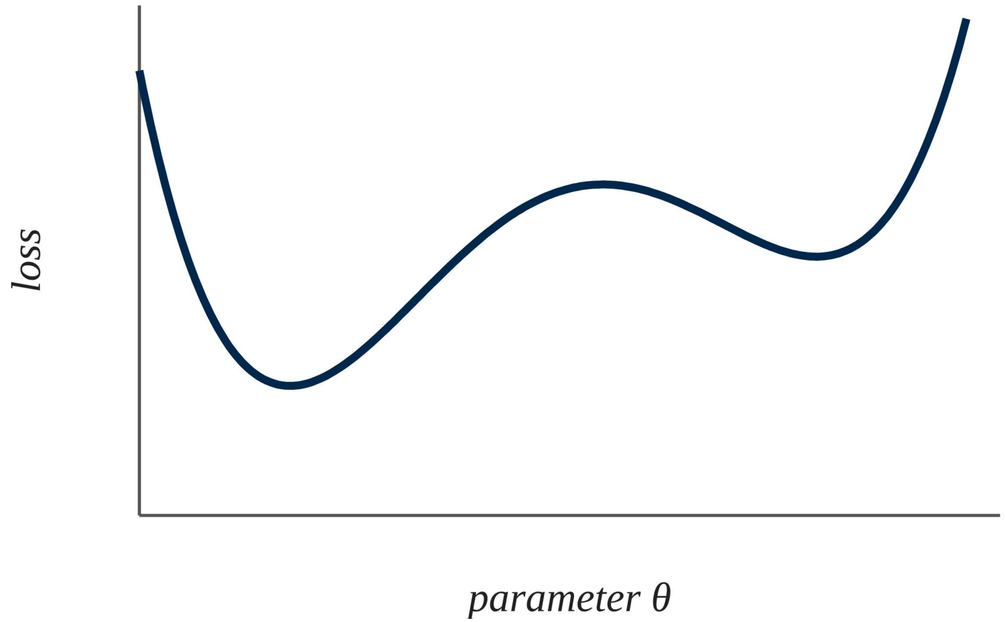
Linear regression: convex loss

So: unique global minimum (no local minima)

Gradient is 0 only at global minimum (no saddle points).

Neural networks: **not convex**. Local minima, saddle points, flat regions, narrow valleys.

What can go wrong with $\theta \leftarrow \theta - \eta g$, and how to improve?



Minibatch SGD

Recall: θ is all the parameters

From now on, we'll always do minibatch: g_t will be average gradient on a small batch of examples.

```
function MINIBATCH-SGD(D_train, Loss, η, B, T):
    θ = random initialization
    for t = 1, ..., T:
        sample a minibatch of B examples from D_train
        g_t = (1/B) Σ over the minibatch of ∇Loss(x, y, θ)
        θ = θ − η g_t
    return θ
```

g_t is a noisy estimate of $\nabla \text{TrainLoss}(\theta)$. Correct on average.

Q: For NNs, we cannot initialize θ with 0 anymore. Why?

A: Hidden units in a layer would compute the same thing, get the same gradient, and stay identical forever.

We'll see initialization in the next lecture (quite subtle in deep learning!)

A narrow valley

Recall GD on this "narrow valley" loss landscape: GD overshoots back and forth.

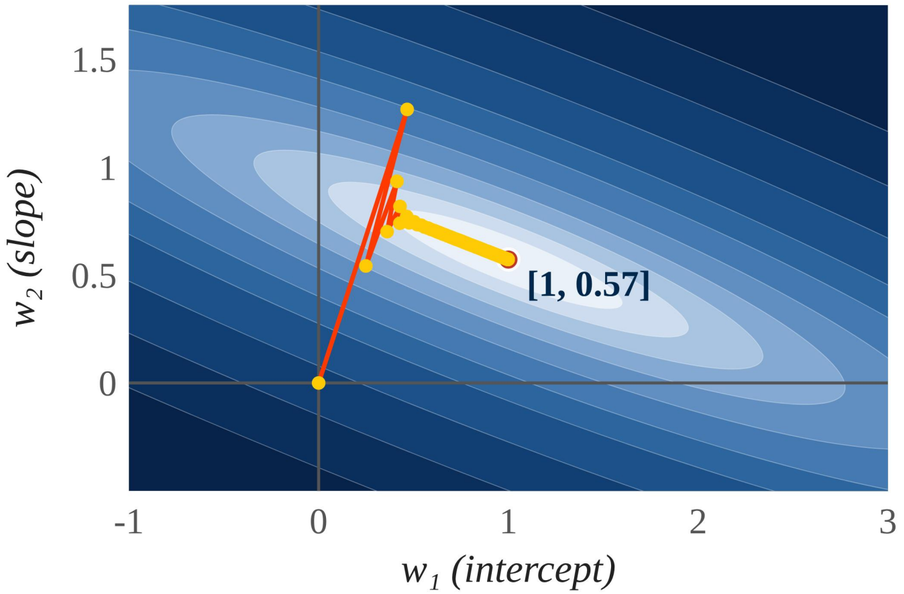
General problem: loss can be much more sensitive to a few directions than others.

The steep direction limits η: too large η will diverge.

But with small η, progress along the flat direction is slow.

So GD zigzags in sensitive directions, and is slow on the others

Idea: **average recent gradients!** Flips will "cancel out", the more consistent directions "add up".



Momentum

Keep a running average u_t of the gradients, and step on that direction instead:

$$u_t = \beta\, u_{t-1} + (1-\beta)\, g_t, \qquad u_0 = 0$$

$$\theta_t = \theta_{t-1} - \eta\, u_t$$

$\beta \in [0, 1)$. Typical: $\beta = 0.9$.

Q: What do we get if we use $\beta = 0$?

A: Vanilla SGD.

Physical intuition: a ball rolling downhill, building up velocity.

This is the **SGD + Momentum** optimizer

B. T. Polyak, Some methods of speeding up the convergence of iteration methods. USSR Computational Mathematics and Mathematical Physics (1964).

Exponential moving average

Unrolling $u_t = \beta\, u_{t-1} + (1-\beta)\, g_t$:

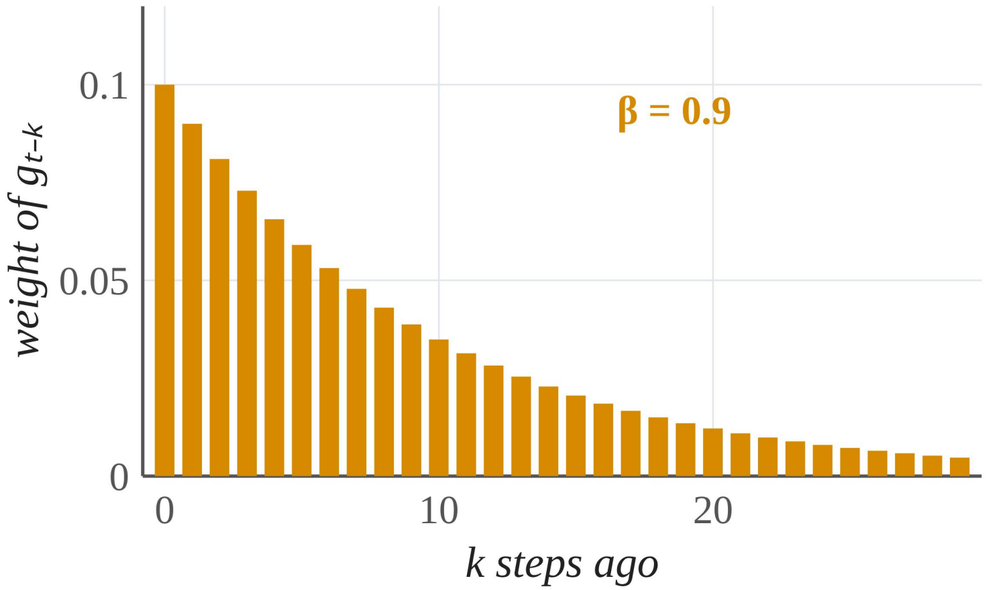
$$u_1 = (1-\beta)\, g_1$$

$$u_2 = \beta(1-\beta)\, g_1 + (1-\beta)\, g_2$$

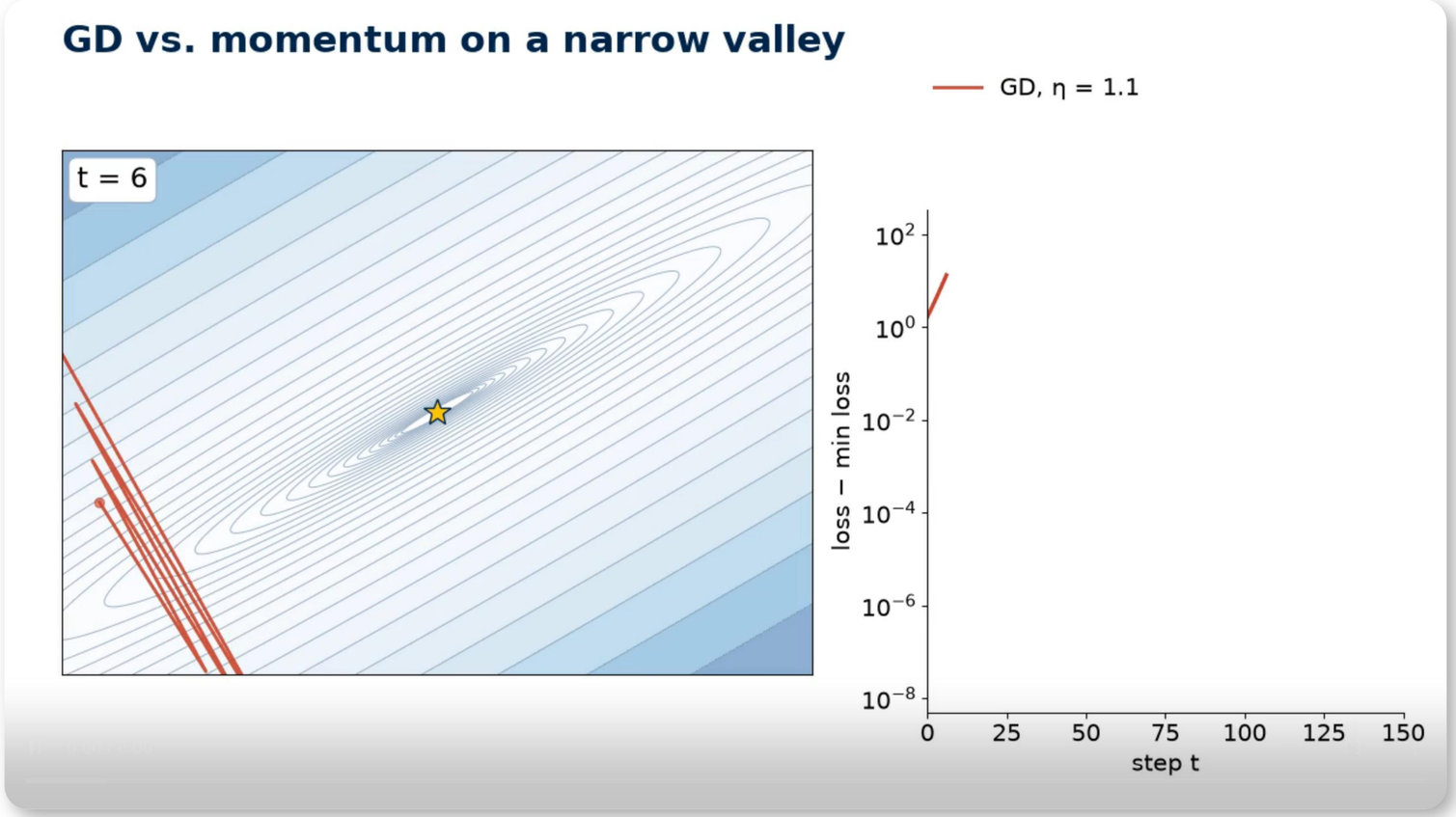
$$u_t = (1-\beta) \sum_{k=0}^{t-1} \beta^k\, g_{t-k}$$

Recent gradients have (geometrically) higher weight than older ones.

Roughly an average of the last $\frac{1}{1-\beta}$ gradients: 10 for $\beta = 0.9$.



Momentum in a narrow valley



GD diverges for $\eta \geq 1$ here.

With $\eta = 0.9$: smooth zigzags, then moves along the valley.

Momentum: the zigzags average out, so we can safely use $\eta = 2$ and go much faster.

Bias correction

$u_t = \beta\, u_{t-1} + (1-\beta)\, g_t$

We started with $u_0 = 0$

Q: With $\beta = 0.9$, what is u_1?

A: $u_1 = 0.1\, g_1$. Only 10% of the gradient!

Unrolling and separating u_0:

$$u_t = (1-\beta) \sum_{k=0}^{t-1} \beta^k\, g_{t-k} \;+\; \beta^t\, u_0$$

The weights add up to 1

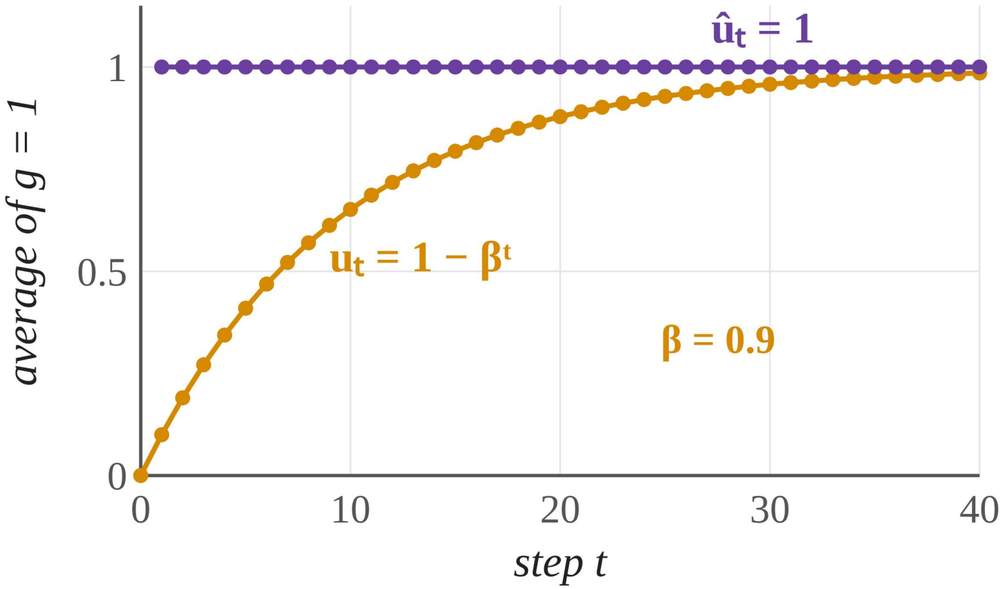
β^t of the weight goes to $u_0 = 0$, a "fake" initialization gradient. Early on, that's most of the weight!

E.g., if every g_t was 1 we'd still have $u_t = 1 - \beta^t$.

Fix: divide by the weight on the real gradients:

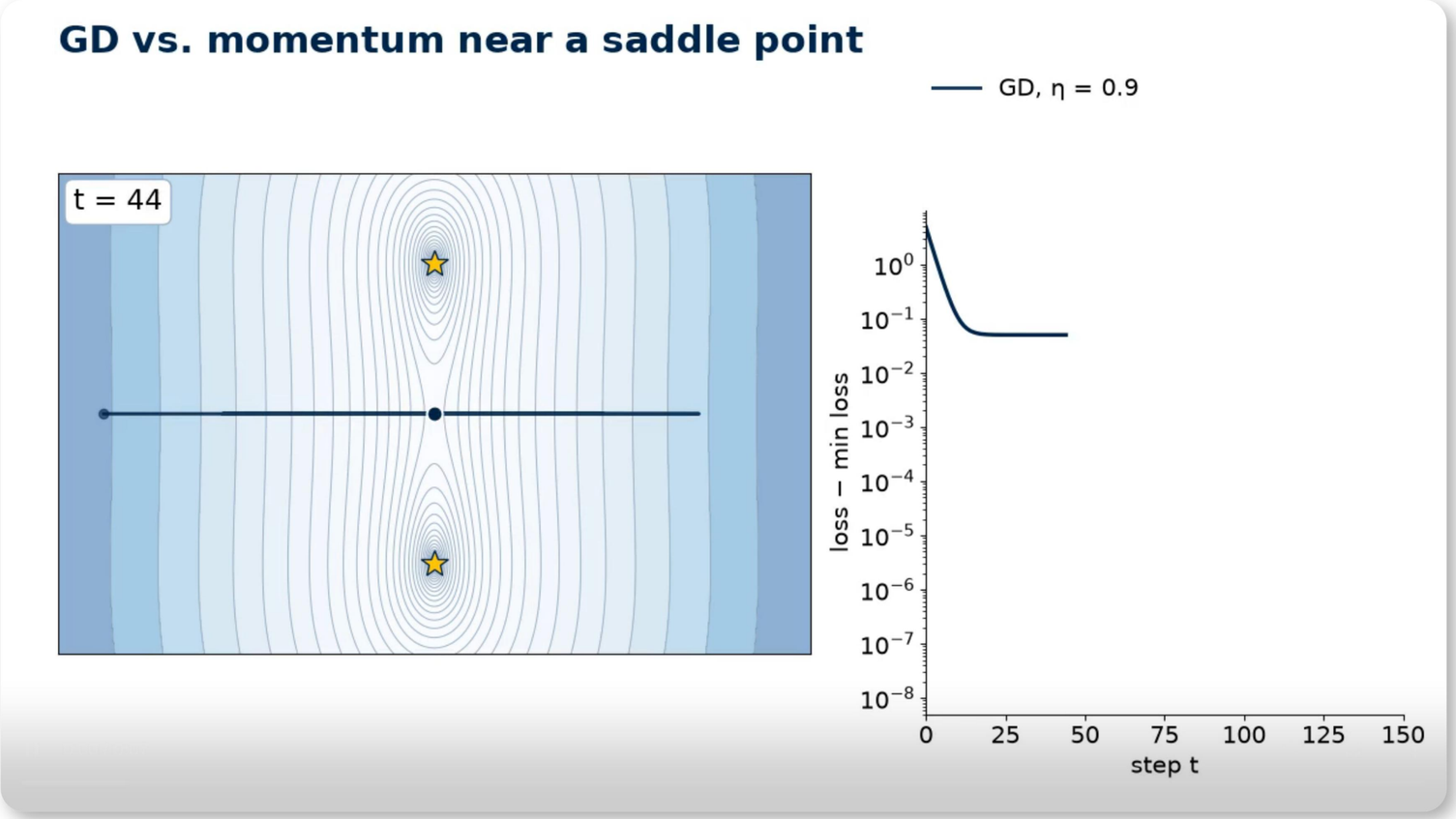
$$\hat{u}_t = \frac{u_t}{1-\beta^t}.$$

$\beta^t \to 0$, so this goes away as we train



Escaping a saddle point

Q: We start almost exactly on the ridge, so the gradient near the saddle is tiny. GD vs. momentum?



Near the saddle, the gradient is almost 0. The little we get in y takes a long time to turn us towards a minimum.

Momentum doesn't help: an average of tiny gradients is still tiny.

In very high dimensions, saddle points are likely a much bigger problem than bad local minima.

Y. Dauphin, R. Pascanu, C. Gulcehre, K. Cho, S. Ganguli, Y. Bengio, Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. NeurIPS (2014).

One step size for every parameter?

Gradient coordinates can have very different sizes: $g_t = [2,\ 100,\ -0.05,\ \ldots]$

One η for all: the "100" makes us use a small η (otherwise we'll diverge)

But then the "0.05" direction barely moves...

This happens in practice:

- Rare features (e.g., rare words): small typical gradients.
- Different layers: average gradients of different sizes.

Idea: scale each coordinate's step by the typical size of *its own* gradient.

At the saddle example earlier: the gradient in the y direction is tiny, but consistent.

It is overtaken by the large gradient in x, but not if we scale it separately

Adaptive step sizes: RMSProp

Idea (RMSProp): keep a running average of *squared* gradients, per coordinate:

$$s_t = \gamma\, s_{t-1} + (1-\gamma)\, g_t^2$$

$$\theta_t = \theta_{t-1} - \eta\, \frac{g_t}{\sqrt{s_t}+\epsilon}$$

Square, root and division are elementwise. $\epsilon \approx 10^{-8}$ avoids dividing by 0.

Q: Suppose a coordinate's gradient is always the same g. How big is its step?

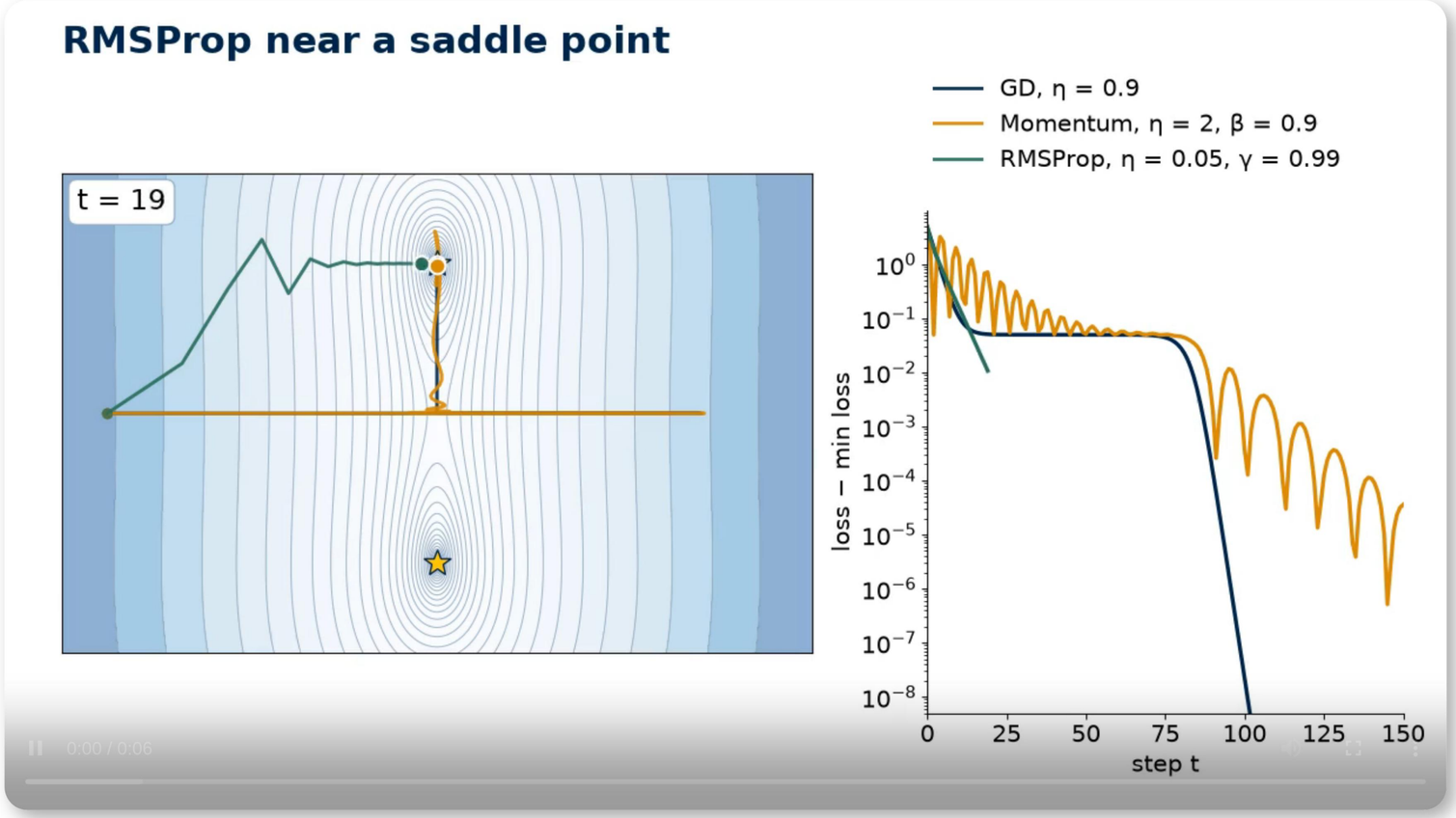
A: $\sqrt{s_t} \approx |g|$, so the step is $\approx \eta \cdot \mathrm{sign}(g)$. Size η, whatever the scale of g!

Adagrad (2011), an earlier variant: sum of all past g^2 instead of an average. Steps only shrink.

T. Tieleman, G. Hinton, Lecture 6.5, RMSProp. Coursera: Neural Networks for Machine Learning (2012).

RMSProp at the saddle

Q: Same saddle. RMSProp: $s_t = \gamma s_{t-1} + (1-\gamma) g_t^2,\ \theta_t = \theta_{t-1} - \eta \frac{g_t}{\sqrt{s_t}+\epsilon}$. What happens?



The gradient in y is tiny, but so is $\sqrt{s_t}$, so the ratio has a similar scale/influence compared to x.

Still "dances around" near the minimum, we'll see how to mitigate that soon.

Adam and beyond

Combining both ideas

	Momentum	RMSProp
Update	$u_t = \beta u_{t-1} + (1-\beta) g_t$ $\theta_t = \theta_{t-1} - \eta\, u_t$	$s_t = \gamma s_{t-1} + (1-\gamma) g_t^2$ $\theta_t = \theta_{t-1} - \eta \frac{g_t}{\sqrt{s_t}+\epsilon}$
Helps with	Zigzags and noise cancel out (e.g., narrow valleys)	Directions with small and large gradients get similar steps (e.g., near saddle points)

Both fix separate issues

Q: Can we have both?

A: Yes Step along the average u_t, and divide by $\sqrt{s_t}$, per coordinate. That's the **Adam** optimizer.

Adam

Momentum (u) + RMSProp (v, before called s), with bias correction for both:

```
function ADAM(η, β₁ = 0.9, β₂ = 0.999, ε = 10⁻⁸):
    u = 0,  v = 0
    for t = 1, ..., T:
        gₜ = minibatch gradient at θ
        u = β₁ u + (1 − β₁) gₜ              # momentum
        v = β₂ v + (1 − β₂) gₜ²             # squared gradients, per coordinate
        û = u / (1 − β₁ᵗ),  v̂ = v / (1 − β₂ᵗ)   # bias correction
        θ = θ − η û / (√v̂ + ε)
```

Note that v needs the same bias correction

Otherwise, for e.g., suppose $v_1 = 0.001\, g_1^2$, and $\sqrt{0.001} \approx 0.03$. Correcting only u, the first step would be about $30\times$ too large.

Typical defaults: $\beta_1 = 0.9$, $\beta_2 = 0.999$.

Still have to tune η.

D. P. Kingma, J. Ba, Adam: A method for stochastic optimization. ICLR (2015).

The optimizer zoo

	Last gradient	Moving average (momentum)
Same step size everywhere	(S)GD	SGD + momentum
Adaptive, per coordinate	Adagrad, RMSProp	Adam

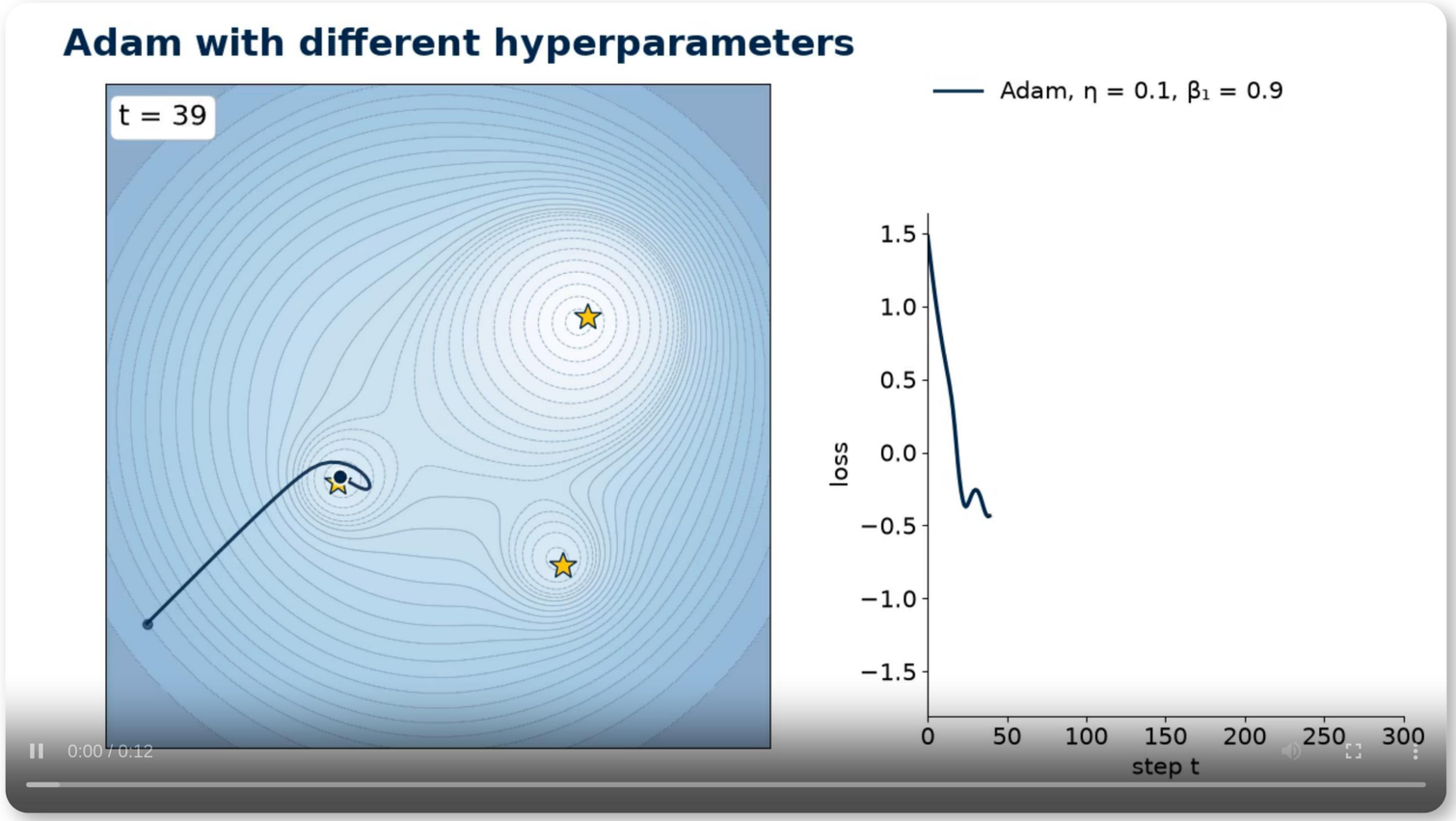
All of them only need g_t. So they work with any model autograd can differentiate.

Adam (and AdamW, next) still typical defaults for deep neural networks.

Hyperparameters matter

Hyperparameters still need to be tuned empirically

Simple example: Adam with three choices of hyperparams:



Here we needed either a larger η or larger momentum to get past the bad first minimum

Weight decay

Recall: penalizing large weights is a form of **regularization**

Effectively smaller hypothesis class (not completely ruling out large weights but biasing agains them).

L2 regularization:

$$\min_\theta \ \text{TrainLoss}(\theta) + \frac{\lambda}{2}\|\theta\|^2$$

The penalty adds $\lambda\theta$ to the gradient. The SGD step becomes:

$$\theta \leftarrow \theta - \eta(g_t + \lambda\theta) = (1 - \eta\lambda)\,\theta - \eta\, g_t$$

Each step "shrinks" the weights by a factor $(1 - \eta\lambda)$: **weight decay**.

AdamW

Can we do that in Adam?

Adam step: $\theta_t = \theta_{t-1} - \eta \frac{\hat{u}_t}{\sqrt{\hat{v}_t}+\epsilon}$. We could add $\lambda\theta$ to g_t, like with SGD.

Q: What happens to $\lambda\theta$ then?

A: It goes into u and v, and gets divided by $\sqrt{\hat{v}}$. Weights with large gradients decay less.

That was not the idea!

Fix: **AdamW optimizer:** just keep the decay separate from u and v:

$$\theta_t = \theta_{t-1} - \eta \left(\frac{\hat{u}_t}{\sqrt{\hat{v}_t}+\epsilon} + \lambda\, \theta_{t-1} \right)$$

Standard optimizer for training Transformer models!

I. Loshchilov, F. Hutter, Decoupled weight decay regularization. ICLR (2019).

Muon: normalizing whole matrices

Even fancier optimizers are a very active research area!

One recent idea: Adam normalizes each coordinate separately. But most parameters are matrices: W for one layer.

Muon: take the momentum M of W (also a matrix), and its SVD $M = U\Sigma V^\top$:

$$W \leftarrow W - \eta UV^\top$$

All singular values set to 1: every direction of the update gets the same size.

The matrix version of Adam's $\mathrm{sign}(g)$ step.

SVD is hard to compute. Muon approximates $UV^\top$ with a few Newton–Schulz iterations (only matrix products).

Recently used to train large LLMs (e.g., Kimi K2), with AdamW for the other (non-matrix) parameters.

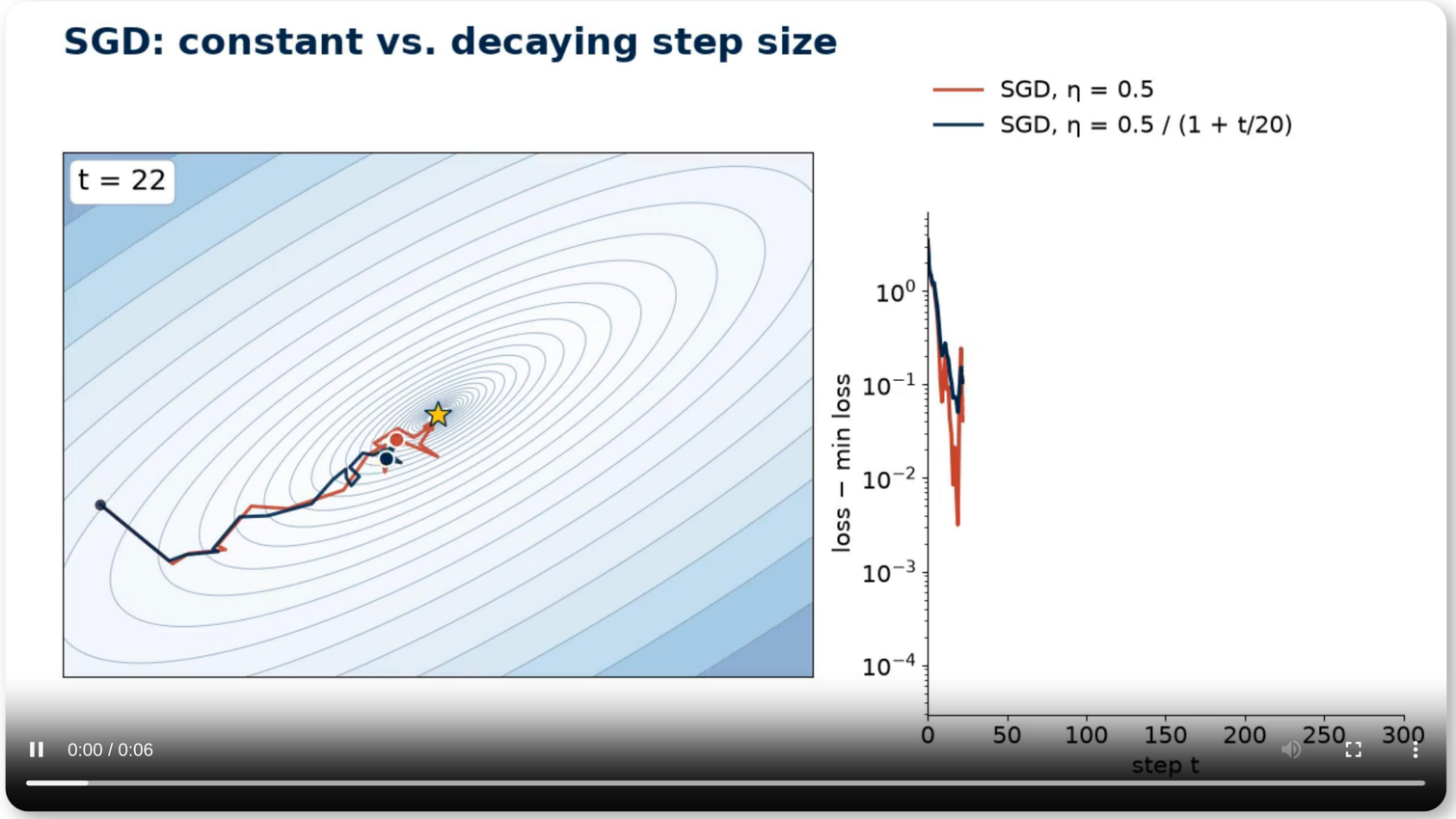
K. Jordan et al., Muon: An optimizer for hidden layers in neural networks (2024).

Learning rate schedule

With a constant η, we can get close to a minimum and then "bounce around"

One solution: decay learning rate during training

Example with vanilla SGD:



Step-size schedules

Idea: make η_t change during training based on iteration t.
Common "schedule": **warmup**, then **decay**.

Warmup, for $t \le T_w$: $\eta_t = \eta_{\max} \frac{t}{T_w}$

Then, with $p = \frac{t - T_w}{T - T_w}$ going from 0 to 1:

$$\text{Cosine: } \eta_t = \eta_{\min} + (\eta_{\max} - \eta_{\min}) \frac{1 + \cos(\pi p)}{2}$$

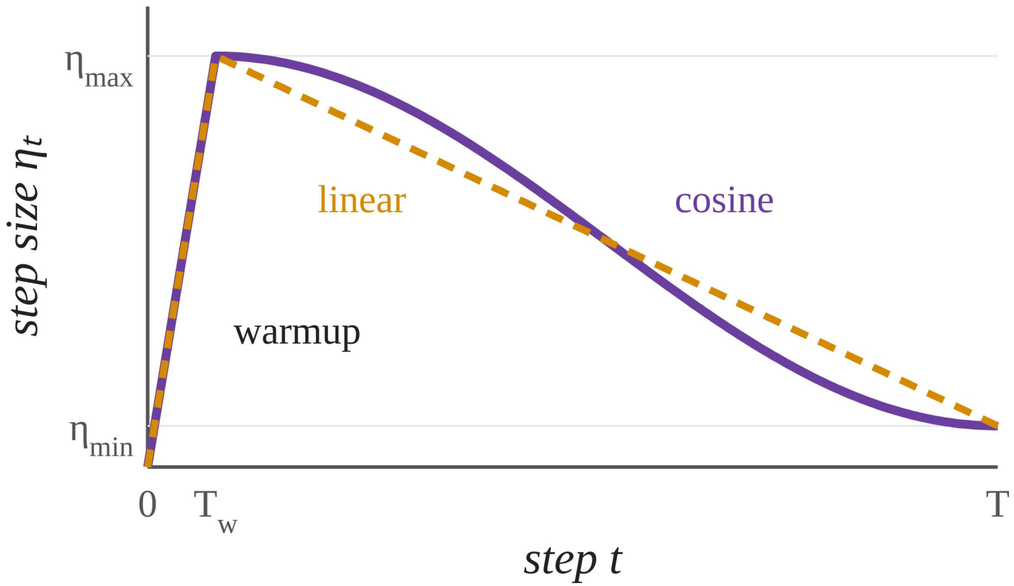
$$\text{Linear: } \eta_t = \eta_{\min} + (\eta_{\max} - \eta_{\min})(1 - p)$$

Warmup: early gradients are large and erratic, and Adam's v is still noisy.

Decay: big steps to make progress, small steps to settle.

There is a big bag of tricks used to train deep neural nets!

Choices are often very much based on experience + experiments



Summary

- **Backpropagation:** the forward pass computes values. The backward pass computes $\partial \text{loss} / \partial f_i$ for every node, one block at a time.
- **Autograd** records the graph as the program runs. All gradients for about the cost of the loss.
- **Momentum:** average recent gradients. Zigzags cancel, consistent directions add up.
- **RMSProp:** scale each coordinate's step by its typical gradient size.
- **Adam** = both, plus bias correction. **AdamW** decouples weight decay. **Muon** normalizes whole matrices.

Next: initialization, then neural network architectures.