DDA4220 · Fall 2026

Lecture 3

Deep Learning and Applications

Training neural networks I: gradients and optimizers

Zhen Liu

CUHK-Shenzhen

Lecture outline

01 Objective and gradient descent

02 Minibatches and backpropagation

03 Learning rate and loss geometry

04 Momentum, Adam and AdamW

* Optional

Parameterizing a network

Fix the architecture. Vary its weights and biases.

$$\theta = \{W^{(1)}, b^{(1)}, \ldots, W^{(L)}, b^{(L)}\}$$

Parameters **Predictor**

$$\theta \longmapsto f_\theta$$

$$\mathcal{H} = \{f_\theta : \theta \in \Theta\}$$

Hypothesis class: functions represented by this architecture.

Empirical risk minimization

Fixed training data $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$

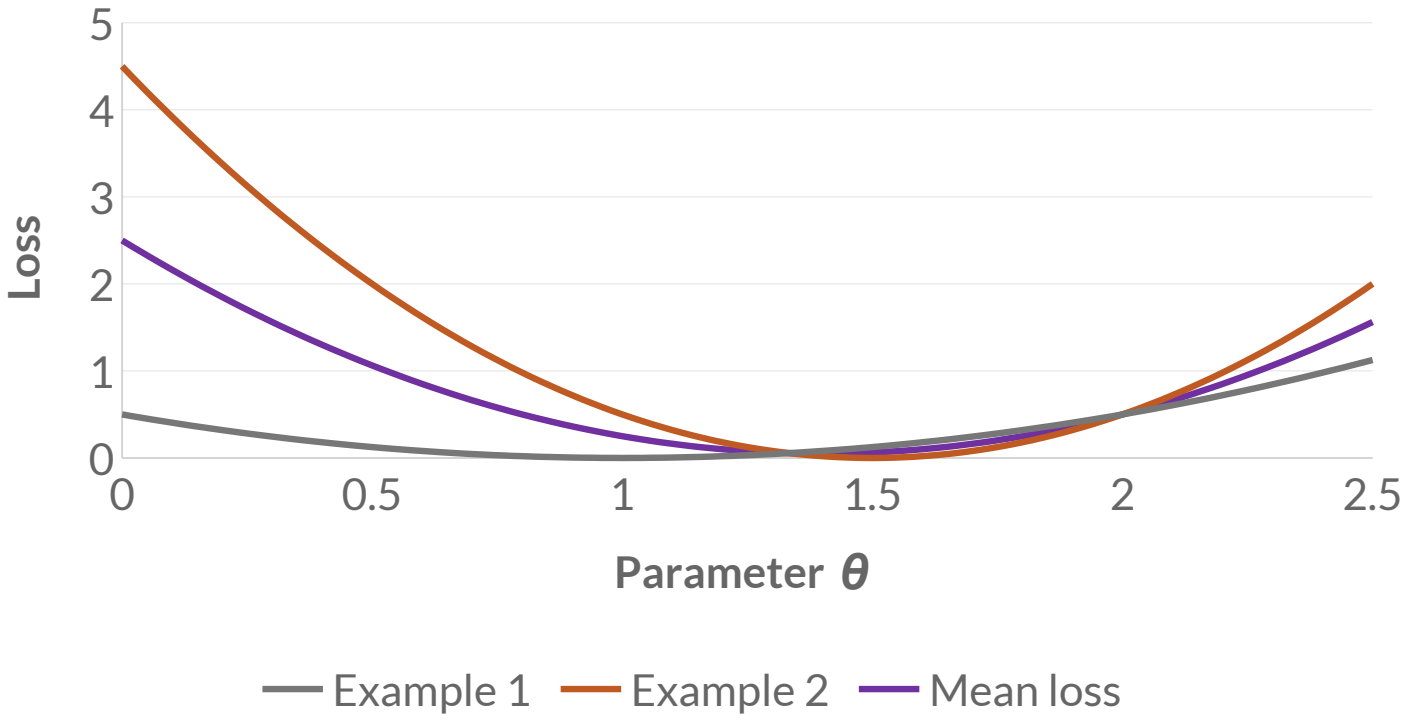
Loss on one example $\ell_i(\theta) = \ell(f_\theta(x_i), y_i)$

Average training loss $\mathcal{L}_\mathcal{D}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \ell_i(\theta)$

Minimize over parameters $\min_{\theta \in \Theta} \mathcal{L}_\mathcal{D}(\theta)$

Loss as a function of parameters

$$f_\theta(x) = \theta x, \qquad (x_1, y_1) = (1, 1), \quad (x_2, y_2) = (2, 3)$$



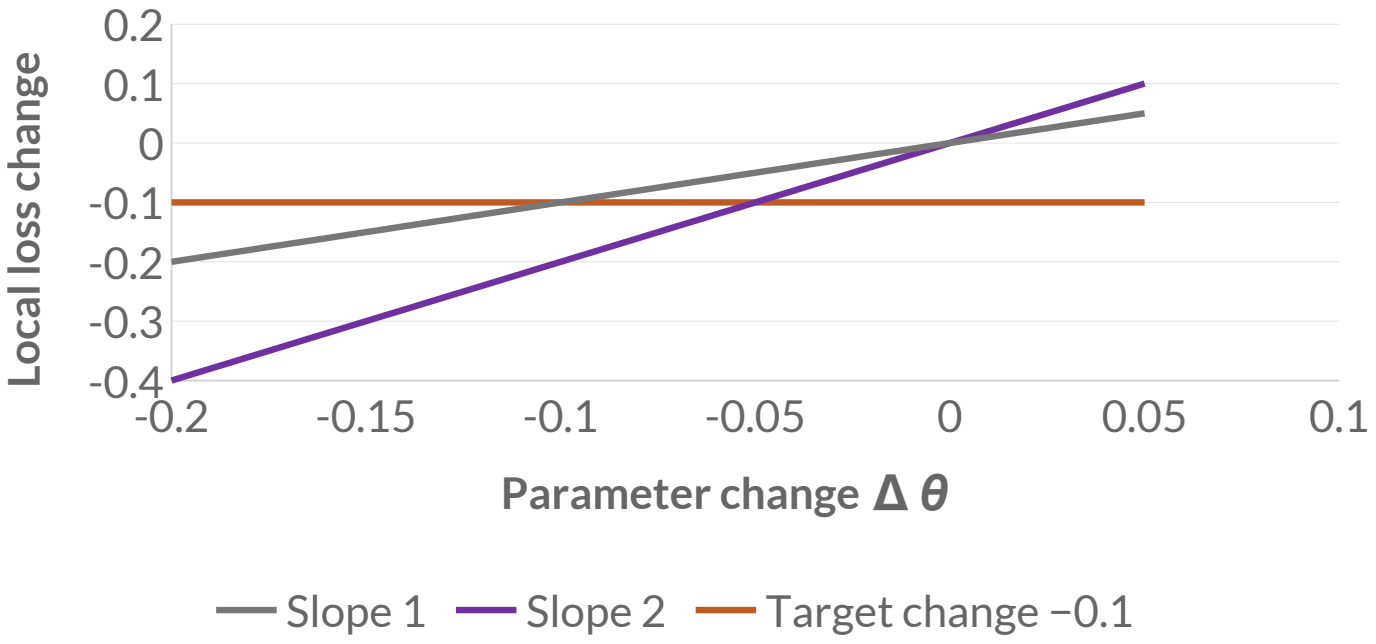
$$\ell_1(\theta) = \tfrac{1}{2}(\theta - 1)^2$$

$$\ell_2(\theta) = \tfrac{1}{2}(2\theta - 3)^2$$

$$\mathcal{L} = \tfrac{1}{2}(\ell_1 + \ell_2)$$

Slope and loss change

$$\Delta\mathcal{L} \approx g\,\Delta\theta, \qquad g = \frac{d\mathcal{L}}{d\theta}$$



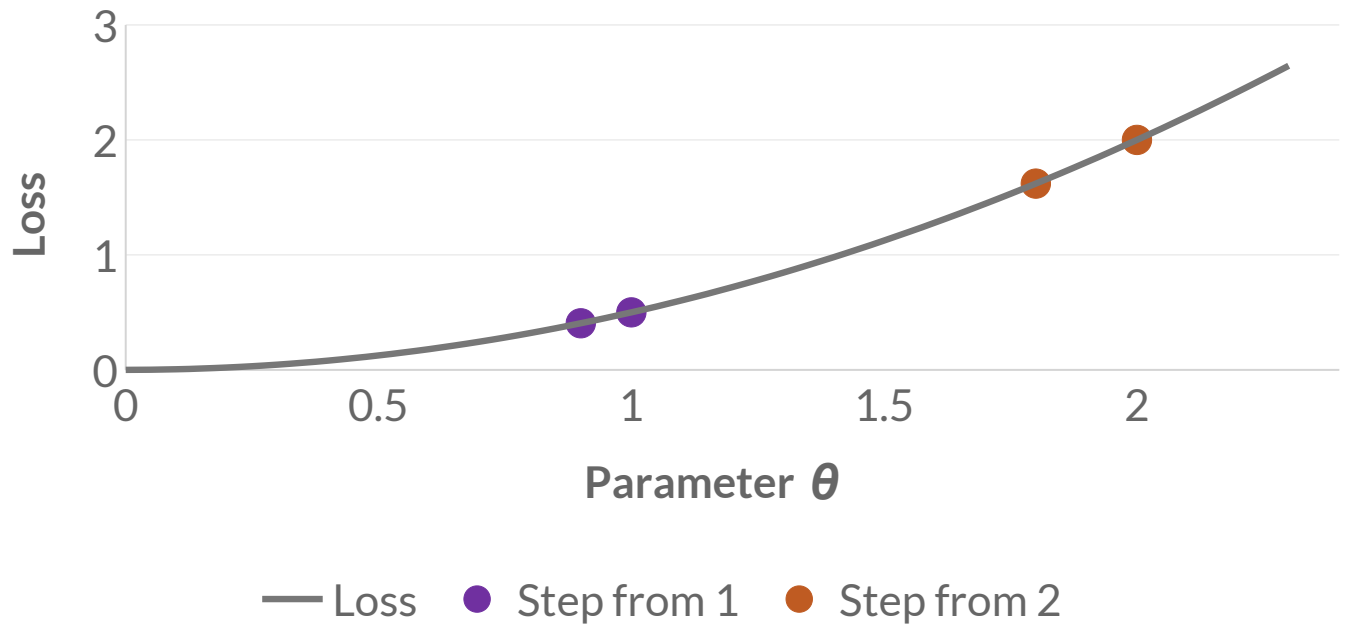
Loss decrease: 0.1

$$g = 1: \quad \Delta\theta \approx -0.1$$

$$g = 2: \quad \Delta\theta \approx -0.05$$

Gradient descent

$$\theta_{t+1} = \theta_t - \eta g_t, \qquad \Delta\theta = -\eta g_t$$



$$\mathcal{L}(\theta) = \tfrac{1}{2}\theta^2, \quad \eta = 0.1$$

$$\theta = 1: \quad g = 1, \quad \Delta\theta = -0.1$$

$$\theta = 2: \quad g = 2, \quad \Delta\theta = -0.2$$

Same learning rate, larger slope, larger parameter step.

Gradients with many parameters

$$g = \nabla_\theta \mathcal{L} = \begin{bmatrix} \partial \mathcal{L} / \partial \theta_1 \\ \vdots \\ \partial \mathcal{L} / \partial \theta_P \end{bmatrix}$$

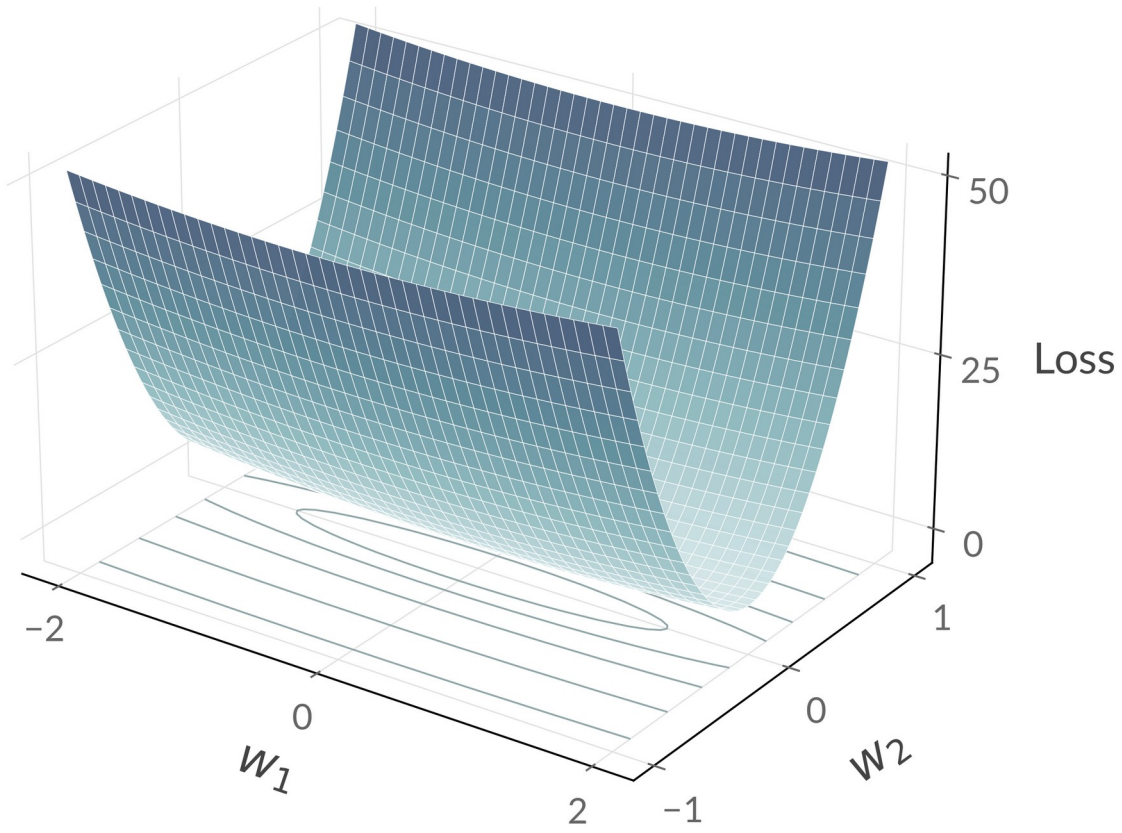
One derivative per parameter.

Each partial derivative holds the other parameters fixed.

$$\Delta \mathcal{L} \approx g^\mathsf{T} \Delta\theta = \sum_{j=1}^{P} g_j \, \Delta\theta_j$$

$$\Delta\theta = -\eta g \quad \Longrightarrow \quad \Delta \mathcal{L} \approx -\eta \|g\|_2^2$$

Loss surface

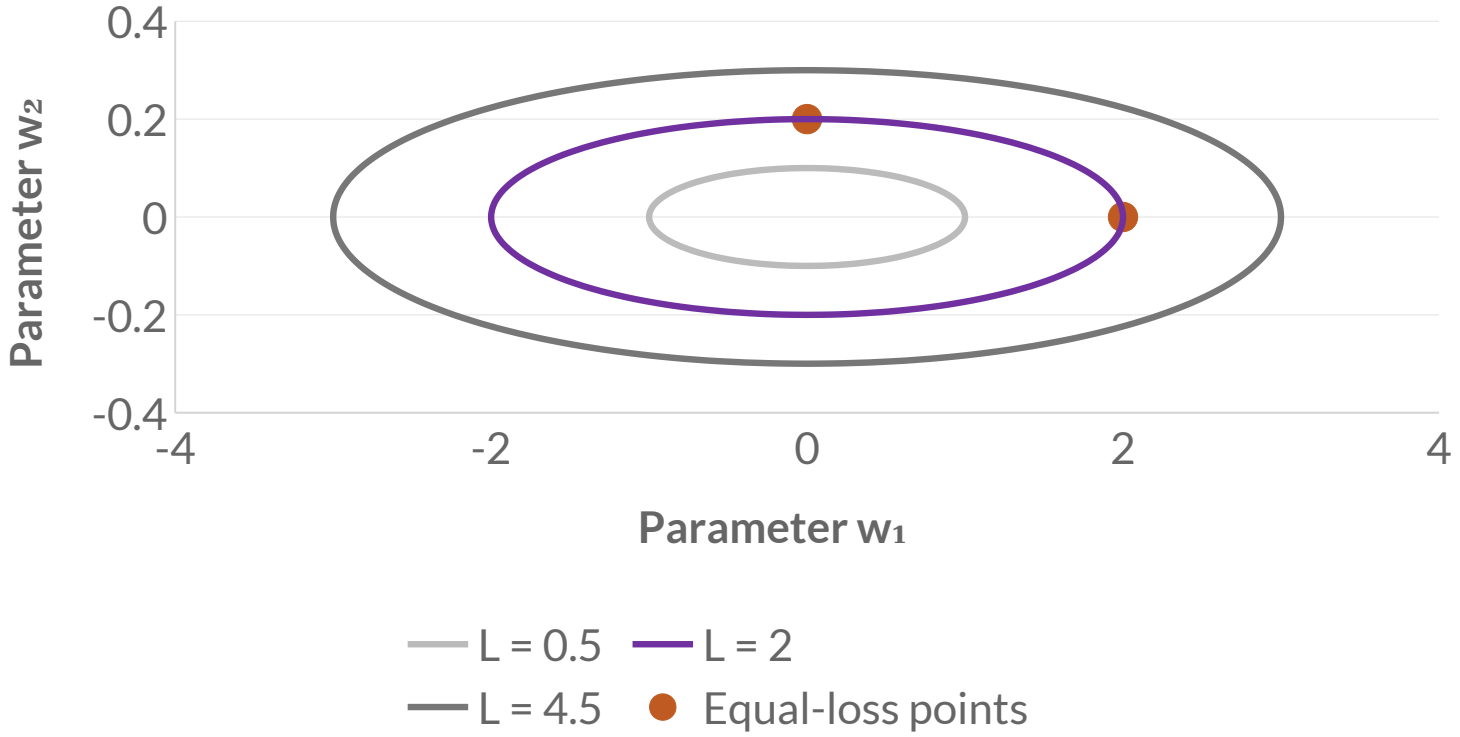


$$\mathcal{L}(w_1, w_2) = \frac{1}{2}(w_1^2 + 100w_2^2)$$

- Horizontal: parameters
- Height: loss

Loss contours

Every point on the same curve has the same loss.



$$\mathcal{L}(w_1, w_2) = \tfrac{1}{2}(w_1^2 + 100w_2^2)$$

$$\mathcal{L}(2, 0) = 2$$

$$\mathcal{L}(0, 0.2) = 2$$

Full-batch gradient descent

Full-batch Gradient Descent

Initialize the network parameters

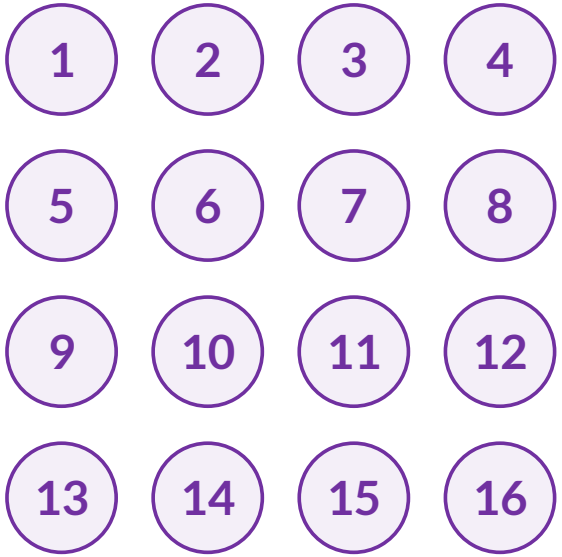
For each update

1	$B_t \leftarrow \{1, \ldots, N\}$	**Input batch**
2	$g_t \leftarrow \frac{1}{\lvert B_t \rvert} \sum_{i \in B_t} \nabla_\theta \ell_i(\theta_t)$	Gradient
3	$d_t \leftarrow g_t$	Update vector
4	$\eta_t \leftarrow \eta$	Learning rate
5	$\theta_{t+1} \leftarrow \theta_t - \eta_t d_t$	

End

Cost of full-batch updates

Full batch: every training example



- Every update waits for all N examples.
- Similar examples can give similar gradients.

$$g_t = \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta \ell_i(\theta_t)$$

1,000,000 examples before one parameter update

Minibatches and parameter updates

Dataset: 16 examples

Highlighted: examples used in step 1

Each step uses the same shared model

Step 1 $\{2, 5, 9, 14\}$

Step 2 $\{1, 7, 9, 16\}$

Update after each selected batch.

Stochastic gradient descent (SGD)

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

$\mathcal{B}$ is the selected batch. $|\mathcal{B}|$ is its number of examples.

$$\theta \leftarrow \theta - \eta g_{\mathcal{B}}$$

$$\mathbb{E}_{\mathcal{B}}[g_{\mathcal{B}}] = \nabla_\theta L \quad \text{under uniform sampling}$$

Averaging over possible batches gives the full-data gradient.

Minibatch SGD

Stochastic Gradient Descent (SGD)

Initialize the network parameters

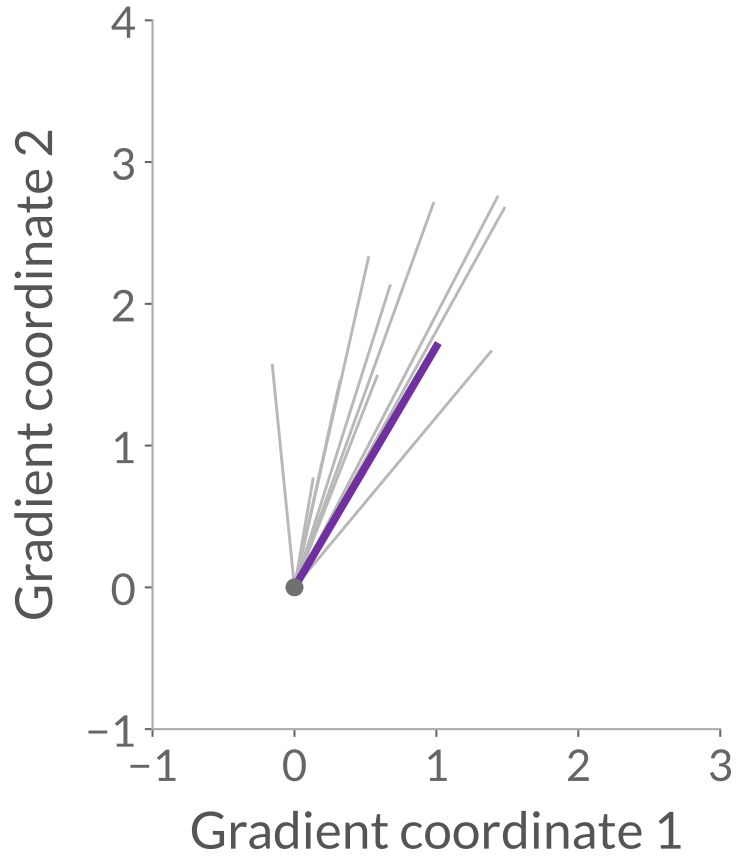
For each update

1	$B_t \leftarrow \text{a sampled minibatch}$	**Input batch**
2	$g_t \leftarrow \frac{1}{\lvert B_t \rvert} \sum_{i \in B_t} \nabla_\theta \ell_i(\theta_t)$	Gradient
3	$d_t \leftarrow g_t$	Update vector
4	$\eta_t \leftarrow \eta$	Learning rate
5	$\theta_{t+1} \leftarrow \theta_t - \eta_t d_t$	

End

Minibatch gradient variability

Purple: mean gradient



$$g_B = \frac{1}{B}\sum_{i=1}^{B} g_i$$

- Parameters stay fixed in this picture.
- Batch gradients vary.

Computing the batch gradient

Stochastic Gradient Descent (SGD)

Initialize the network parameters

For each update

1	$B_t \leftarrow \text{a sampled minibatch}$	Input batch
2	$g_t \leftarrow \frac{1}{\lvert B_t \rvert} \sum_{i \in B_t} \nabla_\theta \ell_i(\theta_t)$	**Gradient**
3	$d_t \leftarrow g_t$	Update vector
4	$\eta_t \leftarrow \eta$	Learning rate
5	$\theta_{t+1} \leftarrow \theta_t - \eta_t d_t$	

End

Computing the gradients

One parameter

$$\ell(w) = \tfrac{1}{2}(wx - y)^2$$

$$\frac{\partial \ell}{\partial w} = (wx - y)x$$

- A short derivative by hand.

Several layers

$$h_1 = \phi(W_1 x + b_1)$$

$$h_2 = \phi(W_2 h_1 + b_2)$$

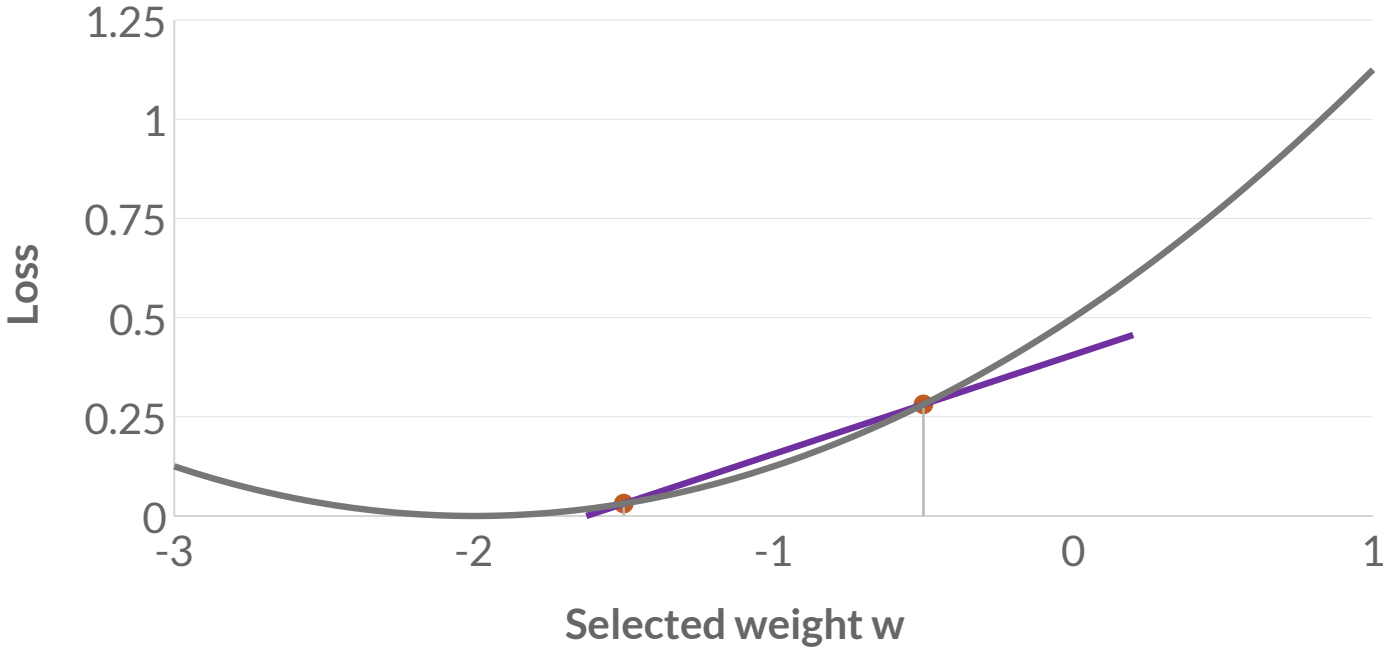
$$\ell = \tfrac{1}{2}(v^\mathsf{T} h_2 + c - y)^2$$

- Longer hand derivations.

Finite differences

$$F'(w) \approx \frac{F(w+\varepsilon) - F(w-\varepsilon)}{2\varepsilon}$$

Orange: two loss evaluations



- Keep the other weights fixed.
- Two forward evaluations for one derivative.

Purple: secant line

Finite differences in many dimensions

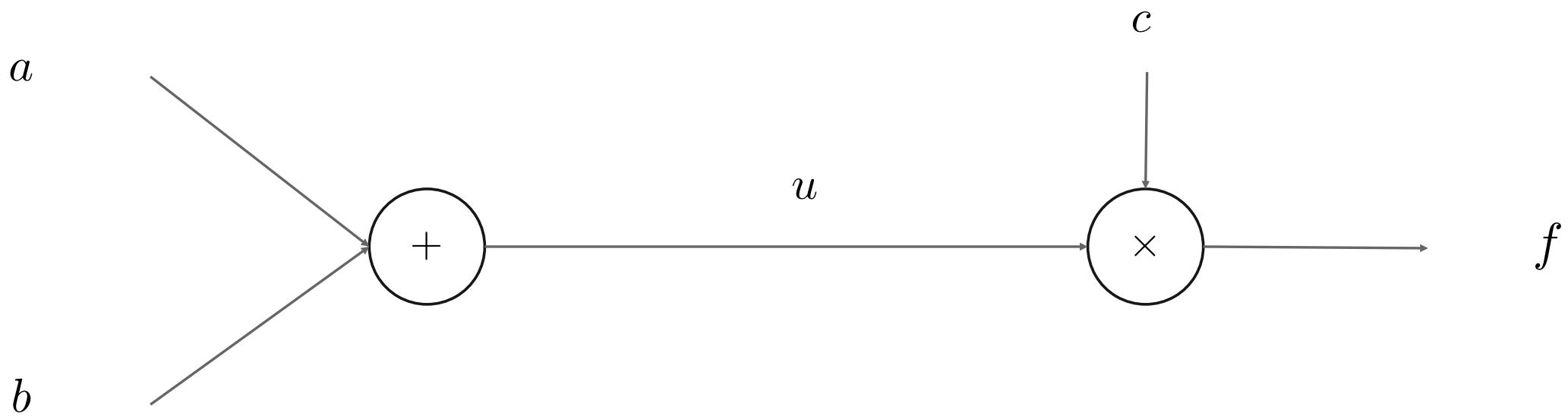
Two forward evaluations for each parameter

Parameter 1	$L(\theta_1 + \varepsilon, \theta_2, \ldots)$	$L(\theta_1 - \varepsilon, \theta_2, \ldots)$
Parameter 2	$L(\theta_1, \theta_2 + \varepsilon, \ldots)$	$L(\theta_1, \theta_2 - \varepsilon, \ldots)$
Parameter P	$L(\ldots, \theta_P + \varepsilon)$	$L(\ldots, \theta_P - \varepsilon)$

$$P = 10^6 \quad \Longrightarrow \quad 2{,}000{,}000 \text{ forward evaluations}$$

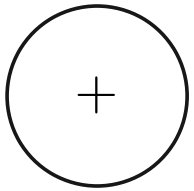
Computation graphs

Circles are operations; arrows carry values.

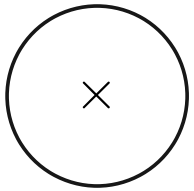


$$u = a + b, \qquad f = uc = (a + b)c$$

Local derivatives and the chain rule



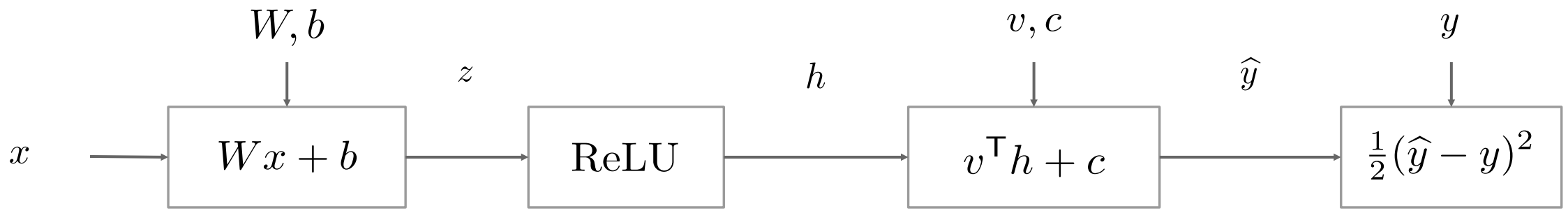
$$u = a + b: \quad \frac{\partial u}{\partial a} = 1, \quad \frac{\partial u}{\partial b} = 1$$



$$f = uc: \quad \frac{\partial f}{\partial u} = c, \quad \frac{\partial f}{\partial c} = u$$

$$\frac{\partial f}{\partial a} = \frac{\partial f}{\partial u}\frac{\partial u}{\partial a}$$

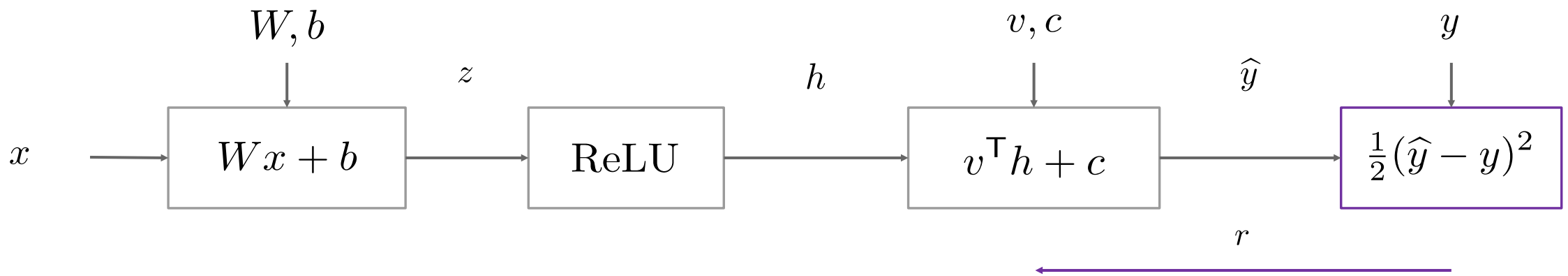
The MLP as a computation graph



Save the intermediate values during the forward pass.

$$z = Wx + b, \qquad h = \mathrm{ReLU}(z), \qquad \widehat{y} = v^\mathsf{T} h + c$$

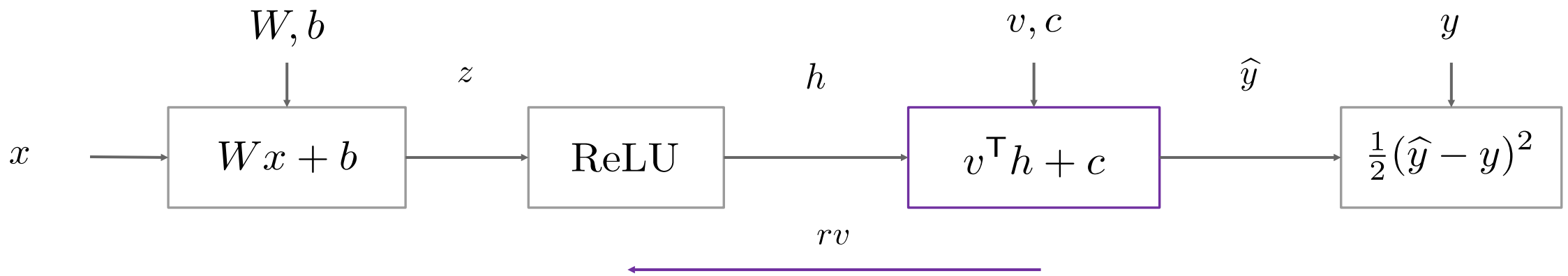
Backward pass: loss derivative



$$\ell = \tfrac{1}{2}(\widehat{y} - y)^2, \qquad r = \frac{\partial \ell}{\partial \widehat{y}} = \widehat{y} - y$$

A positive residual means increasing the prediction increases the loss.

Backward pass: output layer



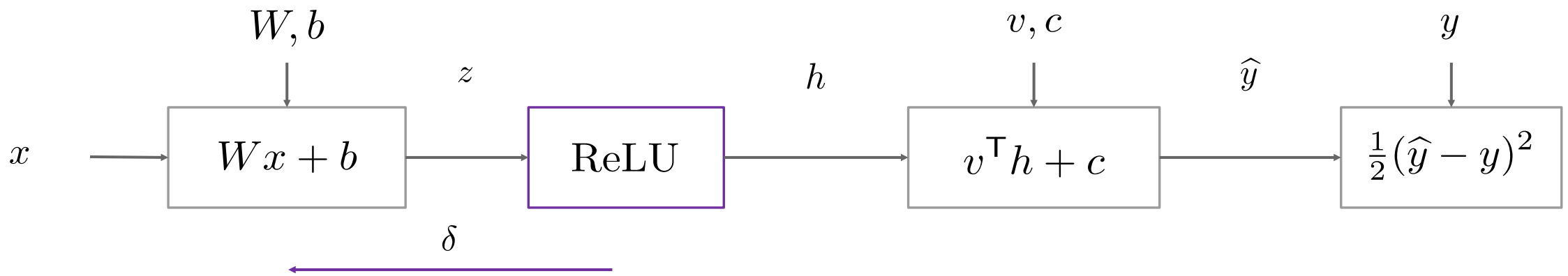
Parameter gradients

$$\nabla_v \ell = rh, \qquad \frac{\partial \ell}{\partial c} = r$$

Passed to the hidden layer

$$\frac{\partial \ell}{\partial h} = rv$$

Backward pass: ReLU



$$\delta = \frac{\partial \ell}{\partial z} = \frac{\partial \ell}{\partial h} \odot \mathbf{1}\{z > 0\}$$

Positive preactivation: pass the incoming derivative.

Negative preactivation: this path contributes zero.

Layerwise gradients in matrix form

a: layer input h: layer output ϕ : activation δ : loss sensitivity at z

$$z = Wa + b, \quad h = \phi(z), \quad \delta = \frac{\partial \ell}{\partial h} \odot \phi\prime(z)$$

Parameter gradients

$$\nabla_W \ell = \delta a^\mathsf{T}, \qquad \nabla_b \ell = \delta$$

Pass to the previous layer

$$\frac{\partial \ell}{\partial a} = W^\mathsf{T} \delta$$

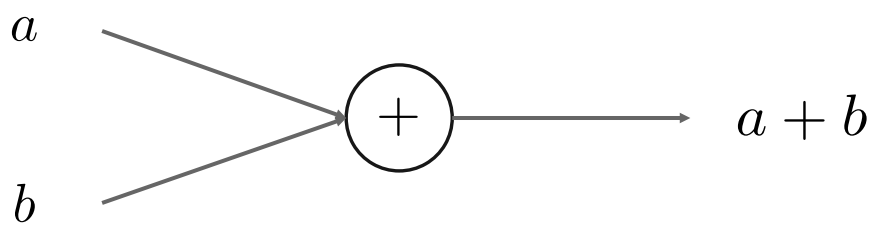
- Weight gradient = output sensitivity × input activation.

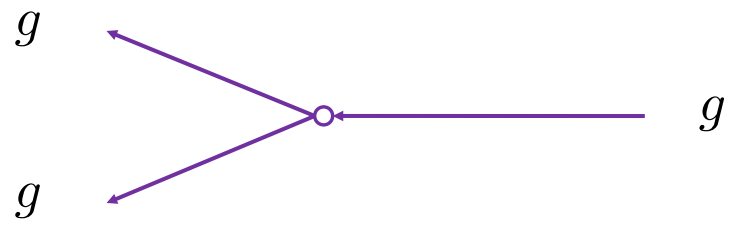
Forward and backward graph operations

Forward

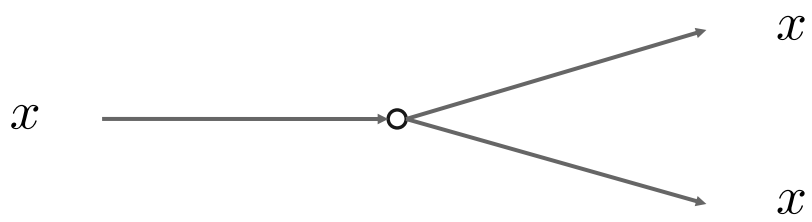
Backward

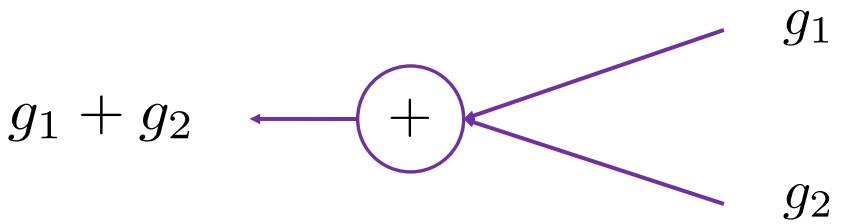
Sum





Copy





Adapted from CS 7643

Gradients for a minibatch

$$\mathcal{L}_B(\theta) = \frac{1}{B}\sum_{i=1}^{B} \ell_i(\theta)$$

$$\nabla_\theta \mathcal{L}_B(\theta) = \frac{1}{B}\sum_{i=1}^{B} \nabla_\theta \ell_i(\theta)$$

Every term uses the same current parameters.

Average the contributions, then make one parameter update.

The same rules for a batch

Forward

$$Z = WA + b$$

Output sensitivity

$$\Delta = \frac{\partial L_{\mathcal{B}}}{\partial Z}$$

Parameter gradients

$$\nabla_W L_{\mathcal{B}} = \Delta A^{\mathsf{T}}$$

Bias gradient: sum columns of Δ

Pass to the previous layer

$$\frac{\partial L_{\mathcal{B}}}{\partial A} = W^{\mathsf{T}} \Delta$$

A contains the B examples as columns. Add the same bias b to every column.

Cross-entropy in logit form

Logits s, target distribution q

$$p_k = \frac{e^{s_k}}{\sum_j e^{s_j}}, \qquad \sum_k q_k = 1$$

$$\ell = -\sum_k q_k \log p_k$$

Substitute softmax and collect terms

$$\ell = \log \sum_j e^{s_j} - \sum_k q_k s_k$$

Cross-entropy gradient

$$\ell = \log \sum_j e^{s_j} - \sum_k q_k s_k$$

$$\frac{\partial \ell}{\partial s_k} = \frac{e^{s_k}}{\sum_j e^{s_j}} - q_k = p_k - q_k$$

One-hot target

$$k = y: \;\; p_y - 1 \leq 0$$

Increase this logit

$$k \neq y: \;\; p_k \geq 0$$

Decrease these logits

Backpropagation algorithm

P: batch predictions Q: batch targets B: batch size

Procedure · Backward pass

Require: Saved forward values and mean batch loss $L_{\mathcal{B}}$

$\Delta^{[L]} \leftarrow \partial L_{\mathcal{B}} / \partial Z^{[L]}$

For softmax + cross-entropy: $\Delta^{[L]} = (P - Q)/B$ — Start at the output

for $l = L, L-1, \ldots, 1$ **do**

  $G_W^{[l]} \leftarrow \Delta^{[l]} (A^{[l-1]})^\top, \quad G_b^{[l]} \leftarrow$ sum columns of $\Delta^{[l]}$ — Parameter gradients

  if $l > 1$ **then**

    $\Delta^{[l-1]} \leftarrow ((W^{[l]})^\top \Delta^{[l]}) \odot \phi'_{l-1}(Z^{[l-1]})$ — Pass to the previous layer

  end if

end for

Forward and reverse mode

Forward mode

	x_1	x_2	x_3	x_4	x_5
f_1		J_{12}			
f_2		J_{22}			
f_3		J_{32}			

Vary input x_2

One column: effects on all outputs

Jacobian

$$J_{ij} = \frac{\partial f_i}{\partial x_j}$$

Reverse mode

	x_1	x_2	x_3	x_4	x_5
f_1					
f_2	J_{21}	J_{22}	J_{23}	J_{24}	J_{25}
f_3					

Differentiate output f_2

One row: derivatives for every input

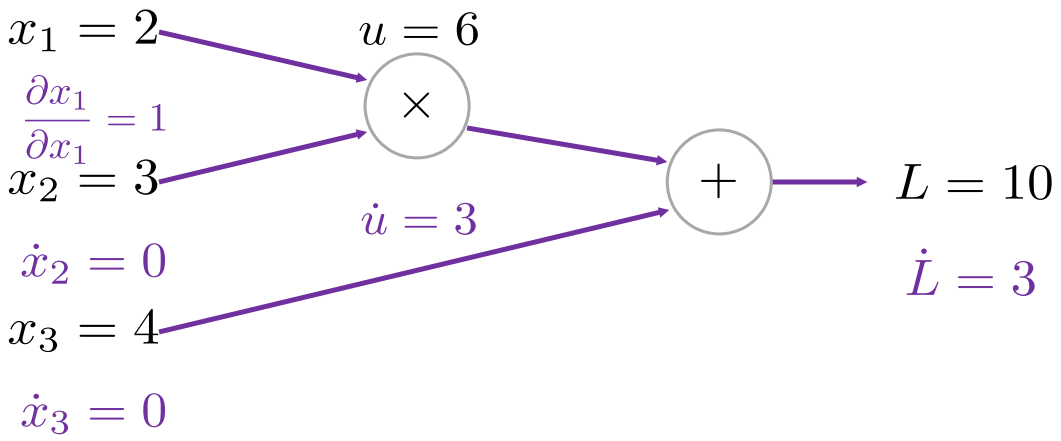
Many parameters, one scalar loss: one reverse pass gives the full gradient.

Many inputs, one output

$$u = x_1 x_2, \quad L = u + x_3, \qquad (x_1, x_2, x_3) = (2, 3, 4)$$

Forward mode

At each node: $\dot{v} = \dfrac{\partial v}{\partial x_1}$

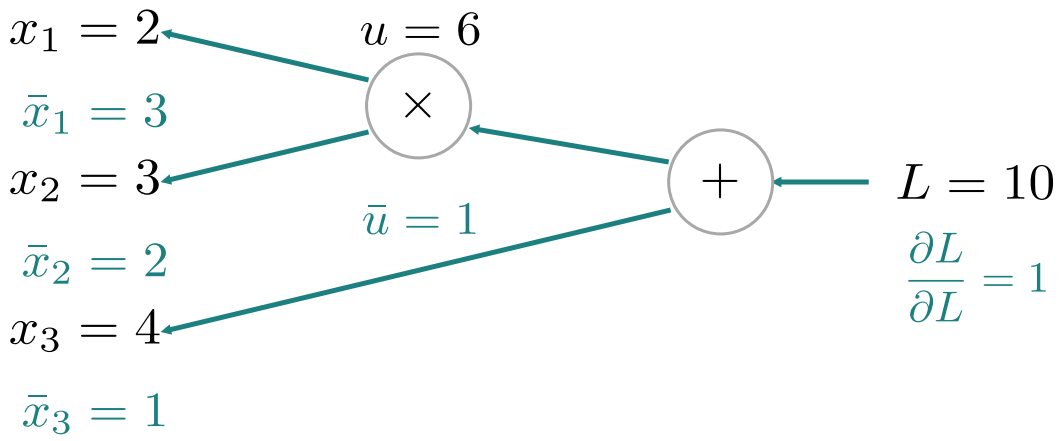


$$\frac{\partial L}{\partial x_1} = 3$$

Repeat for other inputs: 3 passes total

Reverse mode

At each node: $\bar{v} = \dfrac{\partial L}{\partial v}$



$$\nabla_x L = (3, 2, 1)^\mathsf{T}$$

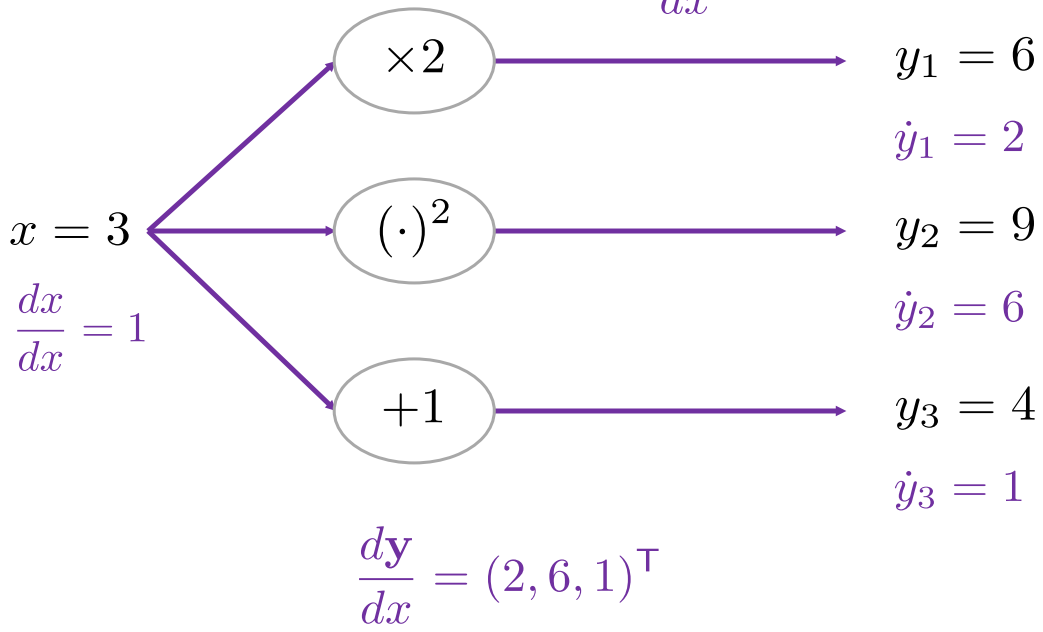
All input derivatives in 1 pass

One input, many outputs

$$\mathbf{y} = (2x,\ x^2,\ x+1)^\mathsf{T}, \qquad x = 3$$

Forward mode

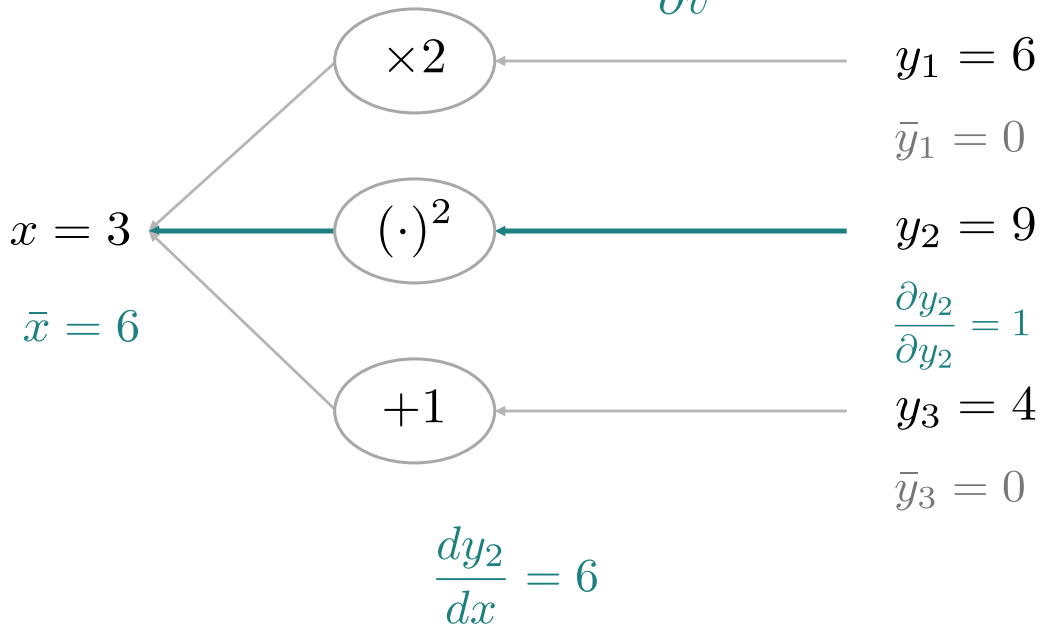
At each node: $\dot{v} = \frac{dv}{dx}$



All output derivatives in 1 pass

Reverse mode

At each node: $\bar{v} = \frac{\partial y_2}{\partial v}$



Repeat for other outputs: 3 passes total

Forward and reverse mode in practice

Reverse mode

Default for neural-network training

$$\underbrace{\theta_1, \ldots, \theta_P}_{\text{many parameters}} \longrightarrow \underbrace{\mathcal{L}}_{\text{one loss}}$$

One backward pass supplies the parameter gradients.

PyTorch: loss.backward()

Forward mode

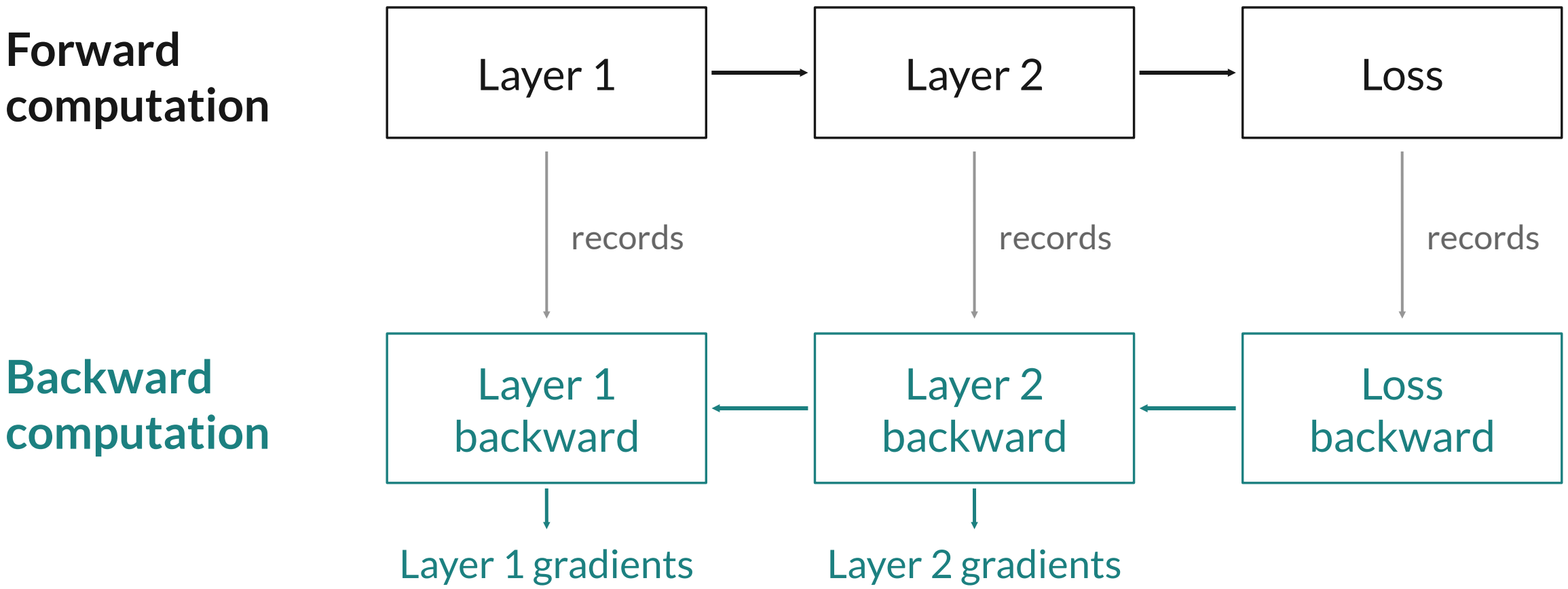
Sensitivities to a few inputs

$$\underbrace{\alpha}_{\text{one parameter}} \longrightarrow \underbrace{(y_1, \ldots, y_T)}_{\text{many outputs}}$$

Example: a simulator parameter affects an entire trajectory.

Derivatives along one input direction.

Automatic differentiation (autodiff)



Backward rules reuse values saved during the forward pass.

Autodiff in PyTorch

```
import torch.nn.functional as F
# Parameters use requires_grad=True to track gradients

hidden = torch.relu(x @ W1.T + b1)      # @ = matrix multiply (matmul)
logits = hidden @ W2.T + b2             # W2.T is the transpose of W2
loss = F.cross_entropy(logits, labels)  # Cross-entropy on logits

loss.backward()                         # Run backpropagation
dW1 = W1.grad                           # Gradient of W1 from backprop
dW2 = W2.grad                           # Gradient of W2 from backprop
```

Minibatch SGD in PyTorch

```
optimizer = torch.optim.SGD(
    model.parameters(), lr=learning_rate
)

for x, labels in loader:                          # One minibatch
    optimizer.zero_grad()                         # Clear old gradients
    loss = F.cross_entropy(model(x), labels)      # Forward pass and loss
    loss.backward()                               # Compute gradients
    optimizer.step()                              # Update parameters
```

Choosing the learning rate

Learning rate is a hyperparameter: a chosen training setting.

Stochastic Gradient Descent (SGD)

Initialize the network parameters

For each update

1	$B_t \leftarrow \text{a sampled minibatch}$	Input batch
2	$g_t \leftarrow \frac{1}{\lvert B_t \rvert} \sum_{i \in B_t} \nabla_\theta \ell_i(\theta_t)$	Gradient
3	$d_t \leftarrow g_t$	Update vector
4	$\eta_t \leftarrow \eta$	**Learning rate**
5	$\theta_{t+1} \leftarrow \theta_t - \eta_t d_t$	

End

SGD convergence conditions

With persistent gradient noise, let the learning rate decrease.

$$\eta_t \longrightarrow 0$$

$$\sum_{t=1}^{\infty} \eta_t = \infty$$

Enough total opportunity to move

$$\sum_{t=1}^{\infty} \eta_t^2 < \infty$$

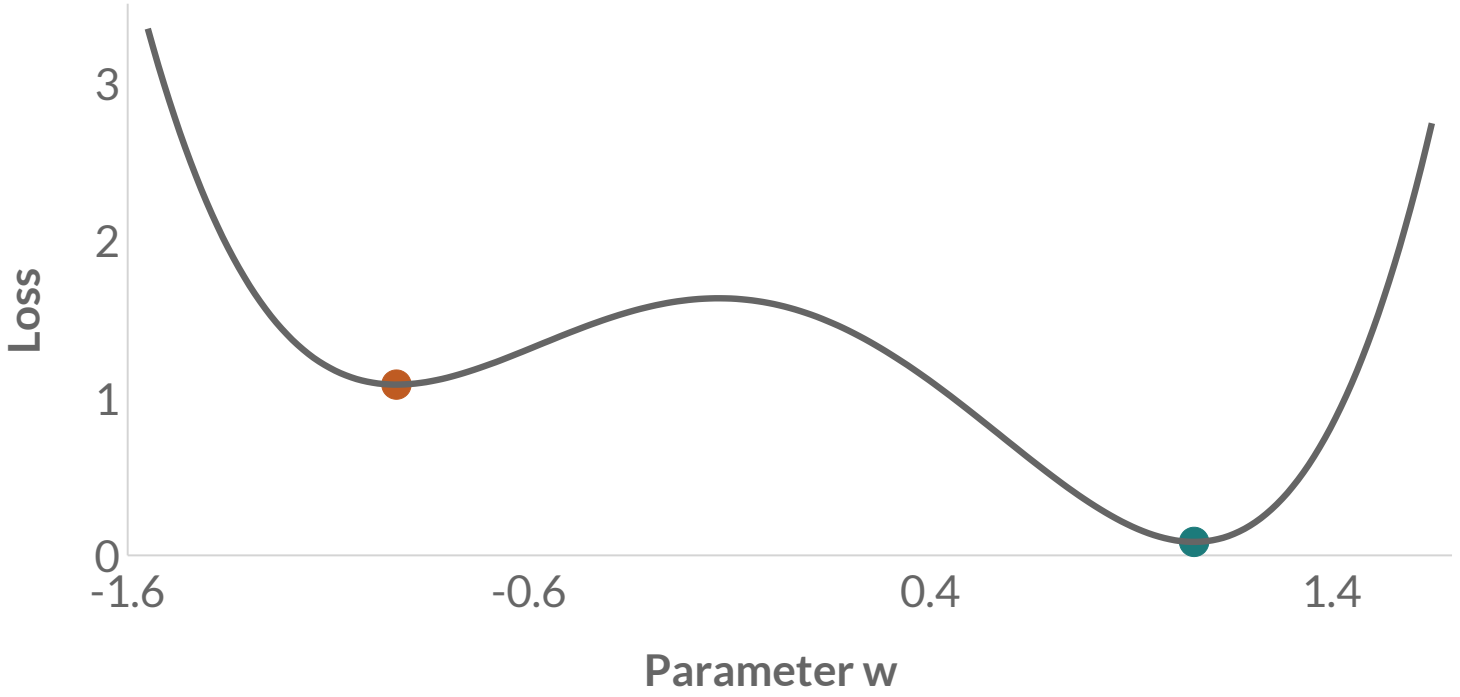
Control the accumulated noise

Smooth, lower-bounded loss; unbiased gradients with bounded noise variance.

Small gradients do not guarantee a global minimum.

Local and global minima

SGD may settle at a local minimum. A global minimum is not guaranteed.



Local minimum

Lowest loss nearby

Global minimum

Lowest loss over all parameter values

Learning rate and curvature

$$\mathcal{L}(w) = \tfrac{1}{2} a w^2, \qquad a > 0$$

a controls curvature: larger a gives a steeper quadratic.

$$w_{t+1} = w_t - \eta a w_t = (1 - \eta a) w_t$$

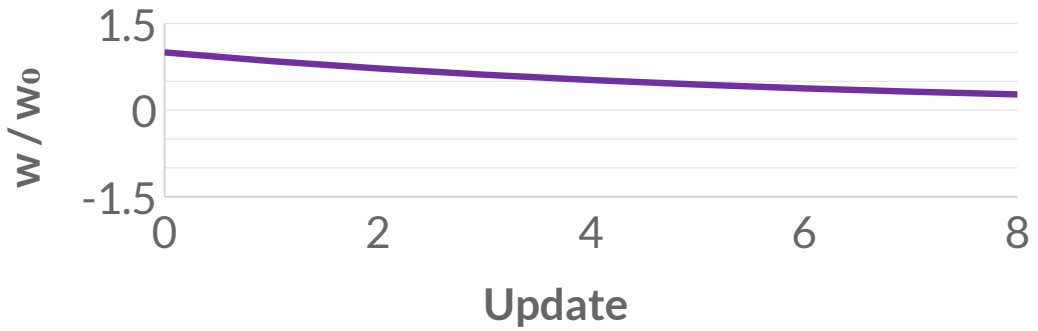
The product η a determines whether each update contracts or expands.

$$|1 - \eta a| < 1 \quad \Longleftrightarrow \quad 0 < \eta a < 2$$

Stable and unstable step sizes

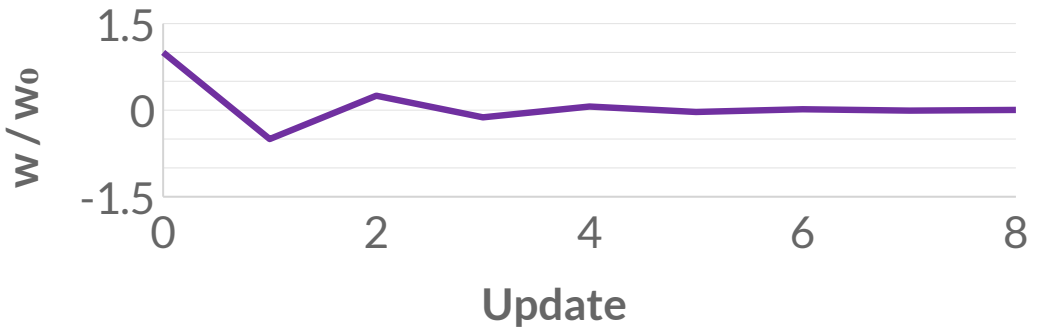
Monotone descent

$0 < \eta a < 1$



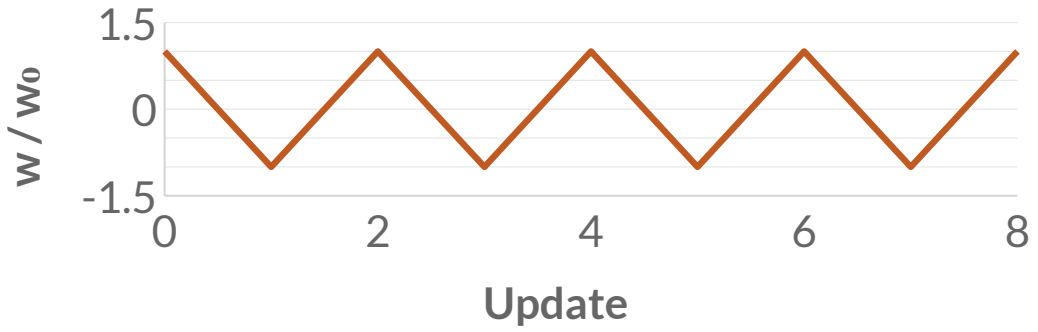
Shrinking oscillation

$1 < \eta a < 2$



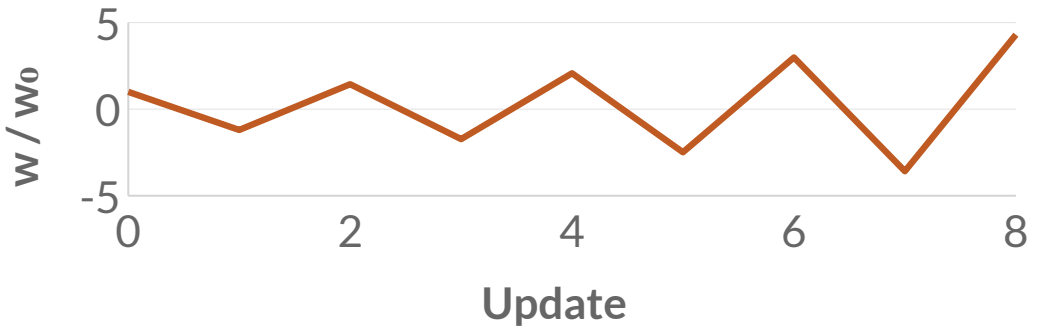
Mirror oscillation

$\eta a = 2$



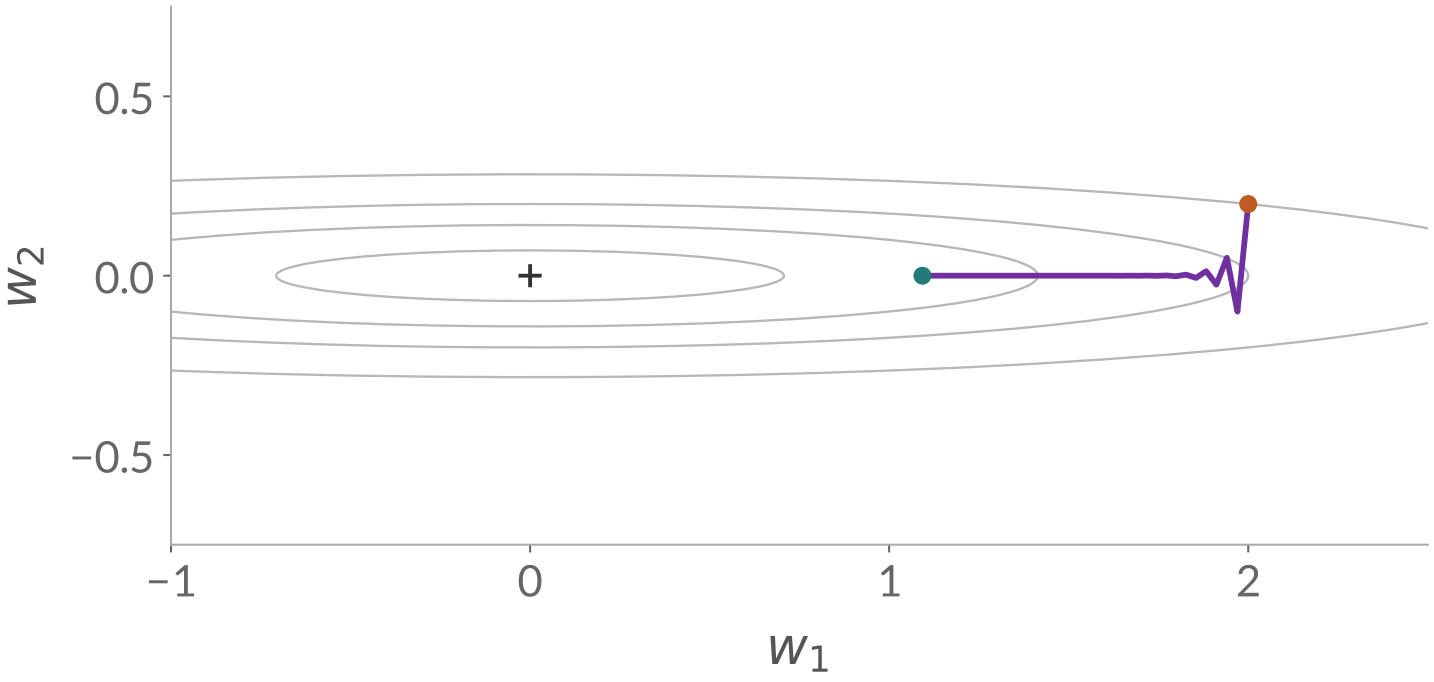
Divergence

$\eta a > 2$



Gradient descent in the valley

Orange: start Teal: after 40 updates



$$\eta = 0.015$$

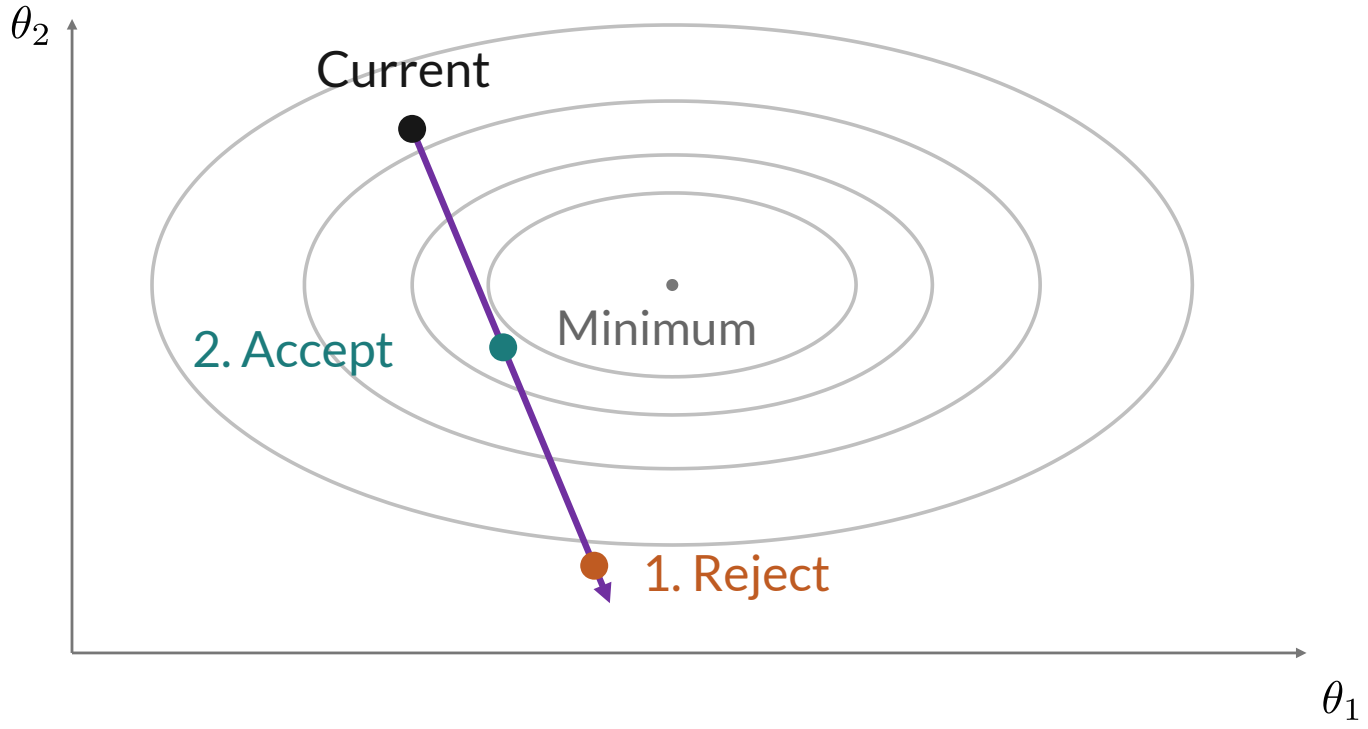
$$w_{1,t+1} = 0.985 w_{1,t}$$

$$w_{2,t+1} = -0.5 w_{2,t}$$

- Fast contraction across the valley.
- Slow progress along it.

Line search

Fix the negative-gradient direction. Try shorter steps.



Minibatch noise

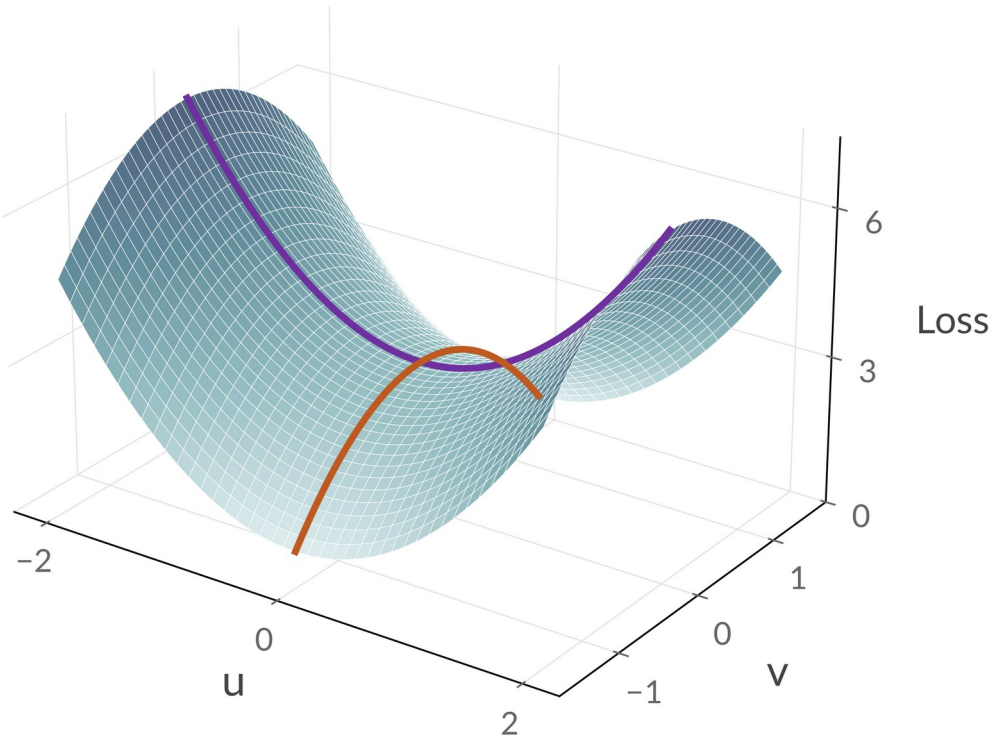
A batch can misjudge the full-loss change.

Computation cost

Every trial requires another forward pass.

Conventional line search is uncommon in large-scale deep learning.

Saddle points



$$\nabla \mathcal{L}(0,0) = 0$$

Upward in one direction, downward in another

A downhill direction remains available.

A zero gradient does not imply a local minimum.

Saddles in high dimensions

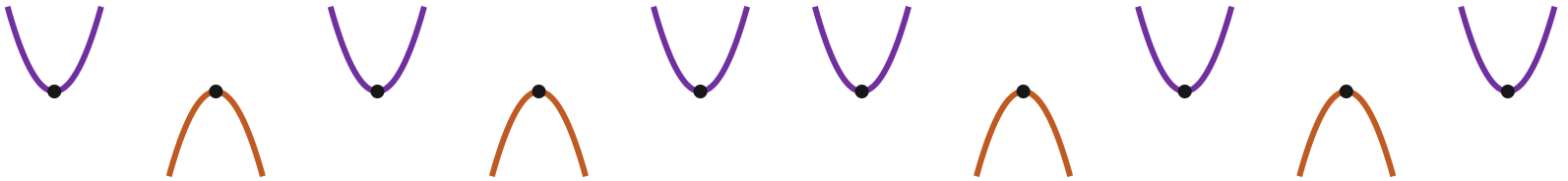
Curvature at a critical point (ignoring flat directions)

Local minimum

Every direction curves upward

Saddle point

Upward and downward curvature



Ten directions give 1,024 sign patterns: one all-up, 1,022 mixed.
The patterns need not be equally likely.

Choosing the update vector

Stochastic Gradient Descent (SGD)

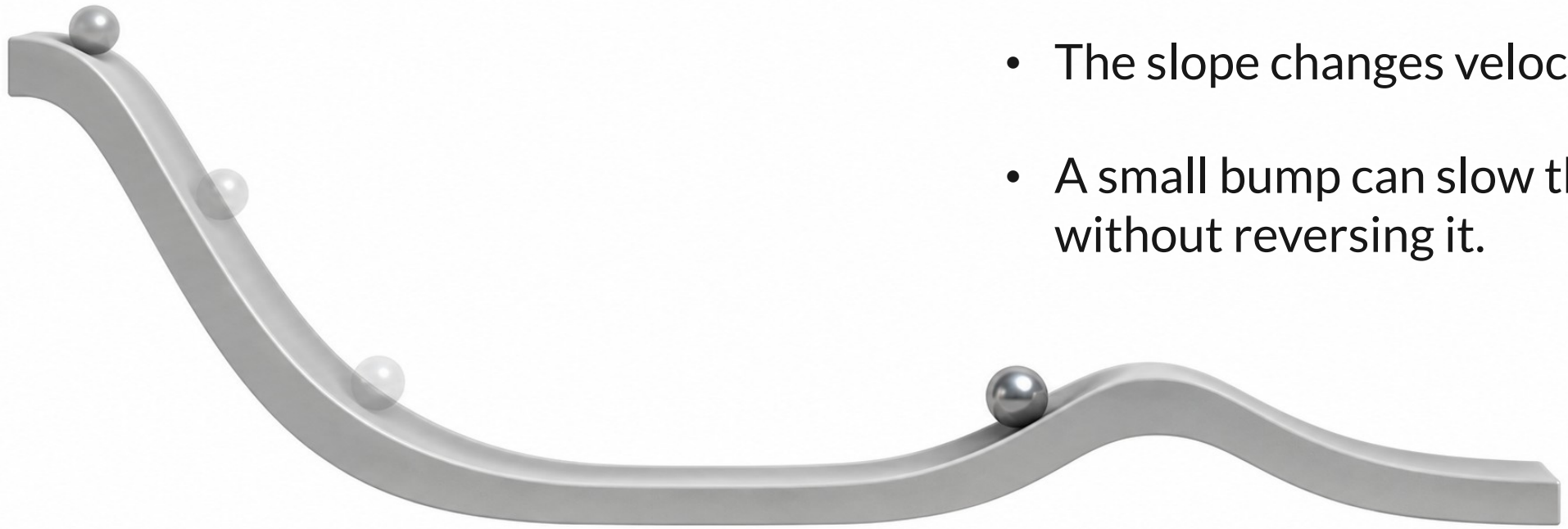
Initialize the network parameters

For each update

1	$B_t \leftarrow \text{a sampled minibatch}$	Input batch
2	$g_t \leftarrow \frac{1}{\lvert B_t \rvert} \sum_{i \in B_t} \nabla_\theta \ell_i(\theta_t)$	Gradient
3	$d_t \leftarrow g_t$	**Update vector**
4	$\eta_t \leftarrow \eta$	Learning rate
5	$\theta_{t+1} \leftarrow \theta_t - \eta_t d_t$	

End

Momentum: the heavy-ball analogy



Earlier positions

The same ball, later

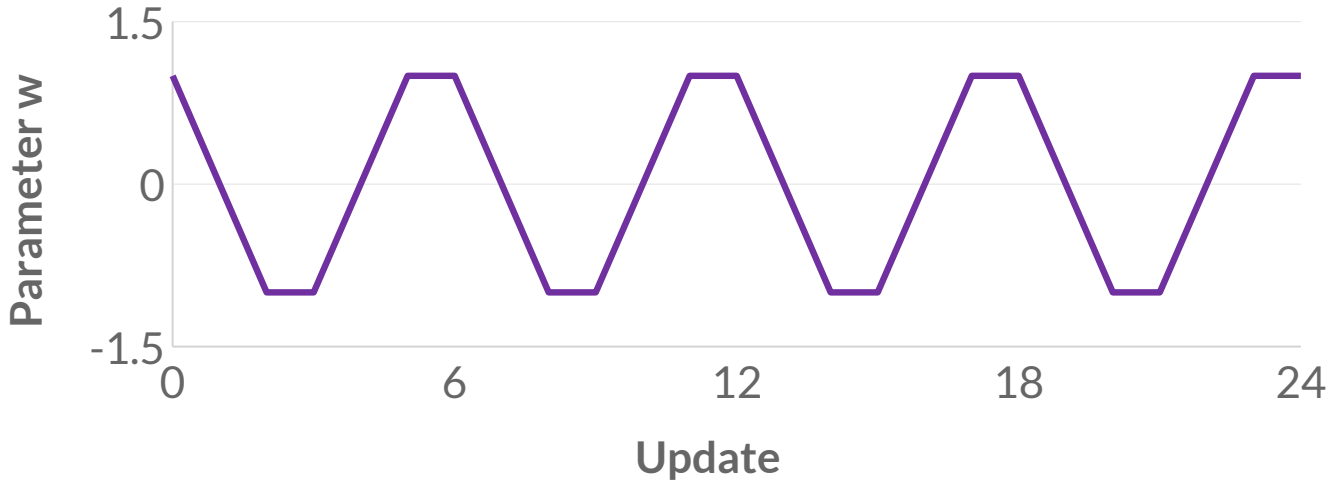
Motion persists

- The slope changes velocity.
- A small bump can slow the ball without reversing it.

Accumulating momentum

$$v_t = v_{t-1} + g_t, \qquad \theta_t = \theta_{t-1} - \eta v_t$$

Retain the previous motion when a new gradient arrives.



$$\mathcal{L}(w) = \tfrac{1}{2}w^2, \quad \eta = 1$$

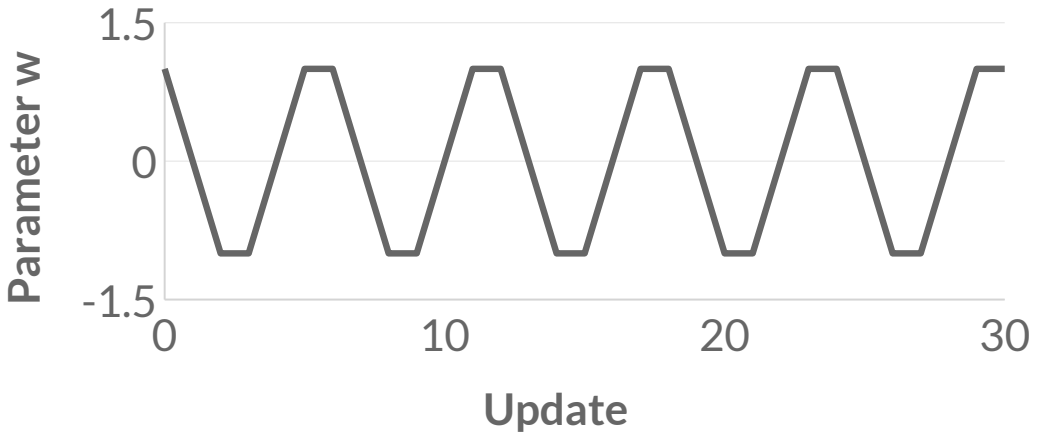
The parameter repeatedly crosses the minimum.

Old motion does not decay.

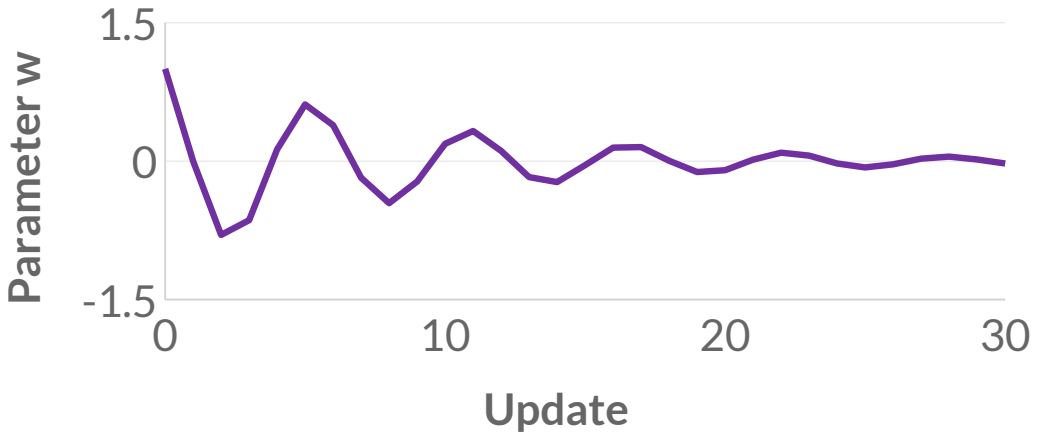
Damping the momentum

$$v_t = \mu v_{t-1} + g_t, \quad w_t = w_{t-1} - v_t, \quad \mathcal{L}(w) = \tfrac{1}{2} w^2$$

Without damping $\mu = 1$



With damping $\mu = 0.8$



Retain a fraction of the old history at each update.

Exponential moving averages

Accumulated history

$$v_t = \mu v_{t-1} + g_t$$

Normalized average

$$m_t = (1-\mu)v_t = \mu m_{t-1} + (1-\mu)g_t$$

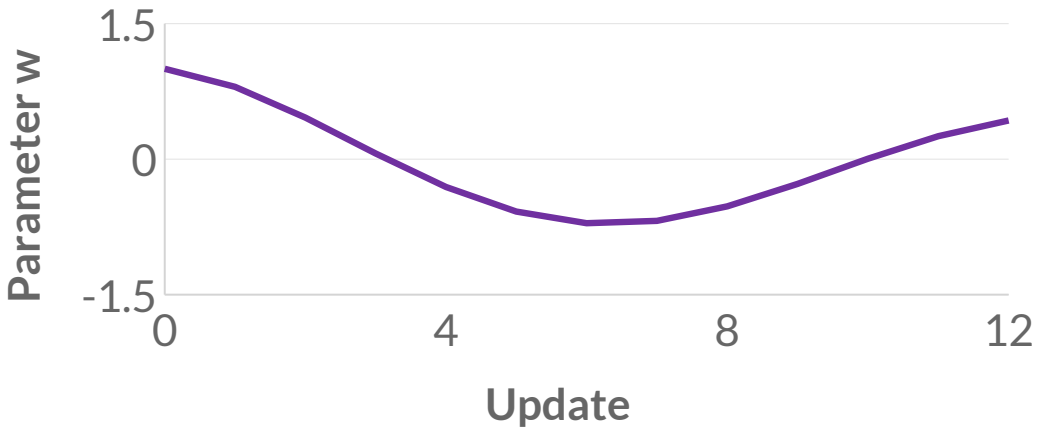
$$\mu = 0.9: \qquad m_t = 0.9 m_{t-1} + 0.1 g_t$$

An exponential moving average (EMA) discounts older gradients.

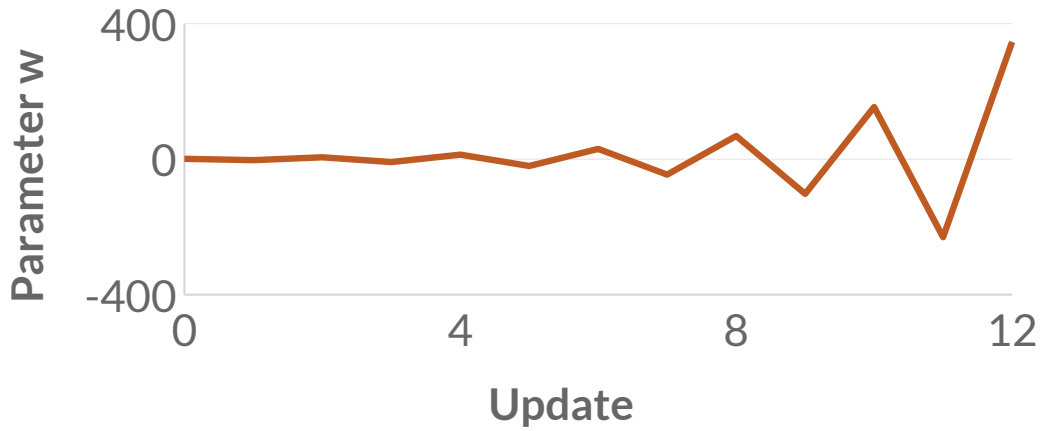
Momentum and learning rate

$$v_t = \mu v_{t-1} + g_t, \quad w_t = w_{t-1} - \eta v_t, \quad \mu = 0.9$$

$\eta = 0.2$



$\eta = 4.0$



Damping retains useful history; the learning rate still controls the step size.

SGD with momentum: two implementations

$$\eta \text{ fixed}, \qquad v_0 = u_0 = 0$$

Gradient buffer

$$v_t = \mu v_{t-1} + g_t$$

$$\theta_t = \theta_{t-1} - \eta v_t$$

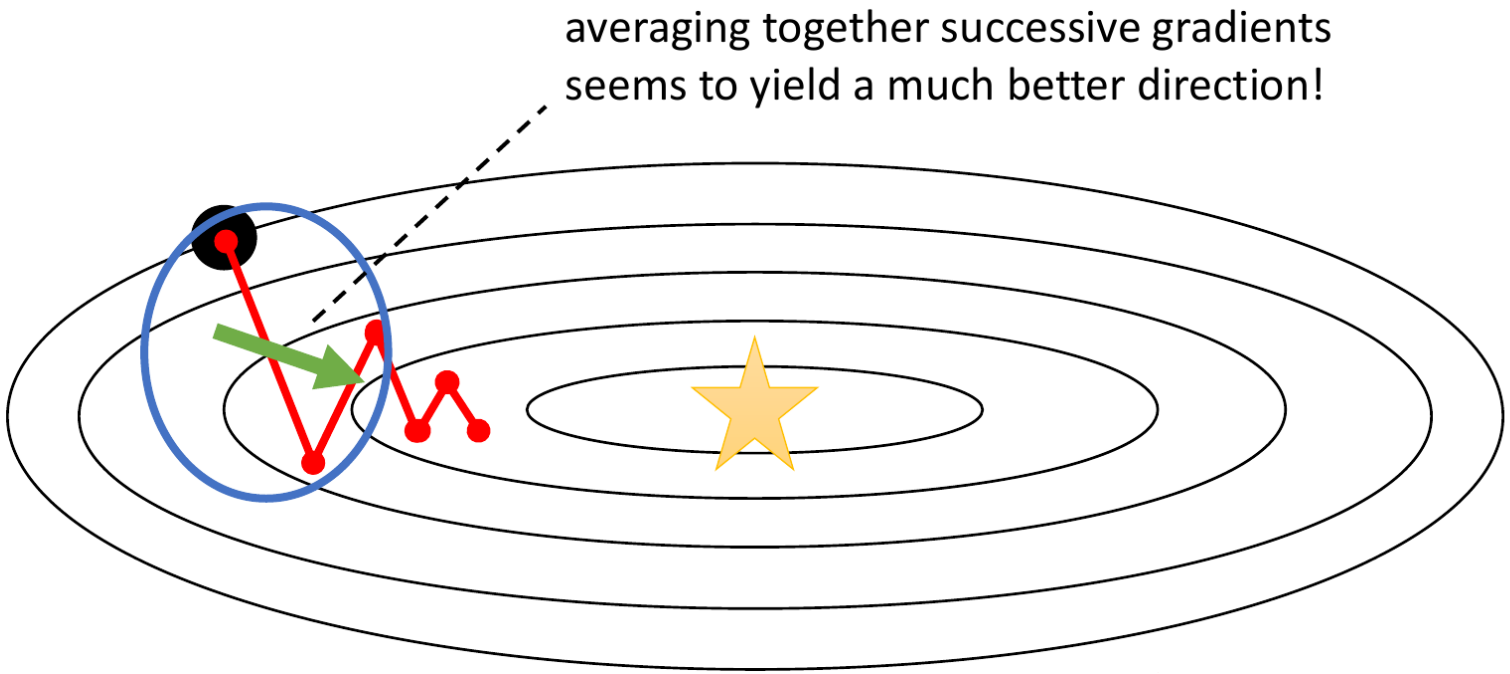
Parameter displacement

$$u_t = \mu u_{t-1} - \eta g_t$$

$$\theta_t = \theta_{t-1} + u_t$$

$$u_t = -\eta v_t$$

Momentum in a narrow valley

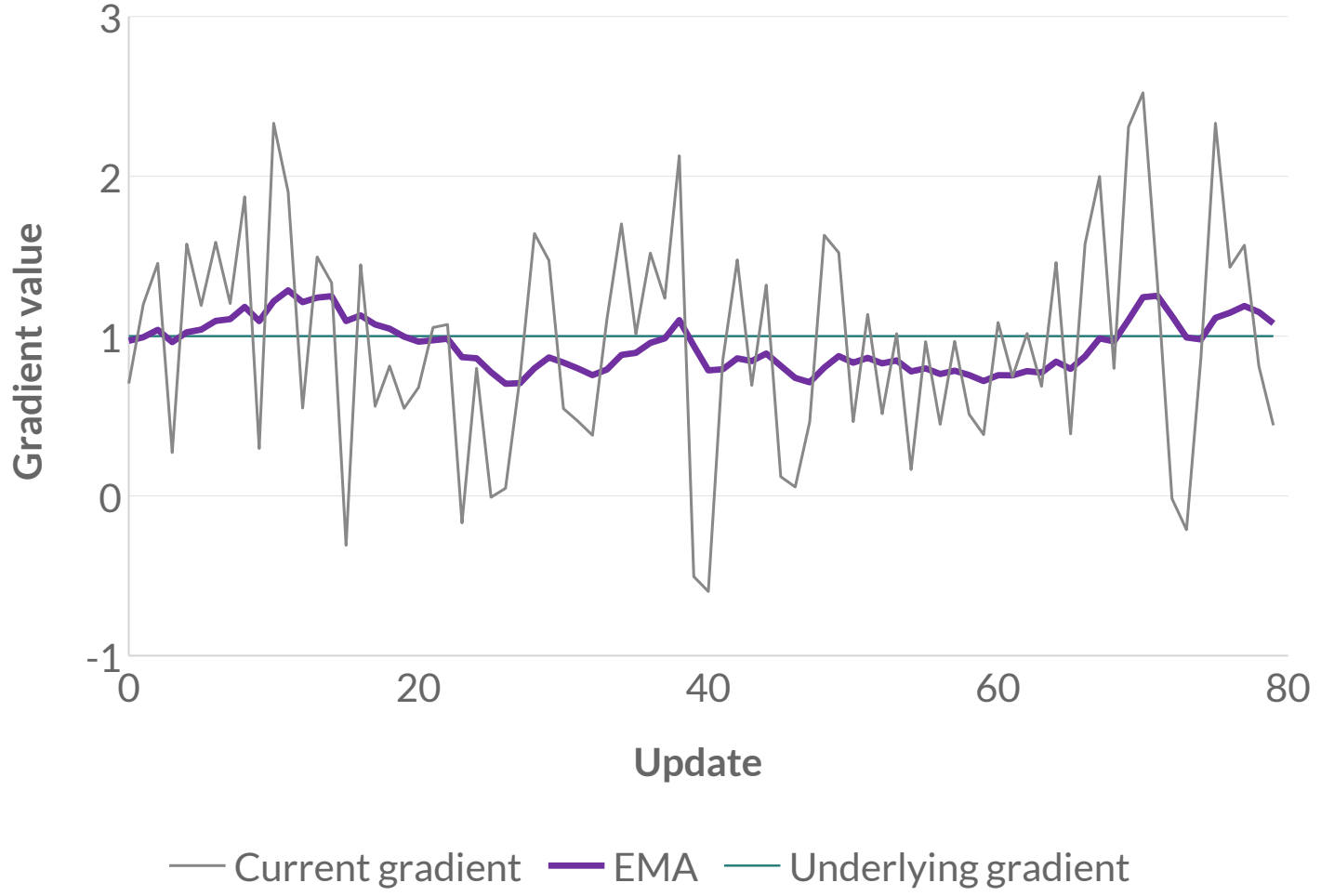


- Across the valley: directions alternate.
- Along the valley: directions agree.

Figure: Berkeley CS182

Momentum and minibatch noise

Illustration: average initialized at 1



$$g_t = 1 + \xi_t$$

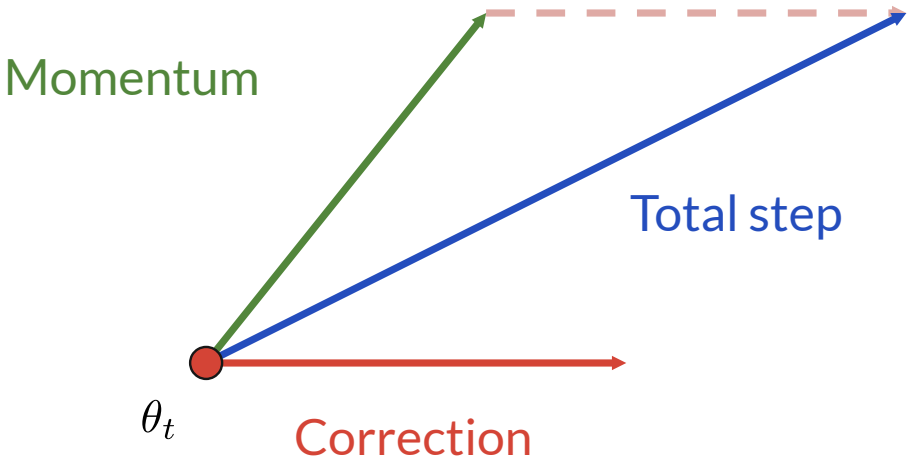
$$m_t = 0.9 m_{t-1} + 0.1 g_t$$

- A consistent direction remains.
- Independent fluctuations partly cancel.

Nesterov momentum

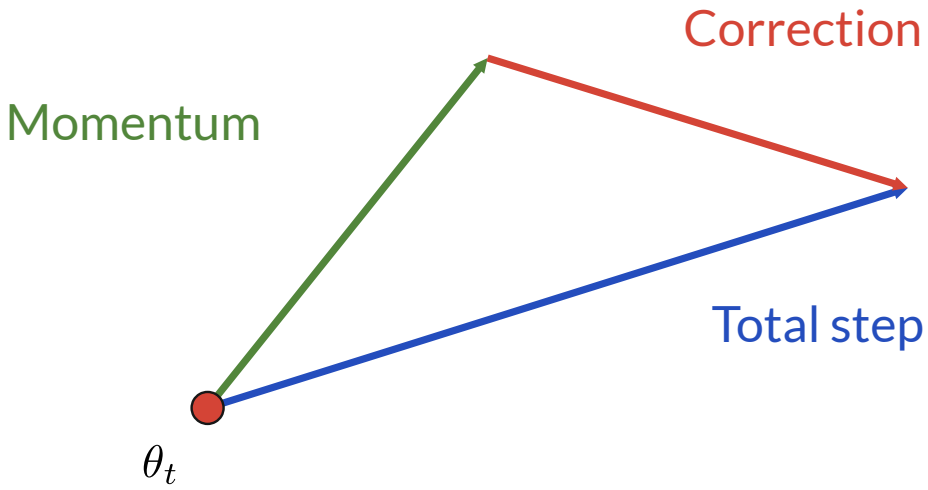
Also called Nesterov accelerated gradient (NAG)

Ordinary momentum



Gradient at the current point

Nesterov momentum



Gradient at the look-ahead point

Evaluate the gradient where momentum would take us,
then use that correction in the combined step.

[Diagram adapted from Georgia Tech CS7643]

Nesterov update

$$\theta: \text{ current parameter}, \qquad v: \text{ stored gradient history}$$

Look ahead

$$\widetilde{\theta} = \theta - \eta\mu v$$

Gradient there

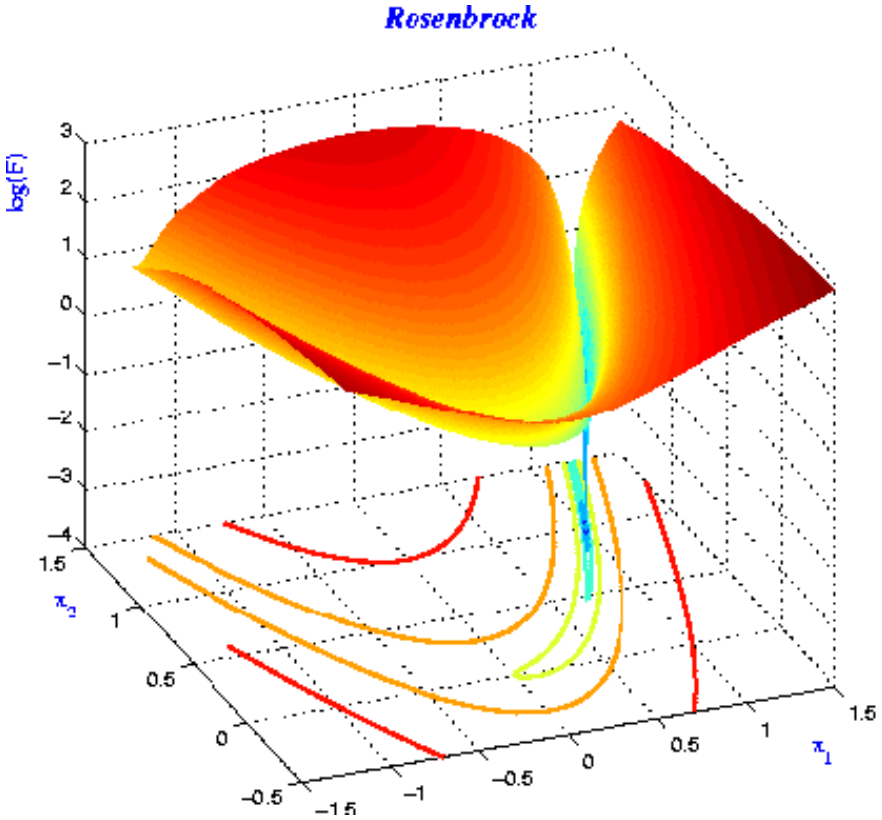
$$g_{\text{look}} = \nabla\mathcal{L}(\widetilde{\theta}), \qquad v_{\text{new}} = \mu v + g_{\text{look}}$$

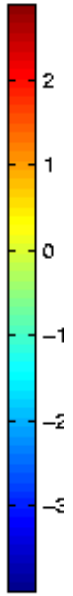
Update once

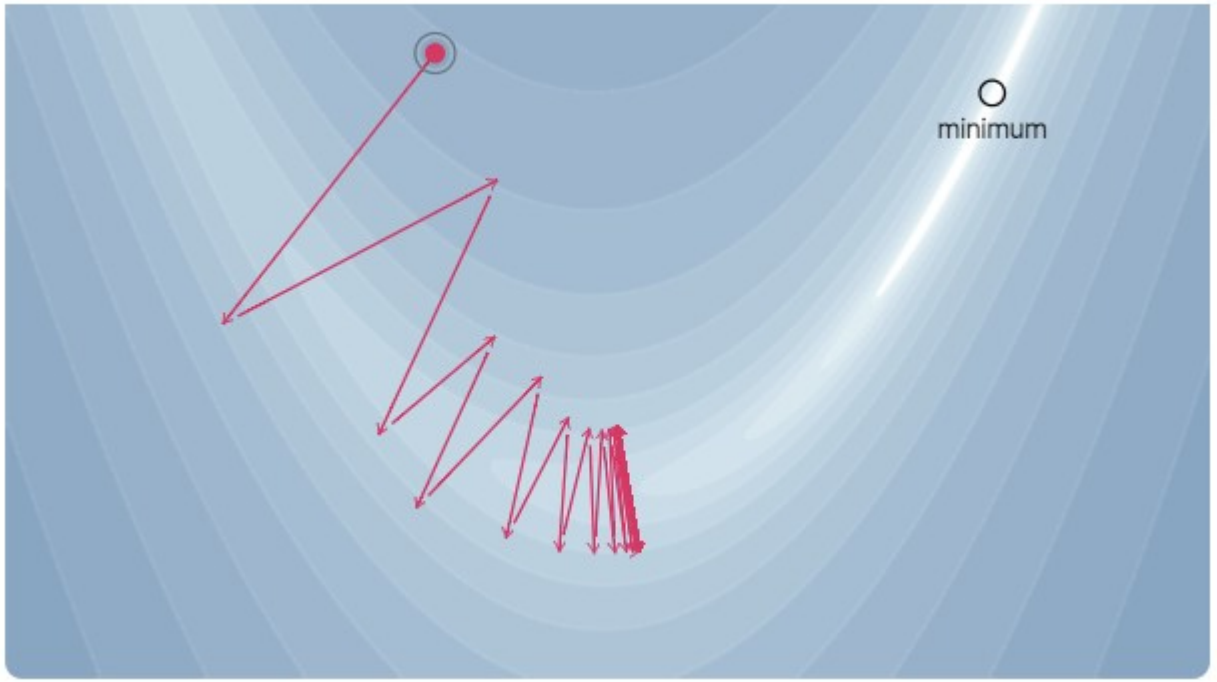
$$\theta_{\text{new}} = \theta - \eta v_{\text{new}} = \widetilde{\theta} - \eta g_{\text{look}}$$

Long, narrow ravines

A step size that is safe across the ravine can be too small along it.





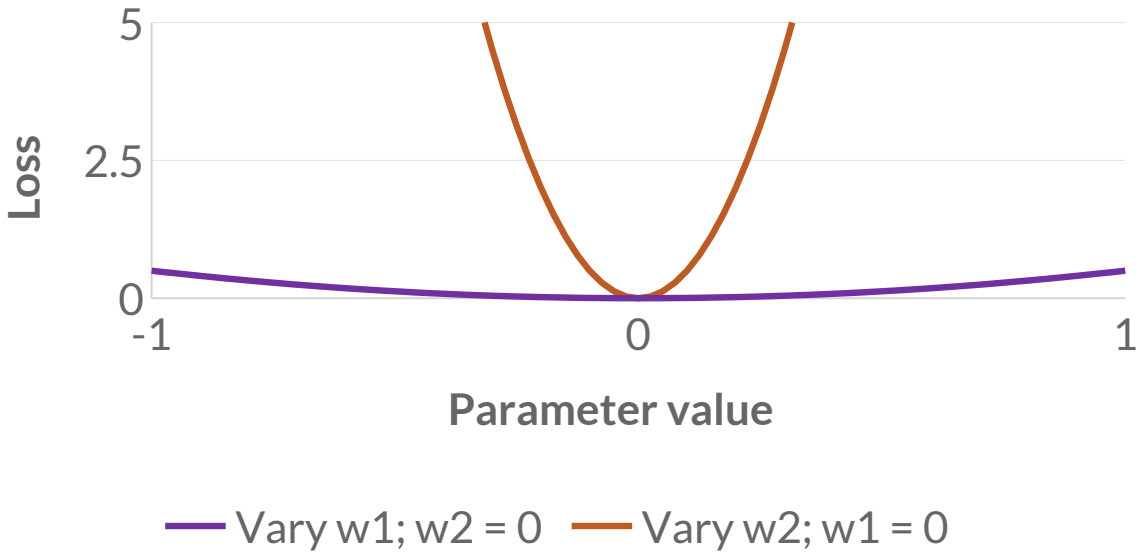


Equal-loss contours; red gradient-descent steps

Hessian

The Hessian is the matrix of second derivatives.

$$H_{ij}(\theta) = \frac{\partial^2 \mathcal{L}}{\partial \theta_i \, \partial \theta_j}$$



Curvature in different parameter directions

$$\mathcal{L} = \tfrac{1}{2}(w_1^2 + 100 w_2^2)$$

$$H = \begin{bmatrix} 1 & 0 \\ 0 & 100 \end{bmatrix}$$

Shallow in w_1; steep in w_2.

Newton's method

Use the local quadratic model to choose the step.

Δ: proposed parameter change

$$\mathcal{L}(\theta + \Delta) \approx \mathcal{L}(\theta) + g^\mathsf{T}\Delta + \tfrac{1}{2}\Delta^\mathsf{T} H \Delta$$

Set the quadratic's slope to zero

$$H_t d_t = g_t, \qquad \theta_t = \theta_{t-1} - \eta_t d_t$$

d: gradient adjusted for curvature

For the same quadratic

$$\underbrace{\begin{bmatrix} 1 & 0 \\ 0 & 1/100 \end{bmatrix}}_{H^{-1}} \underbrace{\begin{bmatrix} w_1 \\ 100 w_2 \end{bmatrix}}_{g} = \underbrace{\begin{bmatrix} w_1 \\ w_2 \end{bmatrix}}_{d}$$

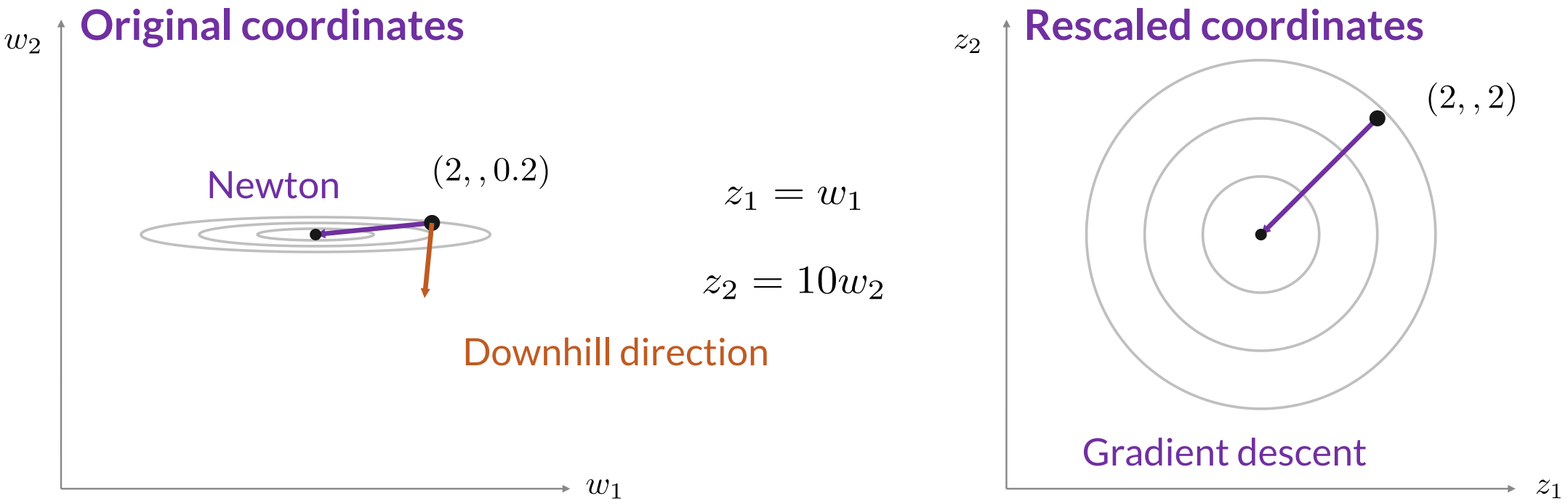
A full step reaches the minimum of this positive quadratic.

Hessian scaling and loss geometry

The same quadratic in two coordinate systems

$$\mathcal{L}(w) = \tfrac{1}{2}(w_1^2 + 100w_2^2)$$

$$\widetilde{\mathcal{L}}(z) = \tfrac{1}{2}(z_1^2 + z_2^2)$$



Newton in the original coordinates is gradient descent in the rescaled coordinates.

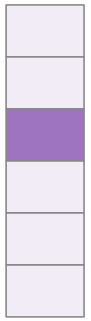
Cost of full Newton updates

Example: a model with 1 million parameters

Gradient

One entry per parameter

Parameters



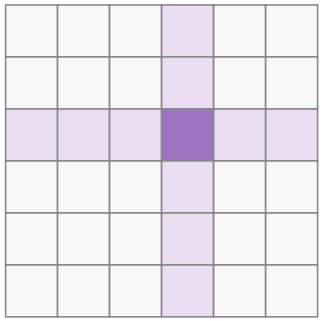
1 million entries

Hessian

One entry for every parameter pair

Parameters

Parameters



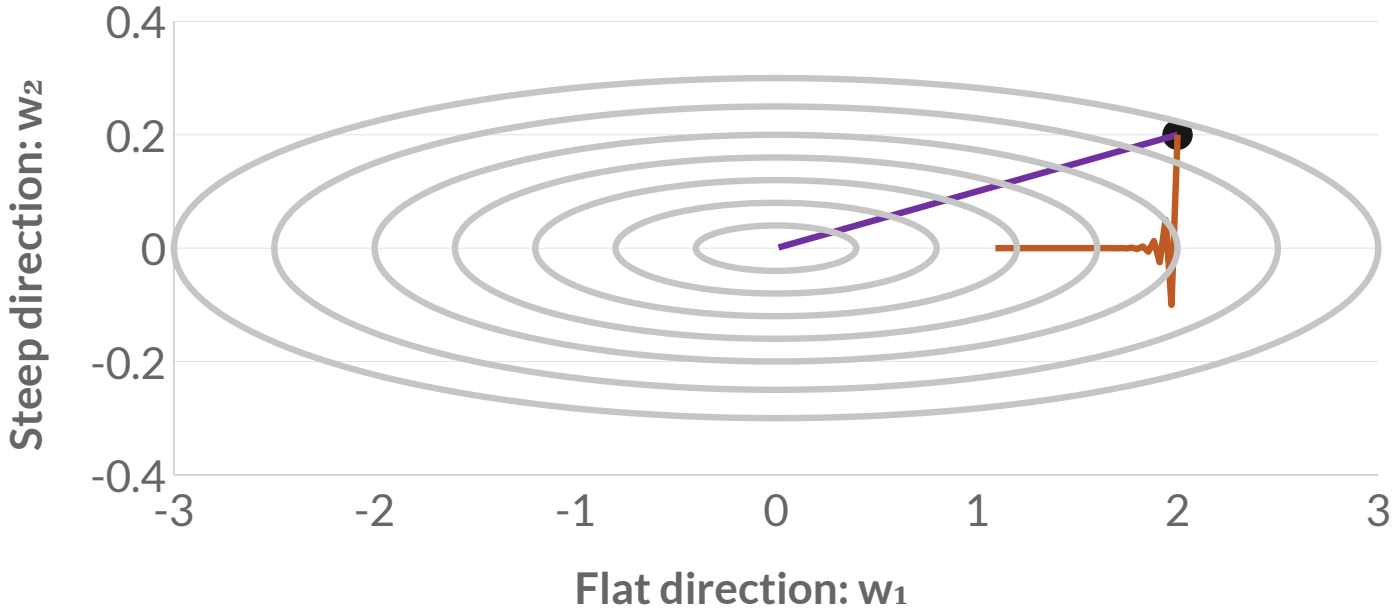
1 trillion entries

$$\underbrace{1 \text{ million} \times 1 \text{ million}}_{1 \text{ trillion entries}} \times 4 \text{ bytes} = 4 \text{ TB}$$

Computing all these second derivatives is expensive.
Newton must also solve a large system of equations to find each update.

Adaptive learning rates

Use gradient history to choose inexpensive coordinate scales.



Shared learning rate

Limited by the steep direction.

Curvature-based reference

Smaller across the valley; larger along it.

Adaptive methods use gradient magnitudes to set coordinate scales.

Adaptive optimizers still need a global learning rate.

AdaGrad

$$s_0 = 0$$

$$s_t = s_{t-1} + g_t \odot g_t$$

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

- Large accumulated squared gradients reduce the step scale.

AdaGrad's coordinate scale

First update: omit epsilon and consider a nonzero gradient.

$$g = (g_1, \ldots, g_P), \qquad s = g \odot g$$

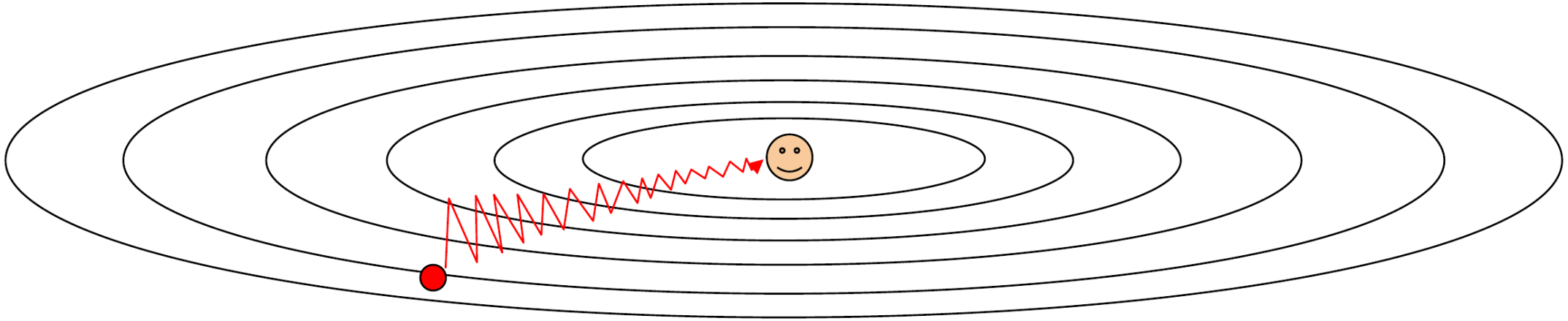
$$\frac{g_j}{\sqrt{s_j}} = \frac{g_j}{|g_j|}$$

On the first step, a larger gradient also receives a larger denominator.

Later steps use accumulated history. A shared learning rate still sets the scale.

AdaGrad in the valley

$$\eta_{t,i} = \frac{\eta}{\sqrt{s_{t,i}} + \varepsilon}$$



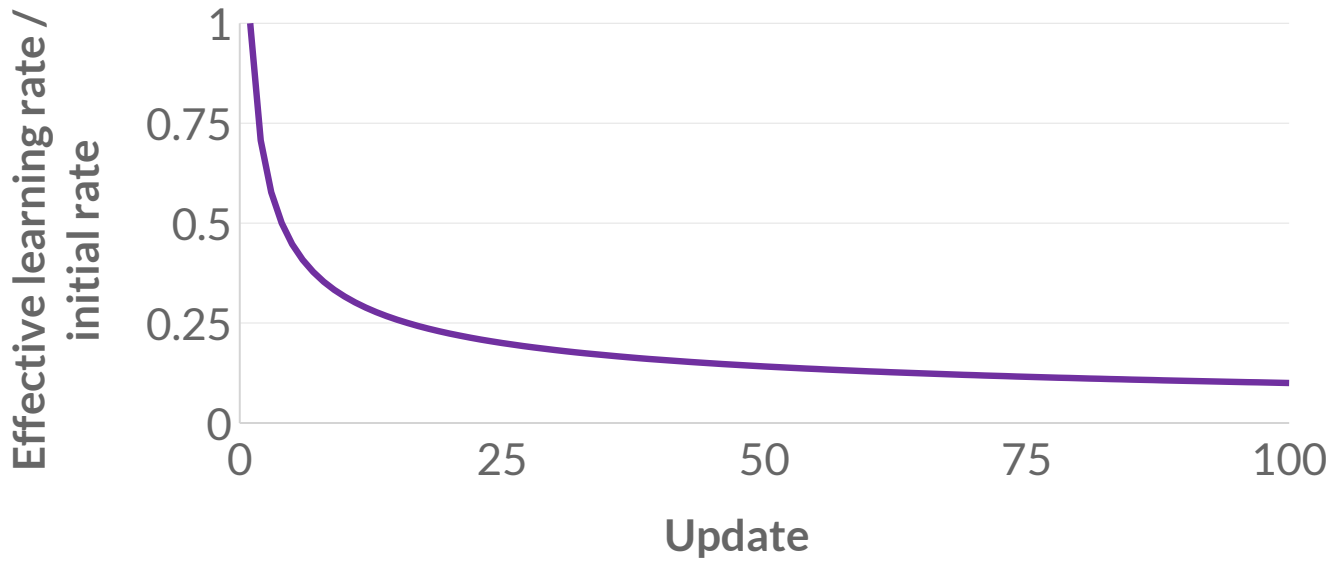
- Repeatedly large gradients increase the denominator.
- That coordinate receives a smaller effective learning rate.

Figure: Georgia Tech CS7643

AdaGrad's decreasing learning rate

Constant nonzero gradient; fixed global learning rate; omit ε.

$$s_t = tg^2, \qquad \frac{\eta_{t,\text{eff}}}{\eta_{1,\text{eff}}} = \frac{1}{\sqrt{t}}$$



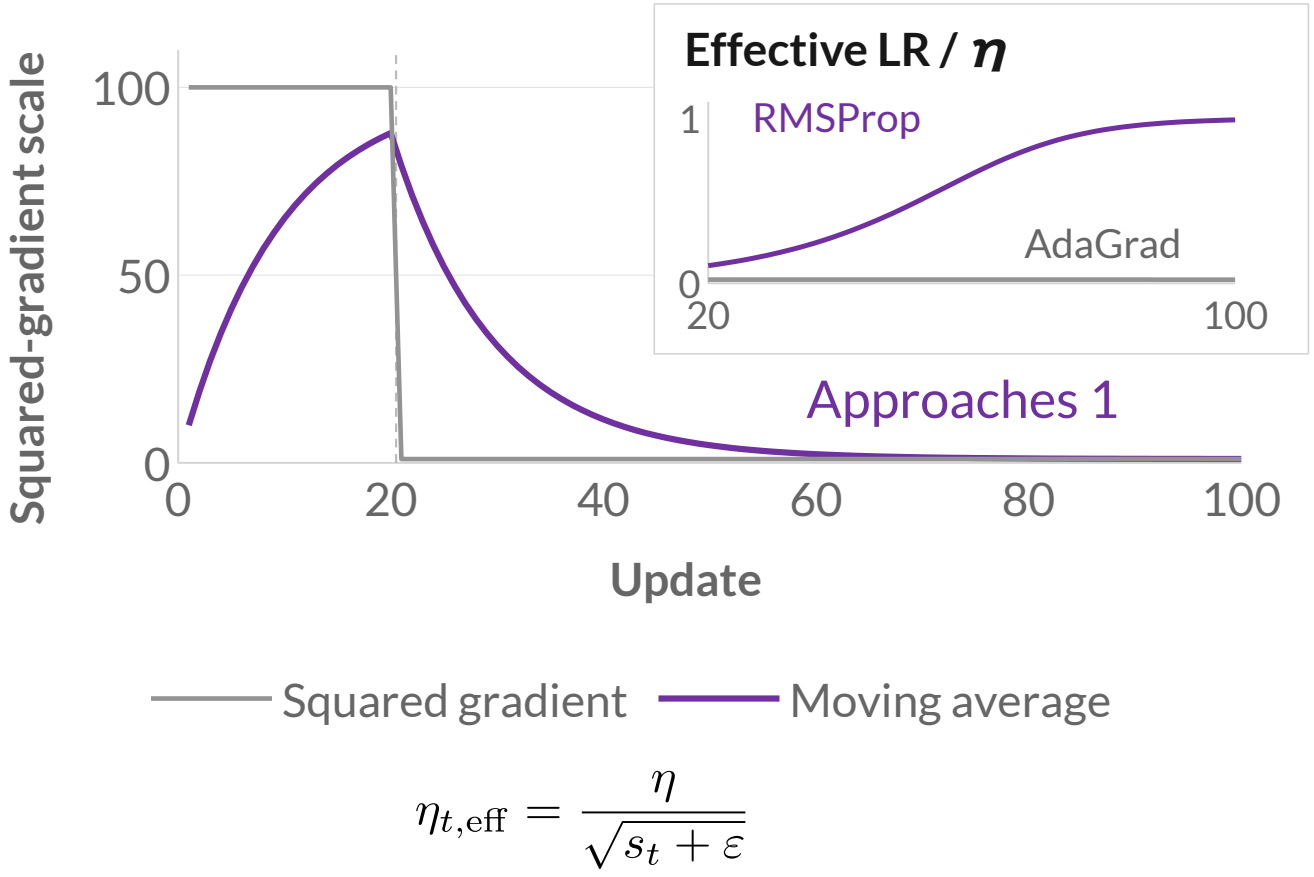
Every squared gradient remains in the accumulator.

The effective learning rate cannot increase.
Training may slow before reaching a useful solution.

Forgetting old gradient scales

Example: one parameter's gradient is 10 for 20 updates, then 1.

Both accumulators start at zero.



$$\eta_{t,\text{eff}} = \frac{\eta}{\sqrt{s_t + \varepsilon}}$$

AdaGrad: permanent sum

$$s_{20} = 20 \times 10^2 = 2000$$

$$s_{21} = 2000 + 1 = 2001$$

Moving average

$$s_t = 0.9 s_{t-1} + 0.1 g_t^2$$

Old large values lose weight.

As the average falls,
the step scale recovers.

RMSProp

Replace the permanent sum with a recent average.

AdaGrad

$$s_t = s_{t-1} + g_t^2$$

RMSProp

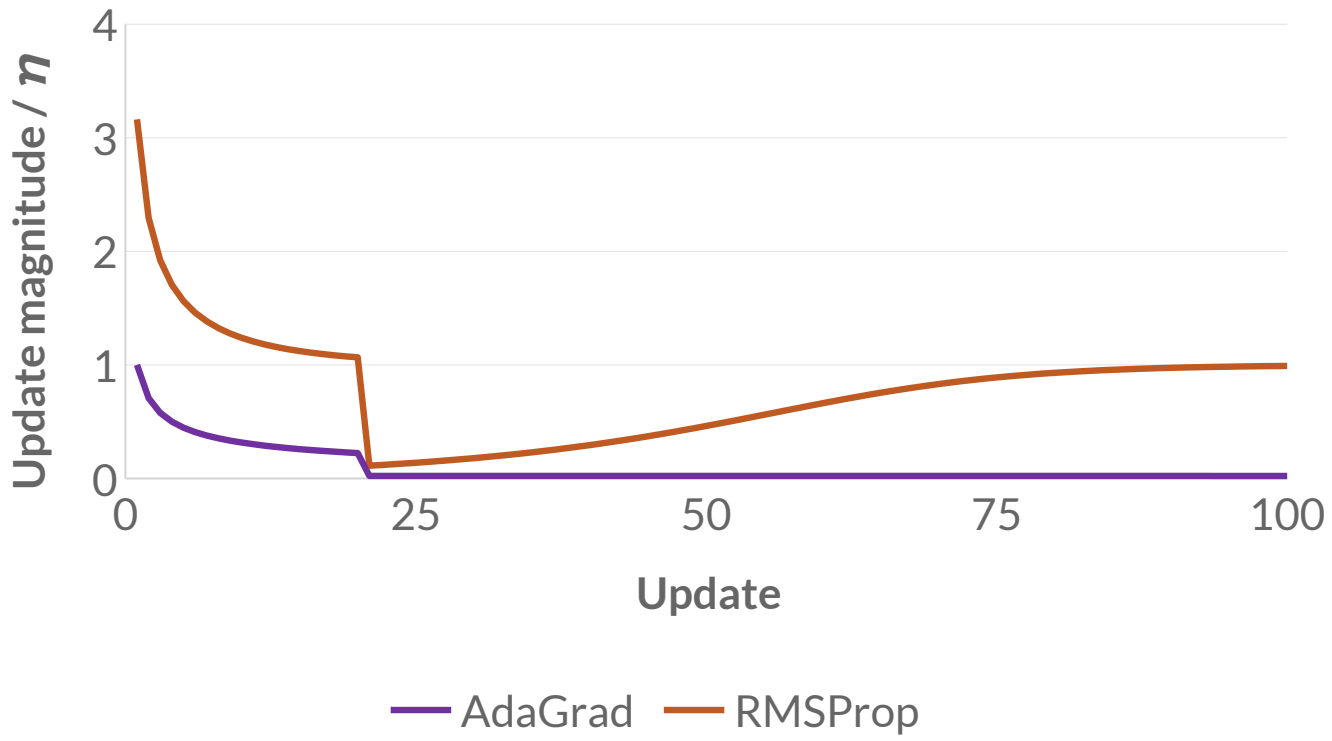
$$s_t = \rho s_{t-1} + (1 - \rho) g_t^2$$

Same update form

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

Recovery after a scale change

Same gradient history: 10 for 20 updates, then 1.



Update magnitude / η

$$\frac{g_t}{\sqrt{s_t} + \varepsilon}$$

AdaGrad retains the large early squared gradients.

RMSProp gradually adapts to the smaller current gradients.

RMSProp's original reference

rmsprop: A mini-batch version of rprop

- rprop is equivalent to using the gradient but also dividing by the size of the gradient.
  - The problem with mini-batch rprop is that we divide by a different number for each mini-batch. So why not force the number we divide by to be very similar for adjacent mini-batches?
- rmsprop: Keep a moving average of the squared gradient for each weight

$$MeanSquare(w,t) = 0.9\ MeanSquare(w,\ t-1) + 0.1\left(\partial E / \partial w(t)\right)^2$$

- Dividing the gradient by $\sqrt{MeanSquare(w,\ t)}$ makes the learning work much better (Tijmen Tieleman, unpublished).

Hinton's 2012 lecture slides, not a paper.

Figure: Hinton's lecture slides (2012)

Momentum and RMSProp

Momentum

Average signed gradients

$$\Delta\theta = -\eta\,\overline{g}_t$$

Reduces disagreement across updates

RMSProp (basic)

Scale each coordinate

$$\Delta\theta = -\eta\frac{g_t}{\sqrt{s_t} + \varepsilon}$$

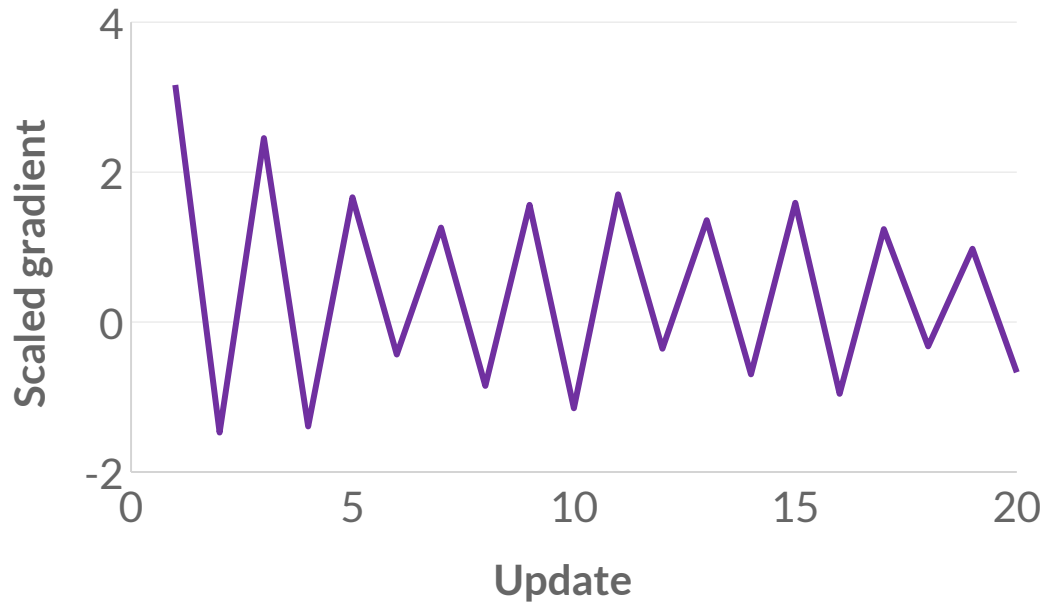
Uses the current gradient

Use the averaged gradient together with the RMS scale.

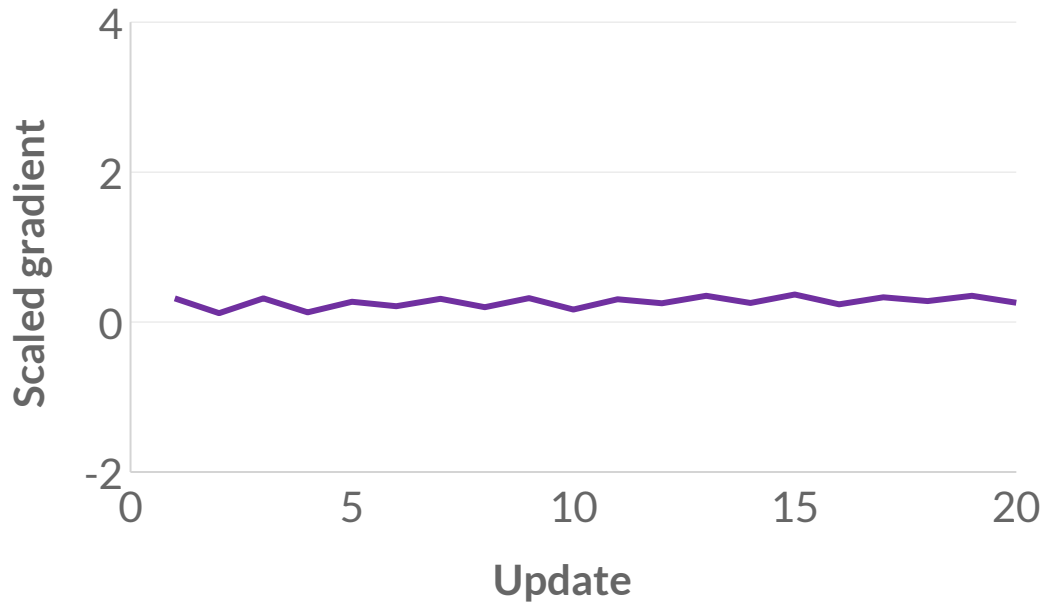
Combining direction and scale

Scaling preserves sign changes; averaging gives a steadier direction.

RMSProp: current gradient



Use the averaged gradient



Adam

Averaged direction + adaptive coordinate scale

Gradient average

From momentum

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

Mean squared gradient

From RMSProp

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

Provisional update

$$\theta_t = \theta_{t-1} - \eta \frac{m_t}{\sqrt{v_t} + \varepsilon}$$

Direction and magnitude in Adam

$$\text{Two gradient observations:} \qquad +4, \ -4$$

Direction can cancel

$$\frac{4 + (-4)}{2} = 0$$

- No agreement on a direction.

Magnitude should not cancel

$$\sqrt{\frac{4^2 + (-4)^2}{2}} = 4$$

- Large gradients in both observations.

Zero initialization in Adam

Every observed gradient is 2.

$$\beta_1 = 0.9, \qquad \beta_2 = 0.999, \qquad m_0 = v_0 = 0$$

Gradient average

$$m_1 = 0.9(0) + 0.1(2) = 0.2$$

Observed gradient: 2

Squared-gradient average

$$v_1 = 0.999(0) + 0.001(4) = 0.004$$

Observed squared gradient: 4

Effect on the parameter update

First update, ignoring epsilon in this example.

From the observed values

$$\frac{g_1}{\sqrt{g_1^2}} = \frac{2}{\sqrt{4}} = 1$$

Using the initial averages

$$\frac{m_1}{\sqrt{v_1}} = \frac{0.2}{\sqrt{0.004}} \approx 3.16$$

The initial zero makes the step 3.16 times larger in this example.

The startup factors do not cancel in the ratio.

Bias correction

Normalize by the total weight on the observed gradients.

$$m_t = (1-\beta_1)g_t + \beta_1(1-\beta_1)g_{t-1} + \cdots$$

$$\underbrace{(1-\beta_1)(1+\beta_1+\cdots+\beta_1^{t-1})}_{\text{total weight on observations}} = 1-\beta_1^t$$

$$\widehat{m}_t = \frac{m_t}{1-\beta_1^t}, \qquad \widehat{v}_t = \frac{v_t}{1-\beta_2^t}$$

$$\widehat{m}_1 = \frac{0.2}{0.1} = 2, \qquad \widehat{v}_1 = \frac{0.004}{0.001} = 4$$

The Adam update

$$\theta_t = \theta_{t-1} - \eta_t \frac{\widehat{m}_t}{\sqrt{\widehat{v}_t} + \varepsilon}$$

Direction	Corrected average gradient
Coordinate scale	Corrected squared-gradient average
Global scale	Learning rate η_t

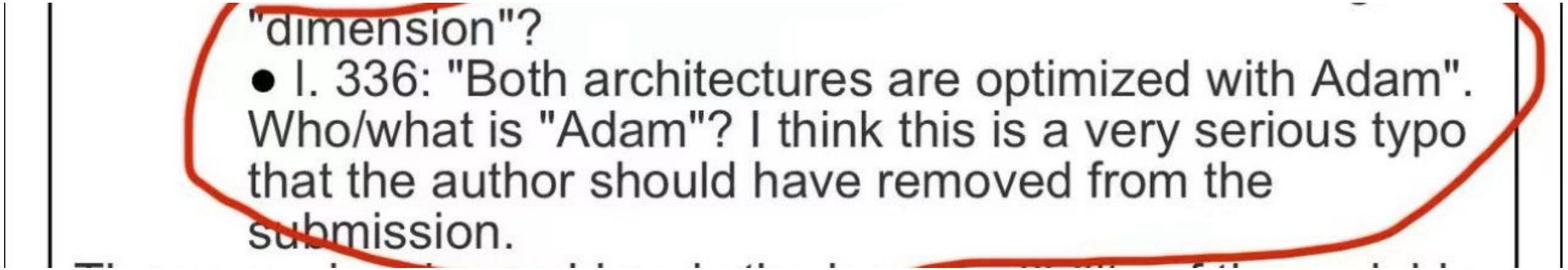
Adam algorithm

Adam

Require: $\theta_0,\ \eta_t,\ \beta_1, \beta_2 \in [0, 1),\ \varepsilon > 0$

1	$m_0 = 0, \quad v_0 = 0$	Two persistent state vectors.
2	**For update t = 1, 2, …**	
3	$g_t = \nabla_\theta \mathcal{L}_{B_t}(\theta_{t-1})$	Minibatch gradient
4	$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$	Direction history
5	$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$	Squared magnitude
6	$\widehat{m}_t = m_t / (1 - \beta_1^t), \quad \widehat{v}_t = v_t / (1 - \beta_2^t)$	Bias correction
7	$\theta_t = \theta_{t-1} - \eta_t \widehat{m}_t / (\sqrt{\widehat{v}_t} + \varepsilon)$	Parameter update

Adam is an optimizer


"dimension"?

- l. 336: "Both architectures are optimized with Adam". Who/what is "Adam"? I think this is a very serious typo that the author should have removed from the submission.

https://x.com/2prime_PKU/status/1948549824594485696

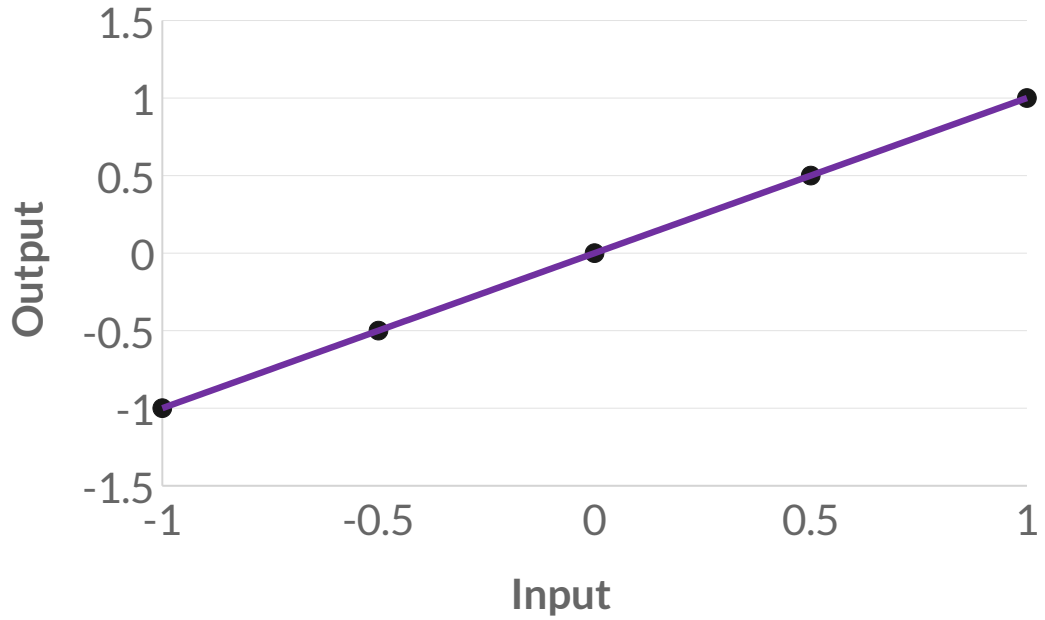
Optimizer trajectories

Animation plays automatically

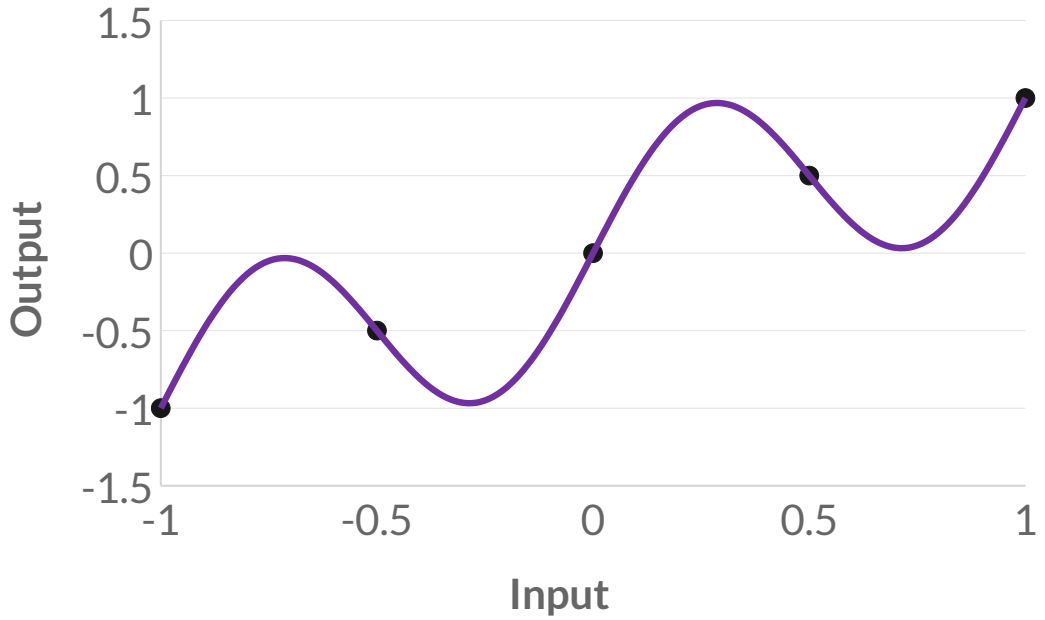
Occam's razor

With comparable data fit, prefer a simpler explanation.

Linear fit



Oscillating fit



Regularized empirical risk minimization

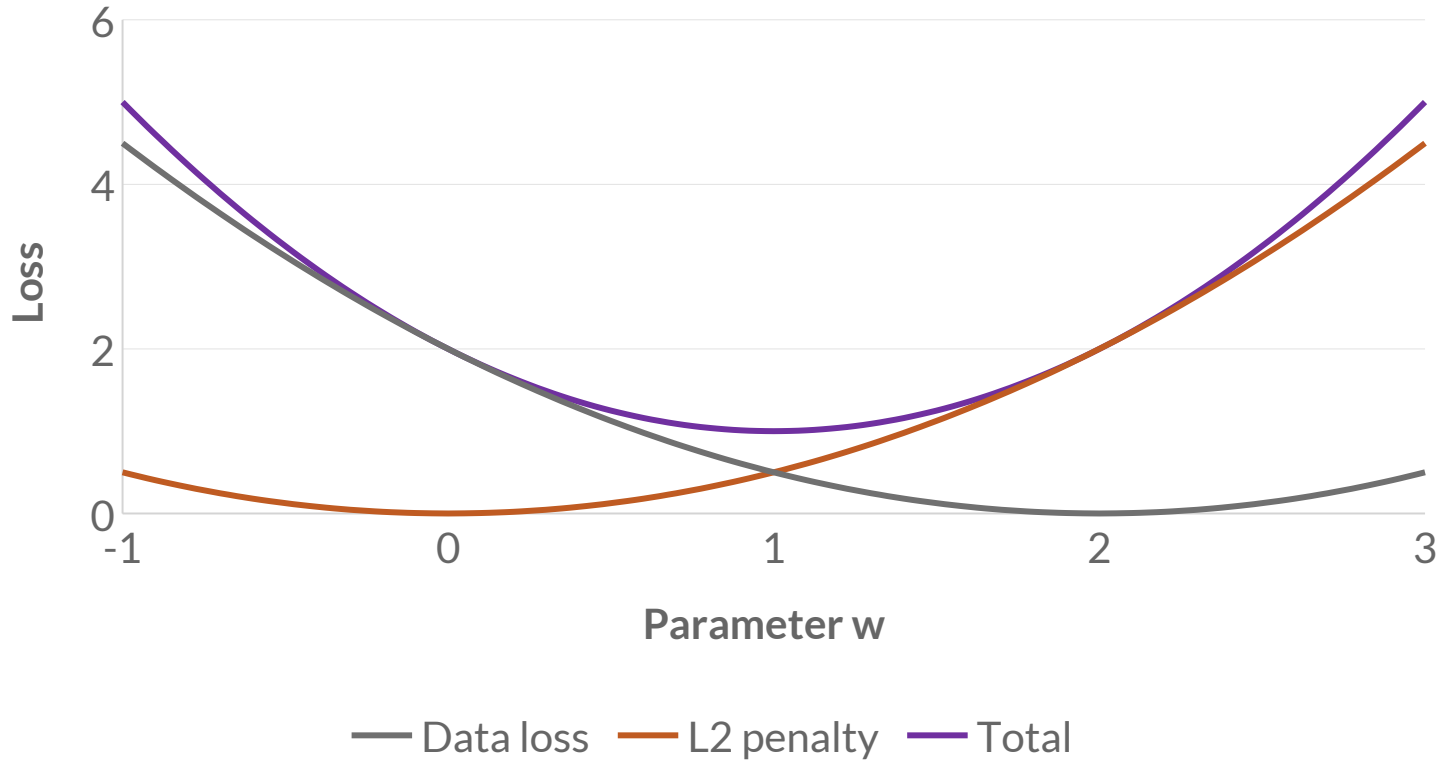
$$\min_{\theta} \quad \underbrace{L_{\text{train}}(\theta)}_{\text{fit to training data}} + \lambda \quad \underbrace{\Omega(\theta)}_{\text{regularizer}}$$

Training loss		Average error on observed samples
Regularizer		A preference among parameter settings
Regularization coefficient	λ	Balances data fit and regularization

L2 regularization

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{data}} + \frac{\lambda}{2}\|\theta\|_2^2$$

Squared L2 norm: sum of squared parameters

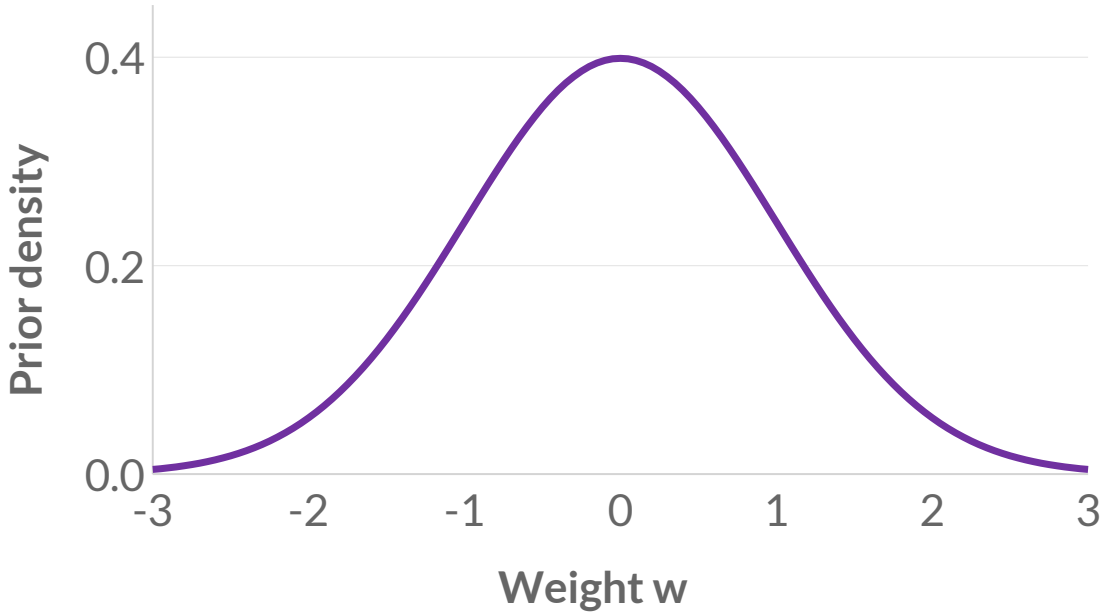


- Penalize large parameter magnitudes.
- Larger λ penalizes large weights more.

L2 regularization as a prior

A prior expresses a preference before seeing the training data.

Gaussian prior



This prior favors small weights.

The negative log of this prior gives a squared-weight penalty:

$$-\log p(w) = \frac{w^2}{2\sigma^2} + \text{constant}$$

σ sets the prior's spread

A narrower prior gives a larger regularization coefficient.

Weight decay in SGD

Plain SGD, without momentum

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

$$\theta_t = (1 - \eta_t\lambda)\theta_{t-1} - \eta_t g_t$$

Zero data gradient

$$\eta = \lambda = 0.1 \quad \Longrightarrow \quad \theta^+ = 0.99\theta$$

$$1 \longrightarrow 0.99, \qquad 10 \longrightarrow 9.9$$

In plain SGD, L2 regularization is proportional weight decay.

L2 regularization and Adam's moments

$$q_t = g_t + \lambda\theta_{t-1}$$

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)q_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)q_t^2$$

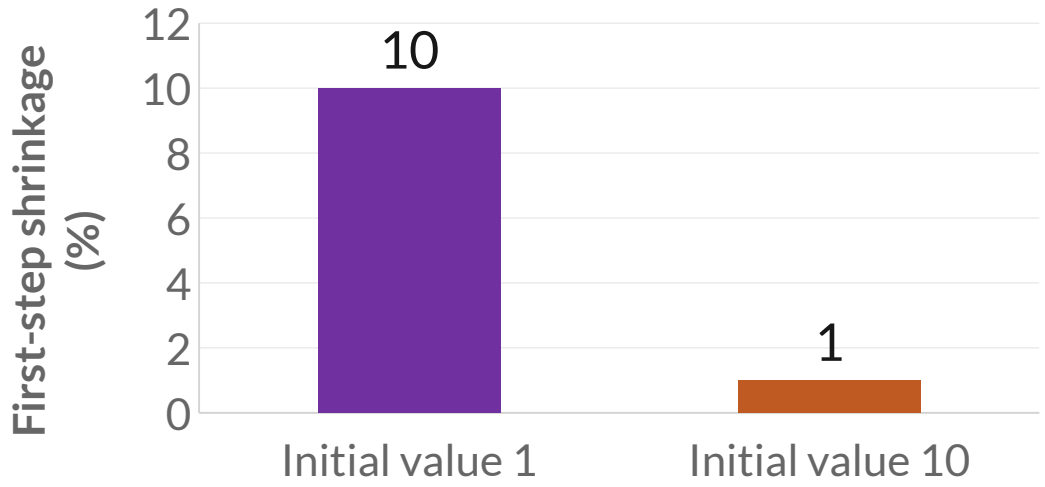
- Adam + L2 feeds the penalty gradient into both moments.
- This does not implement proportional weight decay.

L2 and proportional weight decay

$$\theta_0 = (1, 10), \quad g_{\text{data}} = (0, 0), \quad \eta = \lambda = 0.1$$

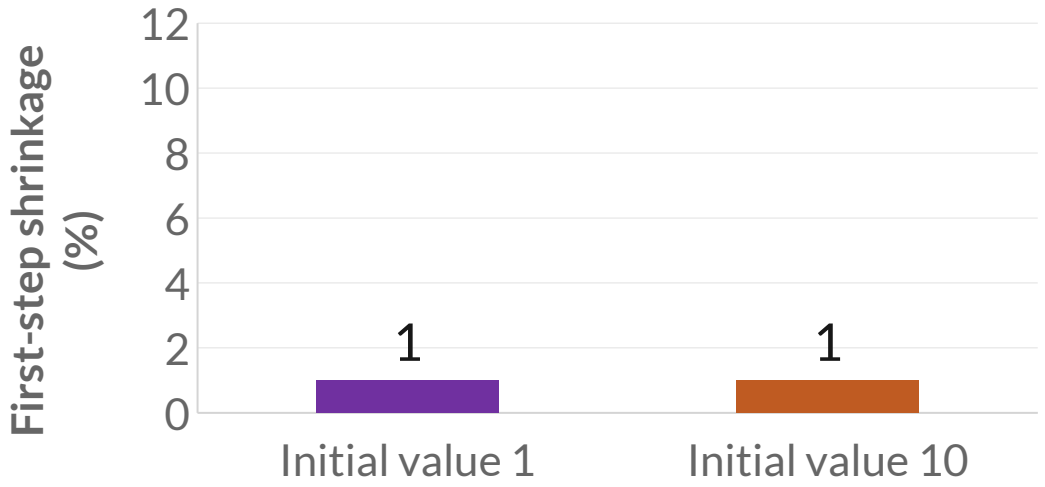
Zero initial moments

Adam + L2



$$\theta_1 \approx (0.9, 9.9)$$

Proportional weight decay



$$\theta_1 = (0.99, 9.9)$$

AdamW

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

Moment estimates	Keep the penalty out of both histories
Decoupled weight decay	Apply the same proportional shrinkage
Separate controls	Tune learning rate and decay coefficient

Adam + L2 and AdamW

One update from the same parameter and moment state

$$g = \nabla \mathcal{L}_{\text{data}}(\theta)$$

	Adam + L2	**AdamW**
1	$q = g + \lambda\theta$	$q = g$
2	$m \leftarrow \beta_1 m + (1-\beta_1) q$	$m \leftarrow \beta_1 m + (1-\beta_1) q$
3	$v \leftarrow \beta_2 v + (1-\beta_2) q^2$	$v \leftarrow \beta_2 v + (1-\beta_2) q^2$
4	$\widehat{m} = m/(1-\beta_1^t), \quad \widehat{v} = v/(1-\beta_2^t)$	$\widehat{m} = m/(1-\beta_1^t), \quad \widehat{v} = v/(1-\beta_2^t)$
5	$\theta \leftarrow \theta - \eta \widehat{m}/(\sqrt{\widehat{v}} + \varepsilon)$	$\theta \leftarrow (1-\eta\lambda)\theta - \eta \widehat{m}/(\sqrt{\widehat{v}} + \varepsilon)$

AdamW in PyTorch

```
optimizer = torch.optim.AdamW(
    model.parameters(), lr=0.001,
    betas=(0.9, 0.999), eps=1e-8,
    weight_decay=0.01,
)

optimizer.zero_grad()
loss = criterion(model(xb), yb)
loss.backward()
optimizer.step()
```

The loss is the data loss. AdamW applies weight decay in step().

Optimizer starting points

Starting points for a small network trained from scratch

SGD with momentum

$$\eta = 10^{-2}, \qquad \mu = 0.9$$

Try nearby orders of magnitude

$$\eta \in \{10^{-3}, 10^{-2}, 10^{-1}\}$$

AdamW

$$\eta = 10^{-3}, \quad (\beta_1, \beta_2) = (0.9, 0.999)$$

$$\varepsilon = 10^{-8}, \qquad \lambda = 10^{-2}$$

Adjust learning rate first. Keep momentum / betas fixed initially.

For finetuning, start from a recipe for that model and task.

Next: initialization and training stability

- Vanishing and exploding gradients
- Weight initialization and normalization
- Learning-rate schedules and gradient clipping