DDA4220 · Fall 2026

Lecture 4

Deep Learning and Applications

Training neural networks II: initialization and normalization

Zhen Liu

CUHK-Shenzhen

Lecture outline

01 Initialization

02 Normalization

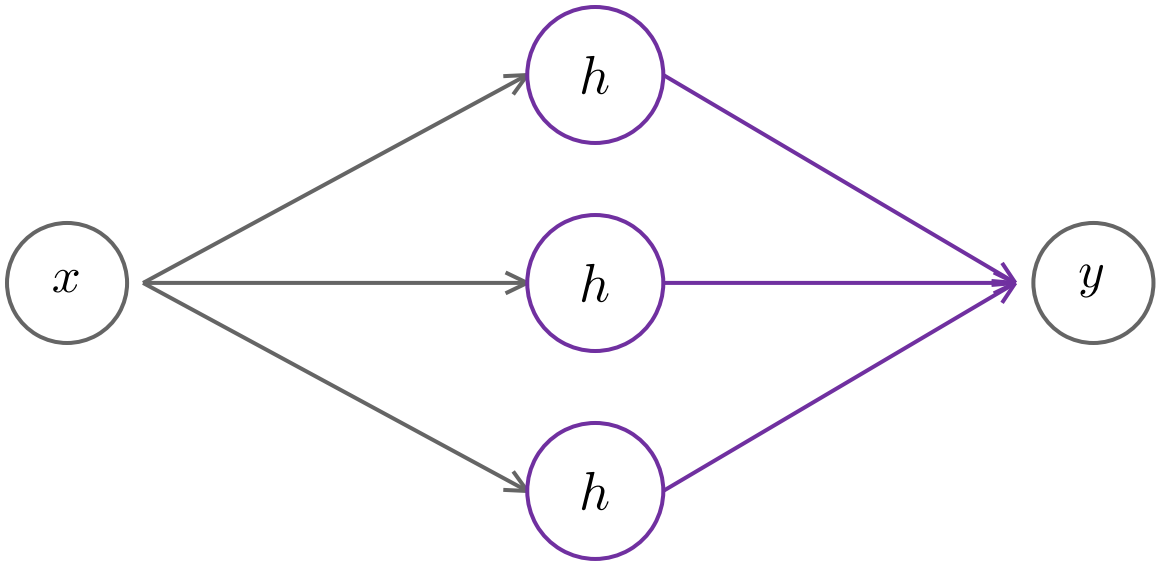
03 Schedules and batches

04 Clipping and stability

* Optional

Identical hidden units

Three identical hidden features



Same input weights and bias

One distinct feature

Effective width: 1

Symmetry survives gradient descent

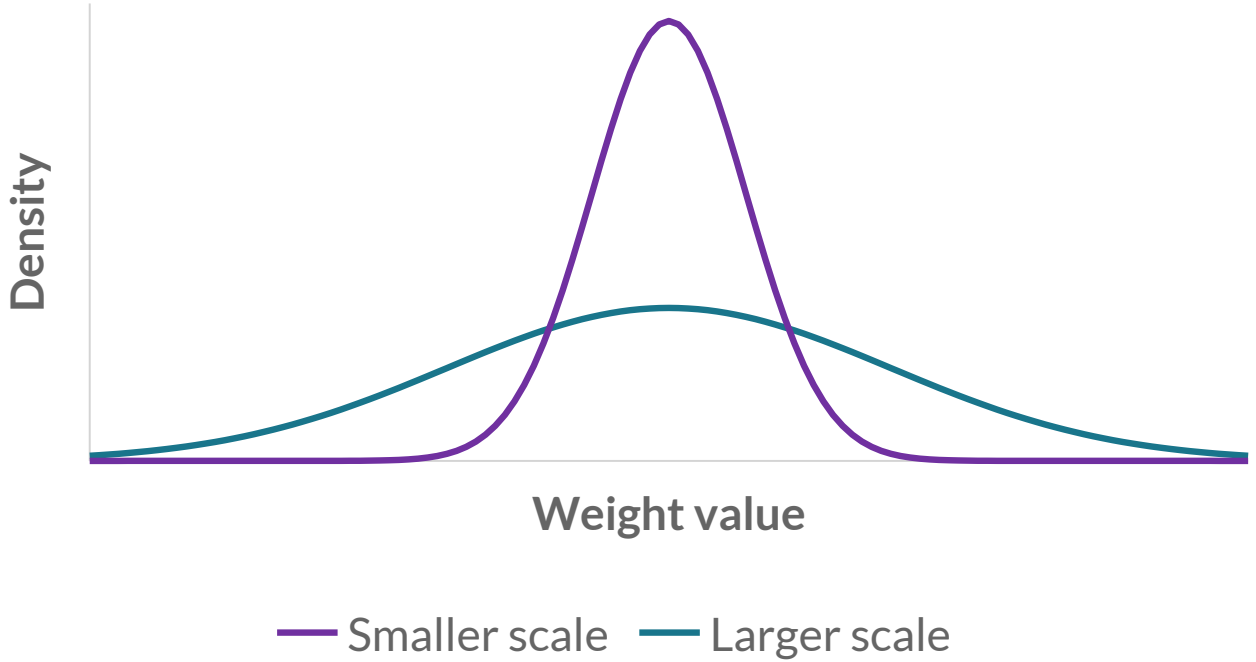
Identical incoming and outgoing connections give identical gradients.

Before the update **After the update**

Gradient descent preserves this symmetry.

Initialize hidden units with different random weights.

Randomness and weight scale



$$w_{ij} \sim \mathcal{N}(0, s^2), \qquad b_i = 0$$

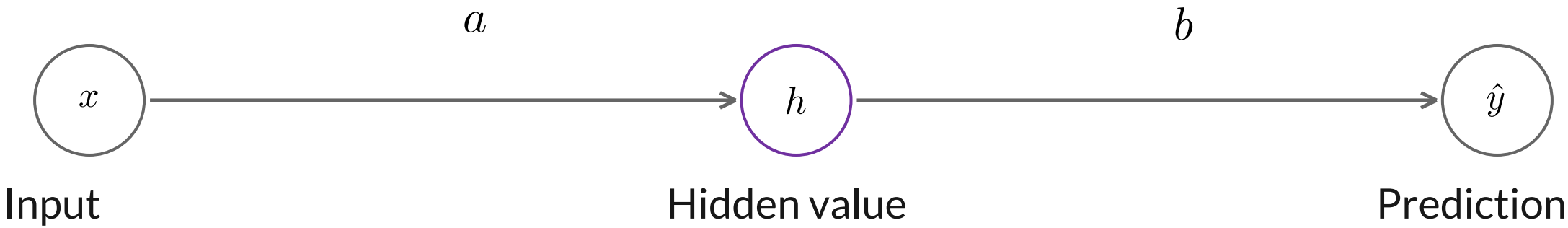
s: weight standard deviation

Each weight gets its own draw.

To choose the scale, examine a network with just two weights.

Two linear layers

Weight scale can matter even without activation functions.



One training example $x = 1, \qquad y = 1$

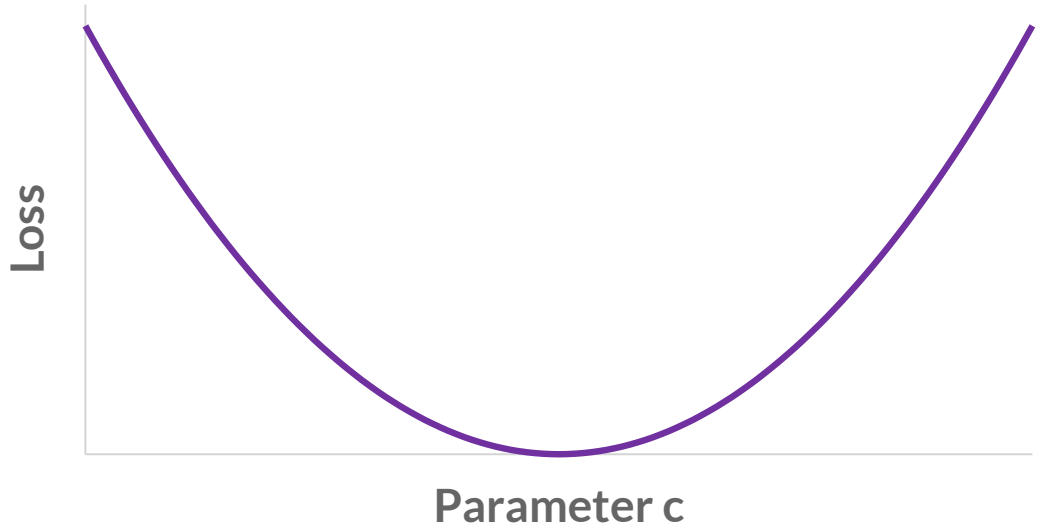
Squared-error loss $L(a, b) = \frac{1}{2}(ab - 1)^2$

Same capacity, different optimization

One linear layer

$$\hat{y} = cx$$

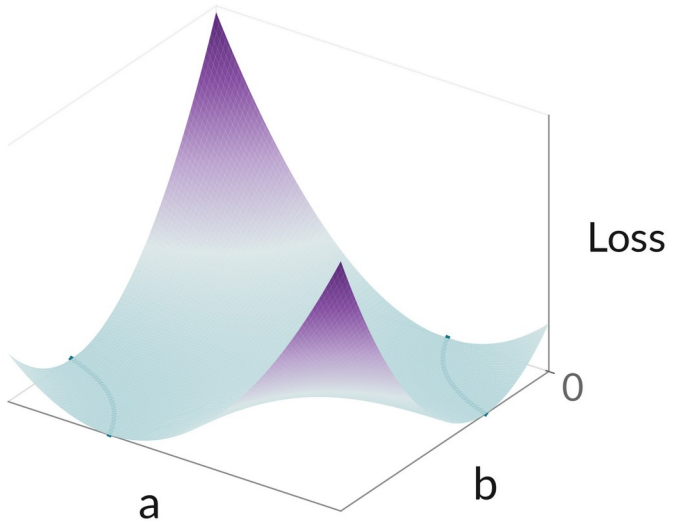
$$c = ab$$



$$L(c) = \frac{1}{2}(c-1)^2$$

Two linear layers

$$\hat{y} = abx$$



$$L(a,b) = \frac{1}{2}(ab-1)^2$$

Small initial weights

Training loss

$$L(a, b) = \frac{1}{2}(ab - 1)^2$$

Gradient for a

$$\frac{\partial L}{\partial a} = (ab - 1)b$$

Gradient for b

$$\frac{\partial L}{\partial b} = (ab - 1)a$$

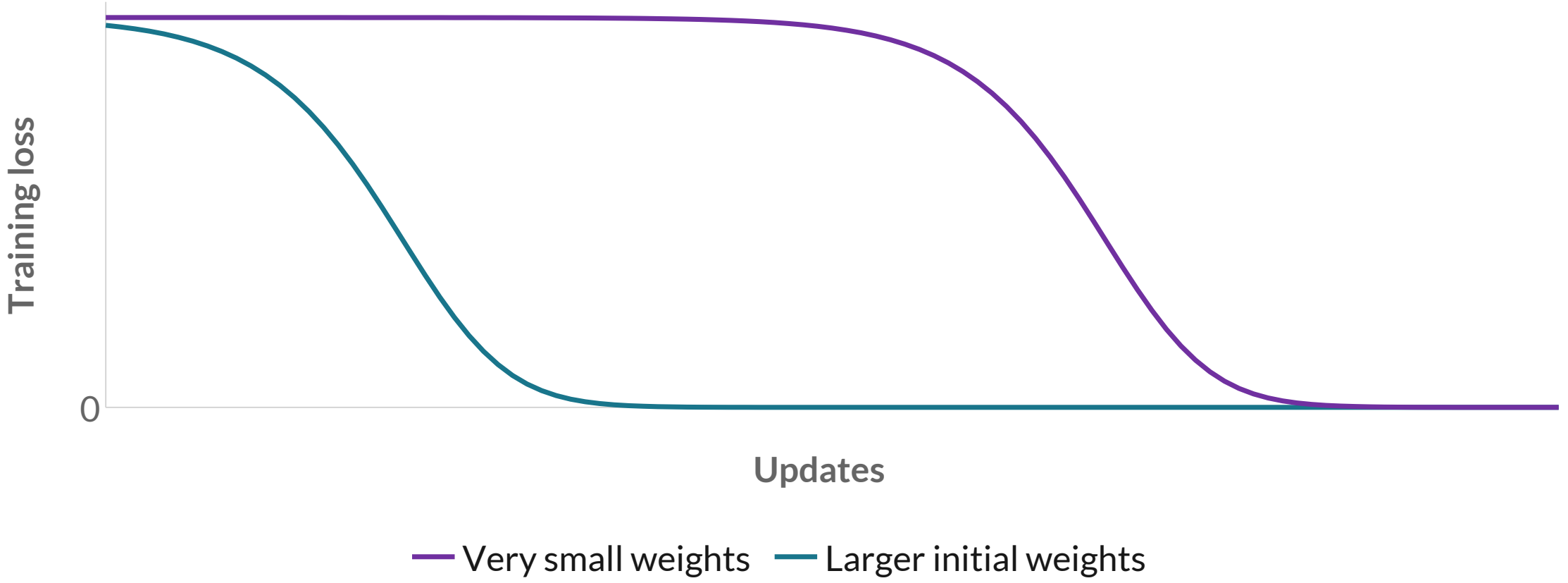
Prediction error × the other layer's weight

Large prediction error can coexist with small gradients.

Learning from small weights

Computed for the two-layer linear model

Same learning rate



Same function, different scales

Original

$$x \xrightarrow{a} h \xrightarrow{b} \hat{y} = abx$$

Rescaled

$$x \xrightarrow{a'=qa} h' \xrightarrow{b'=b/q} \hat{y} = abx$$

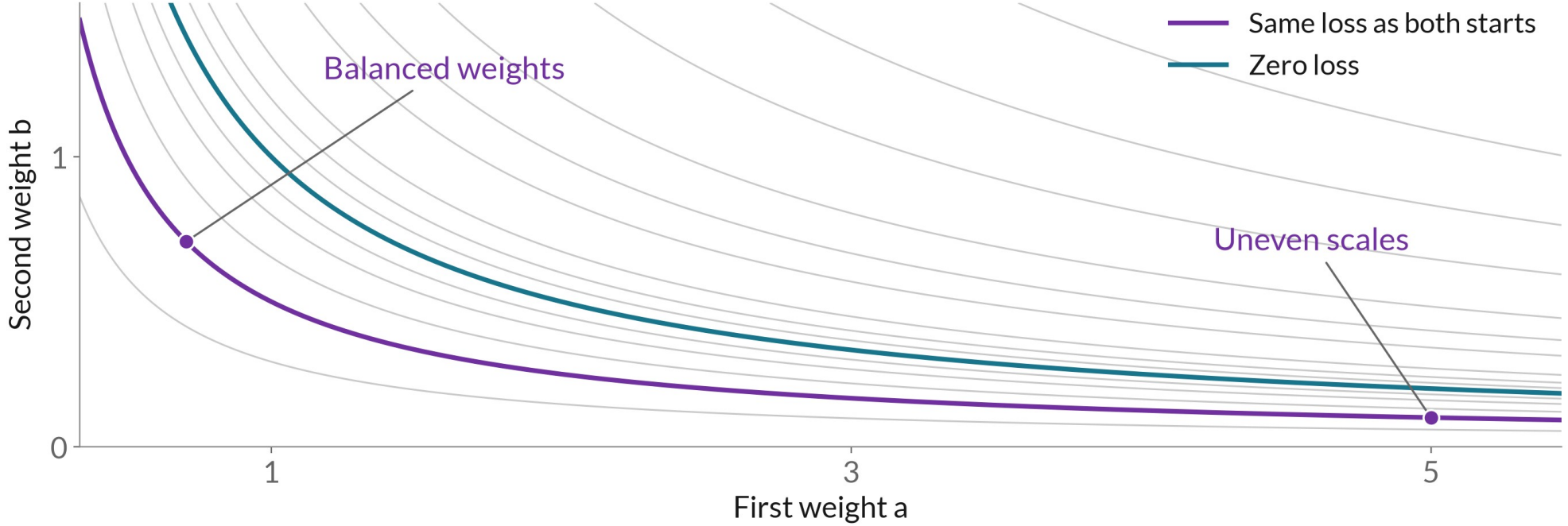
q > 0: a scale factor

Every prediction is unchanged.

$$\frac{\partial L}{\partial a'} = \frac{1}{q}\frac{\partial L}{\partial a}, \qquad \frac{\partial L}{\partial b'} = q\frac{\partial L}{\partial b}$$

Initialization on the loss landscape

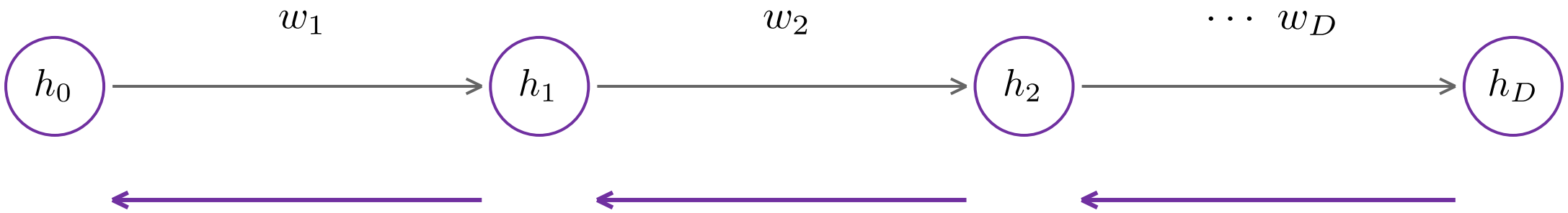
$$L(a, b) = \tfrac{1}{2}(ab - 1)^2$$



Points on one contour share the same loss.

Minimum curve $ab = 1 \qquad (L = 0)$

Linear weights through depth



Backward

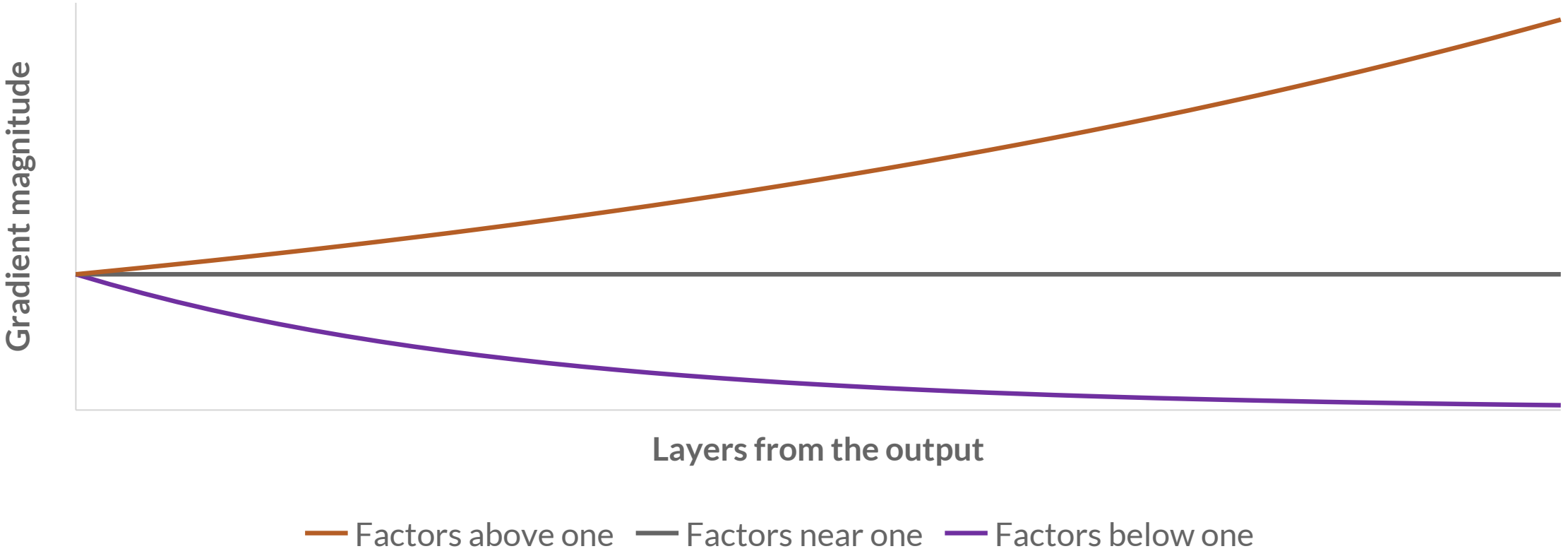
One linear layer

$$g_{\ell-1} = w_\ell g_\ell, \qquad g_\ell = \frac{\partial L}{\partial h_\ell}$$

D linear layers

$$g_0 = g_D \, w_1 w_2 \cdots w_D$$

Vanishing and exploding gradients



Small changes in each factor compound through many layers.

Weight matrices and gradient scale

Two-dimensional example

$$g_{\text{in}} = W^{\mathsf{T}} g_{\text{out}}$$

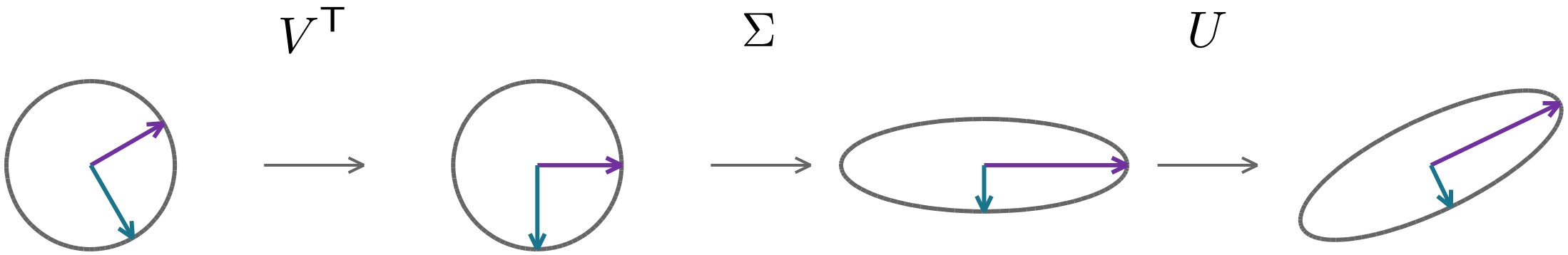
Equal-length gradients

Different resulting lengths

One matrix can shrink some directions and enlarge others.

Singular value decomposition (SVD)

$$W^\mathsf{T} = U\Sigma V^\mathsf{T}$$



Rotate / reflect Stretch along axes Rotate / reflect

Singular values: the diagonal stretch factors in Σ

Gradient scaling through depth

Largest singular value = greatest possible stretch

One linear layer

$$\|W^{\mathsf{T}} g\| \leq \sigma_{\max}(W)\, \|g\|$$

D linear layers

$$\|g_0\| \leq \sigma_{\max}(W_1) \cdots \sigma_{\max}(W_D)\, \|g_D\|$$

‖ g ‖ : gradient length

Reaching the bound also requires the stretching directions to align.

Gradients through nonlinear layers

$$z_\ell = W_\ell h_{\ell-1} + b_\ell, \qquad h_\ell = \phi(z_\ell)$$

Weighted sum z → activation → hidden output h

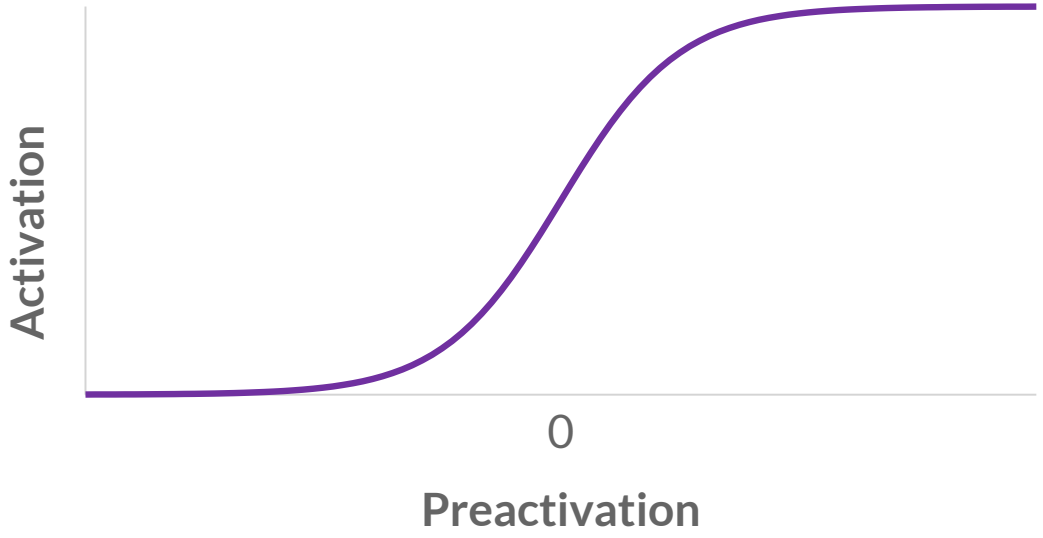
$$g_{\ell-1} = \underbrace{W_\ell^\top}_{\text{linear weights}} \left(\underbrace{\phi'(z_\ell)}_{\text{activation slopes}} \odot \underbrace{g_\ell}_{\text{arriving gradient}} \right)$$

⊙ multiply matching entries

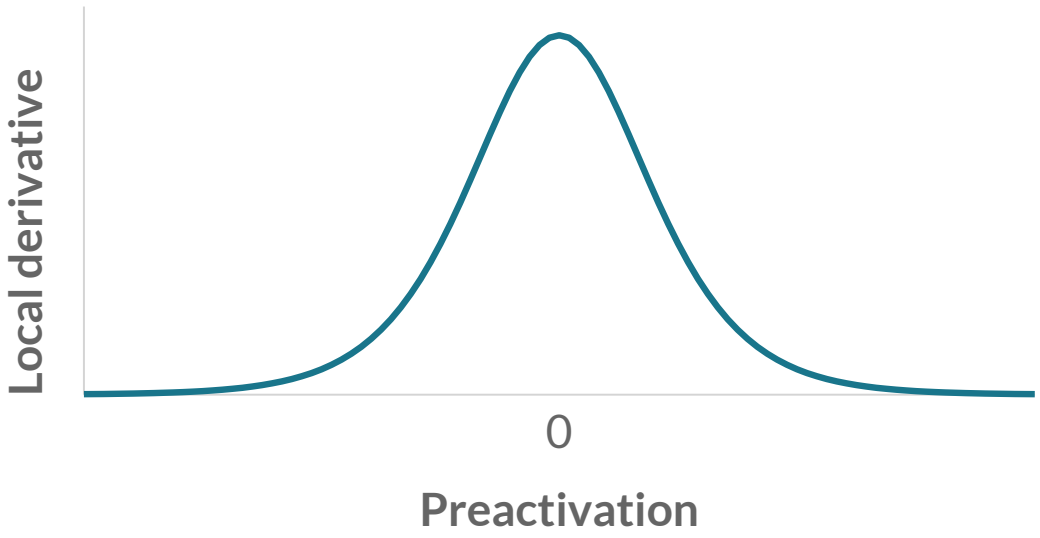
g: gradient of the loss with respect to h

Saturation and small derivatives

Sigmoid



Its derivative

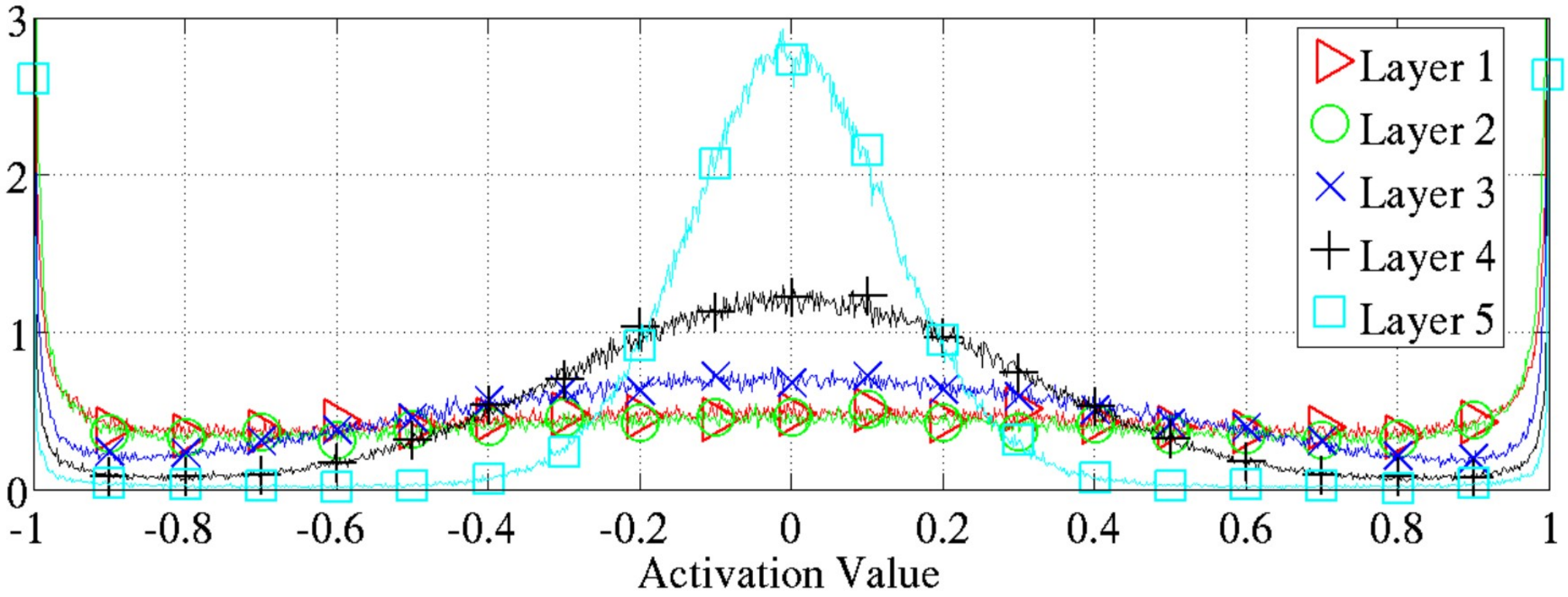


z: input to the activation

$$\sigma'(z) = \sigma(z)(1 - \sigma(z)) \leq \frac{1}{4}$$

Saturation during training

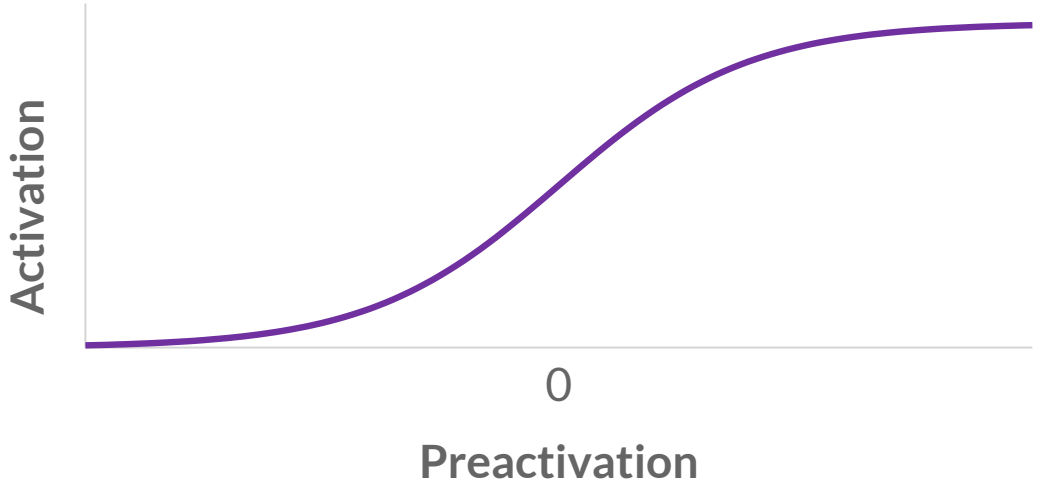
In this experiment, some layers concentrate near the tanh limits.



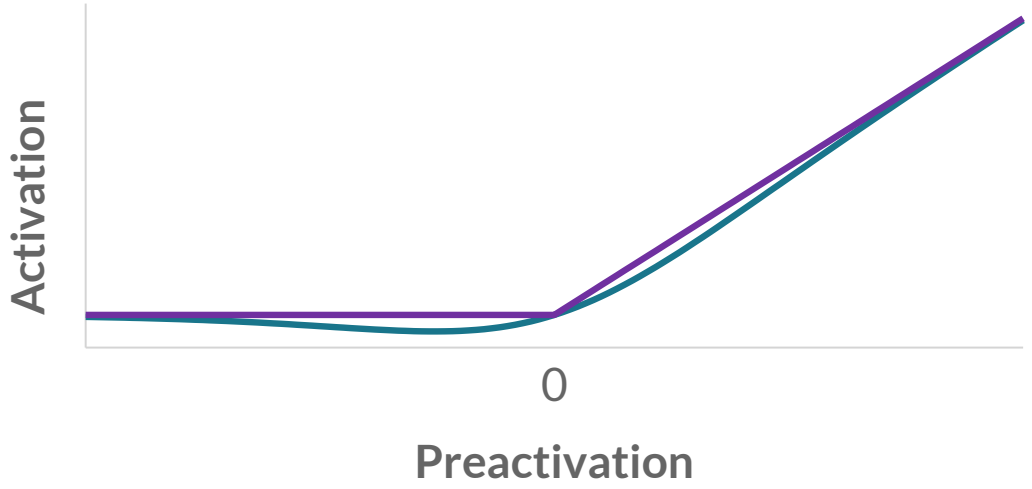
Source: Glorot & Bengio (2010)

ReLU and saturation

Sigmoid



ReLU and SiLU



ReLU SiLU

Small slopes in both tails

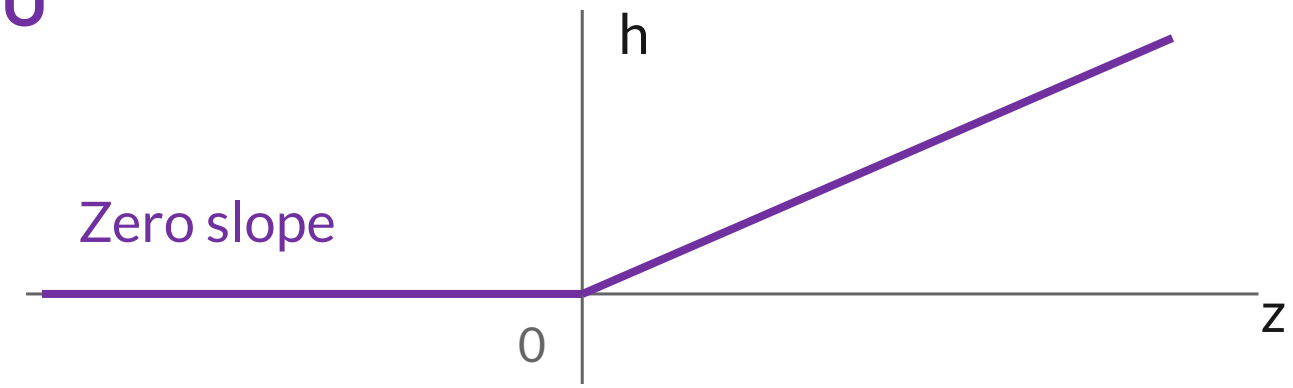
$$\mathrm{ReLU}'(z) = \begin{cases} 0, & z < 0, \\ 1, & z > 0, \end{cases}$$

ReLU still depends on input scale and which units are active.

LeakyReLU has a nonzero slope below zero. SiLU and GELU are smooth.

Dying ReLU units

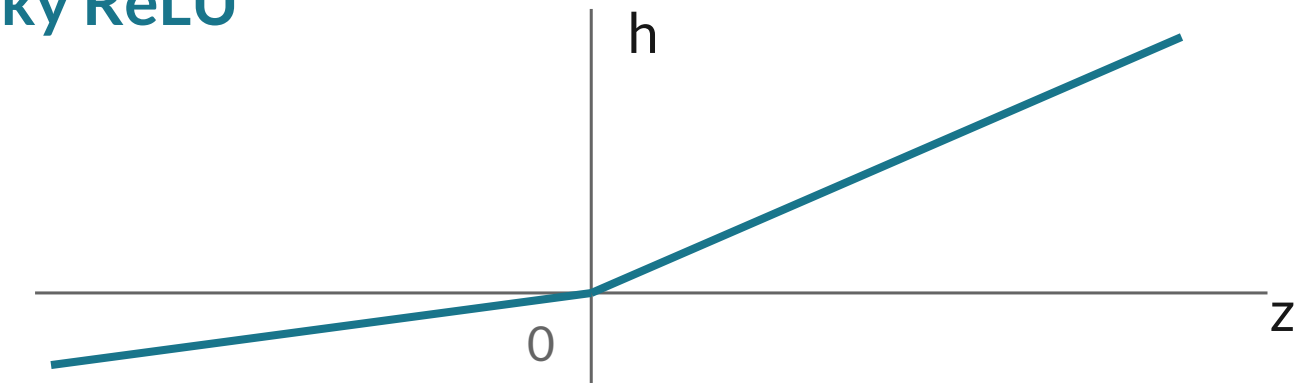
ReLU



$$h = \max(0, z)$$

$$\frac{\partial h}{\partial z} = 0 \quad (z < 0)$$

Leaky ReLU



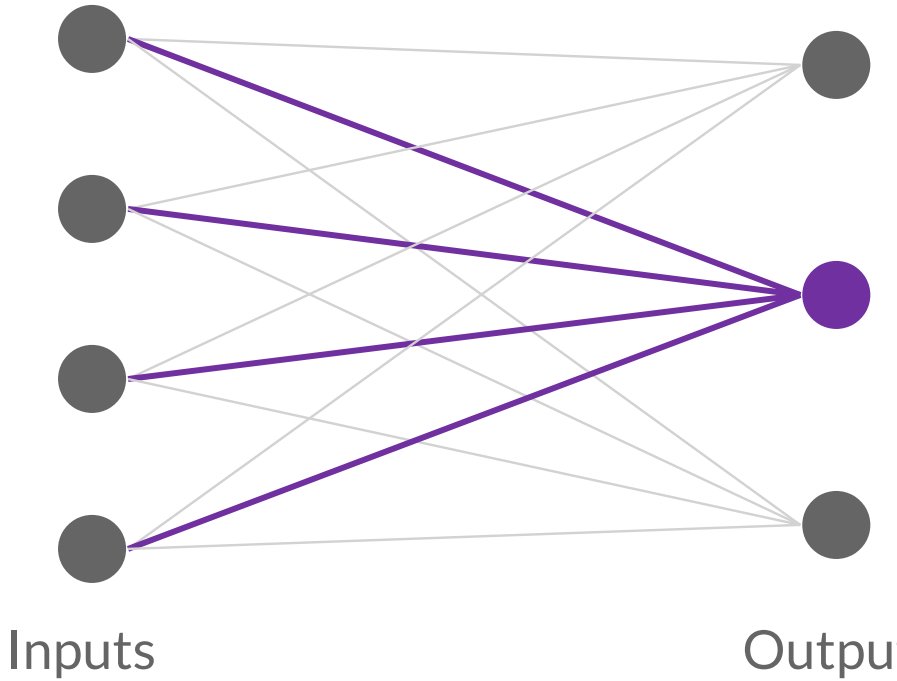
Small nonzero slope

Negative preactivation on all training inputs blocks the unit's gradient.

Fan-in and fan-out

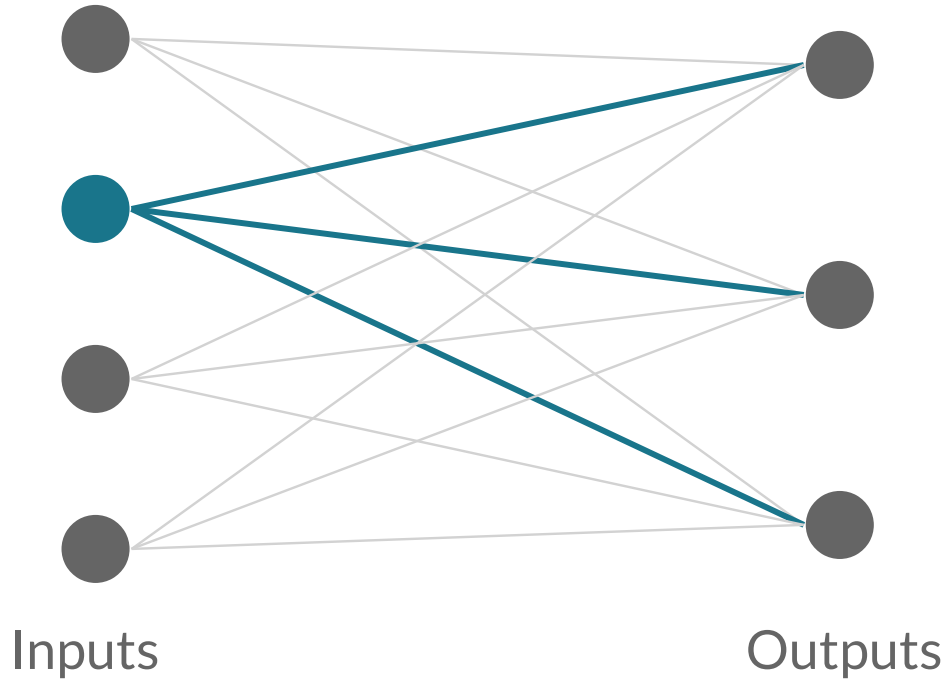
Fan-in

Number of inputs to one unit



Fan-out

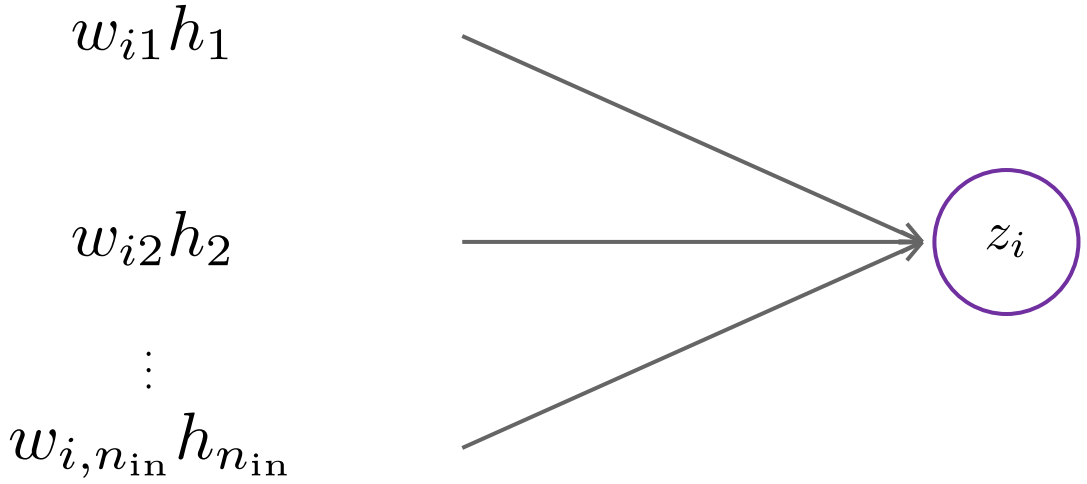
Number of outputs reached by one unit



$$W \in \mathbb{R}^{n_{\text{out}} \times n_{\text{in}}}$$

Scale of a weighted sum

At initialization: independent, zero-mean weights



Positive and negative contributions can cancel.

Squares add on average.

s_w^2 : weight variance (average squared weight here)

Average squared size

$$\mathbb{E}[z_i^2] \approx n_{\text{in}} s_w^2 \, \mathbb{E}[h_j^2]$$

Preserving forward scale

Match the average squared size before and after the linear layer.

Scale multiplier

$$\frac{\mathbb{E}[z_i^2]}{\mathbb{E}[h_j^2]} \approx n_{\text{in}} s_w^2$$

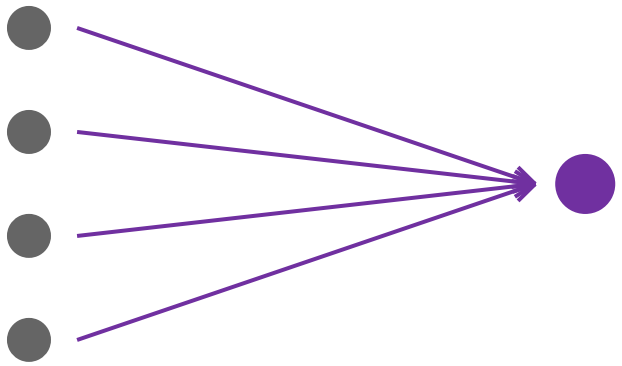
Set it near one

$$n_{\text{in}} s_w^2 \approx 1 \quad \Longrightarrow \quad s_w \approx \frac{1}{\sqrt{n_{\text{in}}}}$$

More inputs require smaller individual weights.

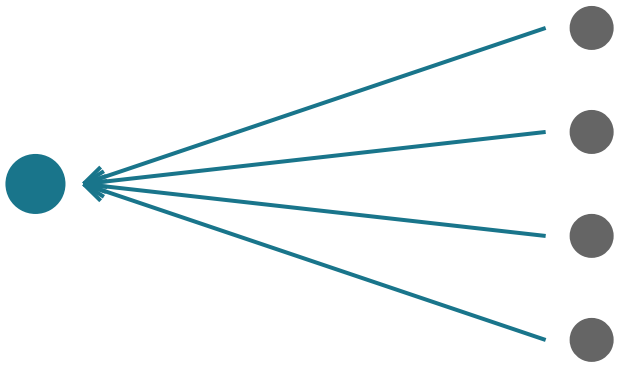
Forward and backward scale

Forward: fan-in



$$\text{forward multiplier} \approx n_{\text{in}} s_w^2$$

Backward: fan-out



$$\text{backward multiplier} \approx n_{\text{out}} s_w^2$$

Balance the two

$$\frac{n_{\text{in}} + n_{\text{out}}}{2} s_w^2 \approx 1$$

Xavier initialization

Weight variance

$$s_w^2 = \frac{2}{n_{\text{in}} + n_{\text{out}}}$$

Normal-distribution version

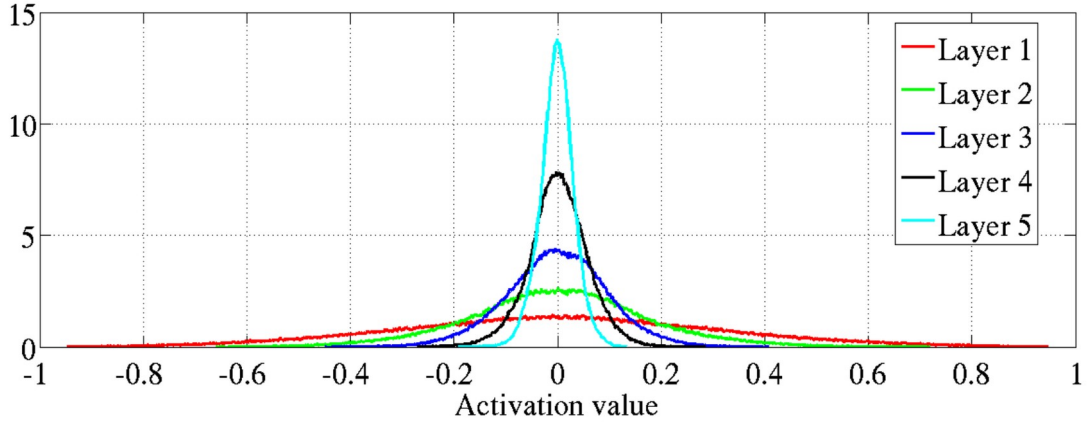
$$w_{ij} \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}} + n_{\text{out}}}\right)$$

Balances the numbers of incoming and outgoing connections.

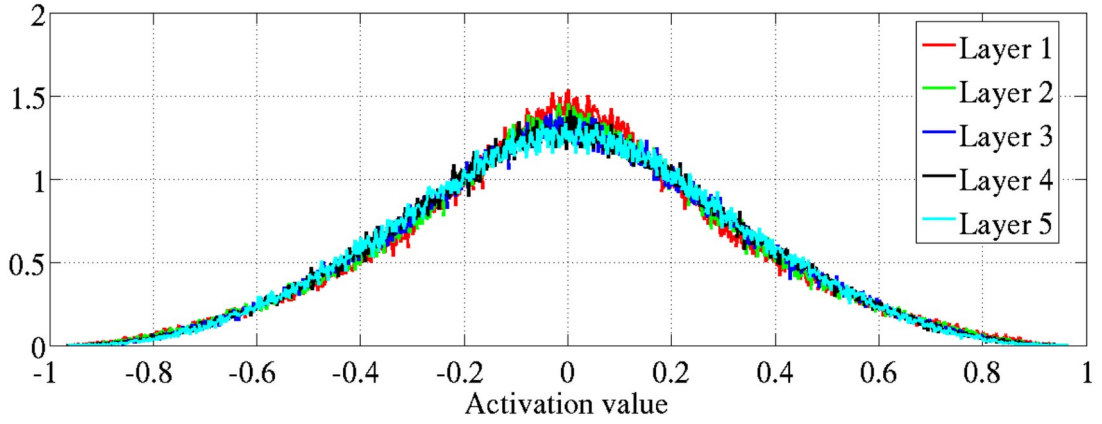
Also called Glorot initialization

Activation distributions at initialization

Baseline initialization



Xavier initialization



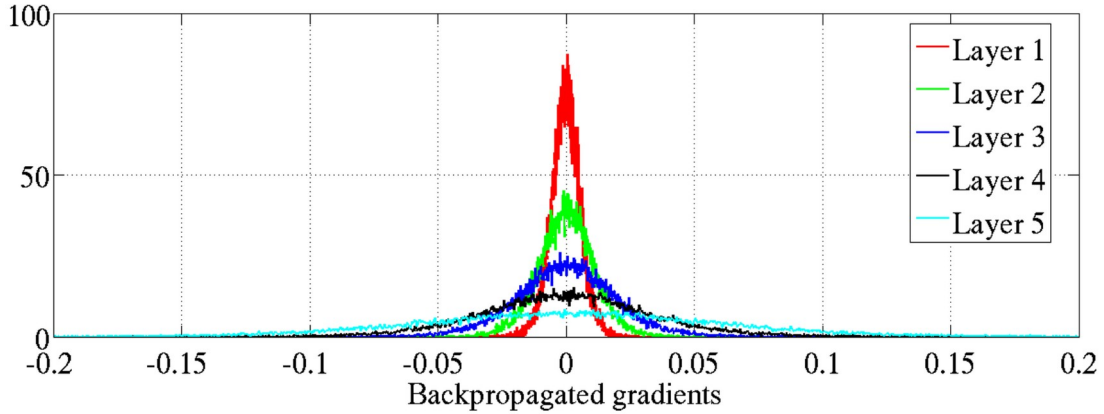
Layer 1 Layer 2 Layer 3 Layer 4 Layer 5

In this example, the baseline initialization shrinks later-layer activations.

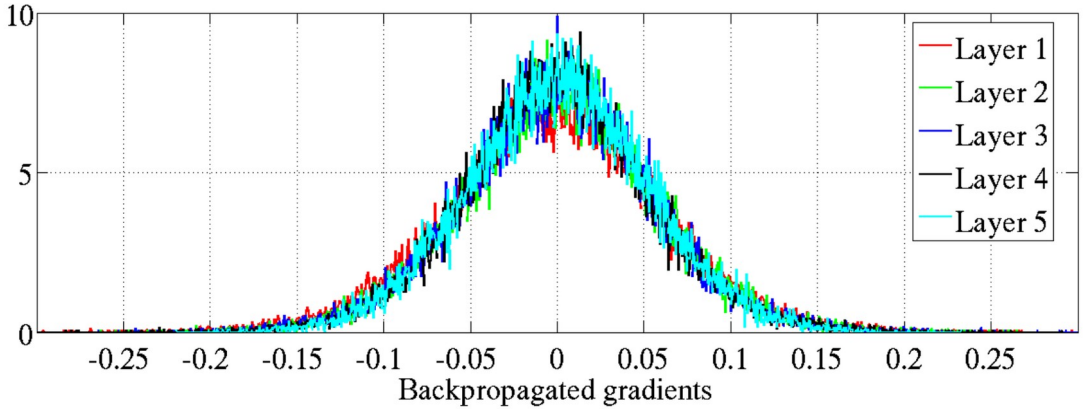
Glorot & Bengio · Initialization

Gradient distributions at initialization

Standard initialization



Xavier initialization

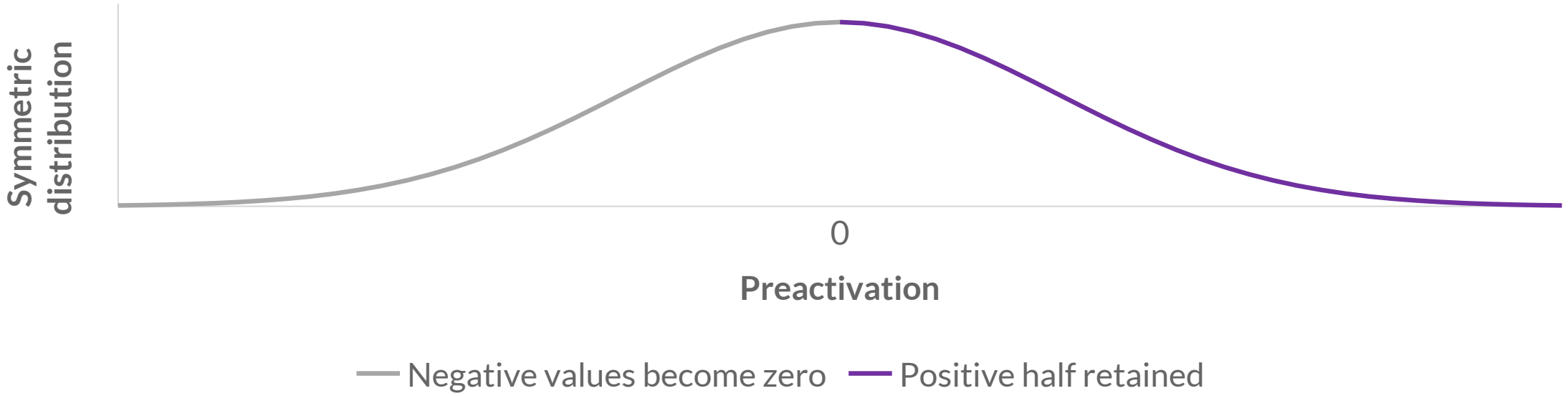


Layer 1 Layer 2 Layer 3 Layer 4 Layer 5

Xavier keeps the gradient spread more comparable across layers.

Glorot & Bengio · Initialization

ReLU and average squared magnitude



$$\mathbb{E}[\mathrm{ReLU}(z)^2] = \tfrac{1}{2}\mathbb{E}[z^2]$$

This halves the second moment, not generally the variance.

From ReLU to He initialization

Linear weighted sum

$$\mathbb{E}[z_i^2] \approx n_{\text{in}} s_w^2 \, \mathbb{E}[h_{\text{in}}^2]$$

After ReLU

$$\mathbb{E}[h_{\text{out}}^2] = \tfrac{1}{2}\mathbb{E}[z_i^2]$$

Whole-layer multiplier

$$\frac{\mathbb{E}[h_{\text{out}}^2]}{\mathbb{E}[h_{\text{in}}^2]} \approx \tfrac{1}{2} n_{\text{in}} s_w^2$$

He initialization

Preserve squared scale

$$\frac{1}{2} n_{\text{in}} s_w^2 \approx 1$$

Solve for weight scale

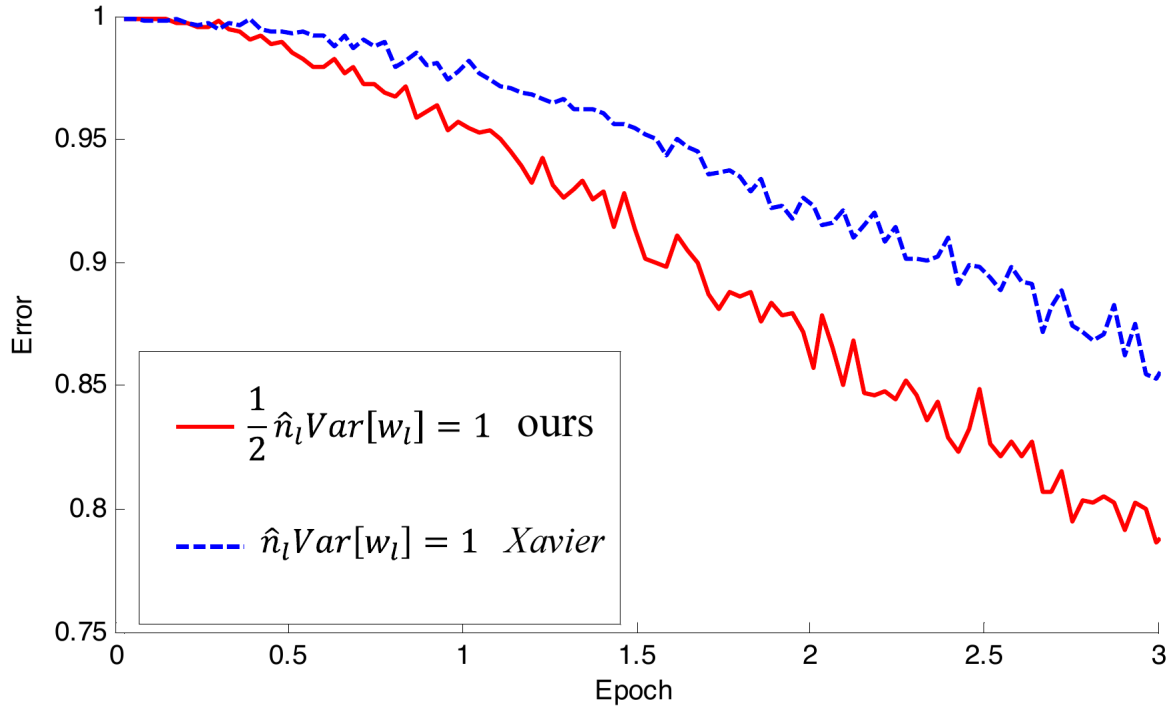
$$s_w^2 = \frac{2}{n_{\text{in}}}, \qquad s_w = \sqrt{\frac{2}{n_{\text{in}}}}$$

Twice the fan-in variance used for a linear layer.

He / Kaiming initialization for ReLU

Initialization in image classification

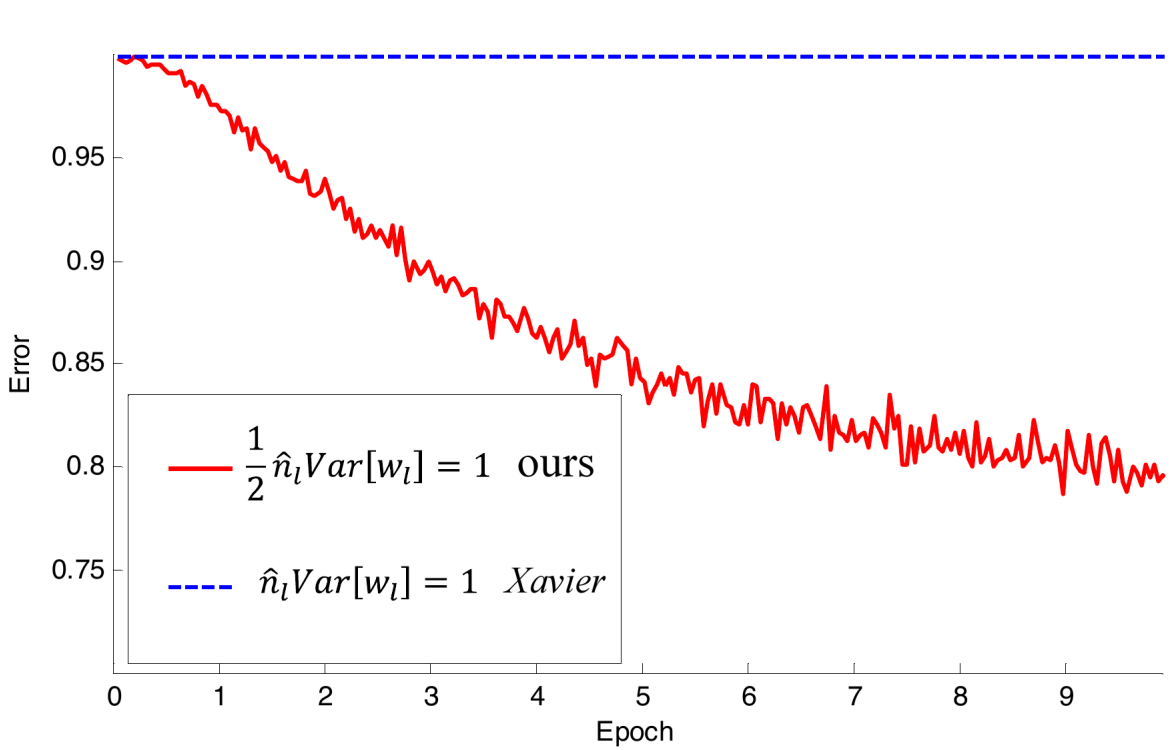
Task: predict the category of a photograph.



- 22-layer ReLU network.
- He (red); Xavier (blue).
- Error begins falling earlier with He initialization.

Source: He et al. (2015)

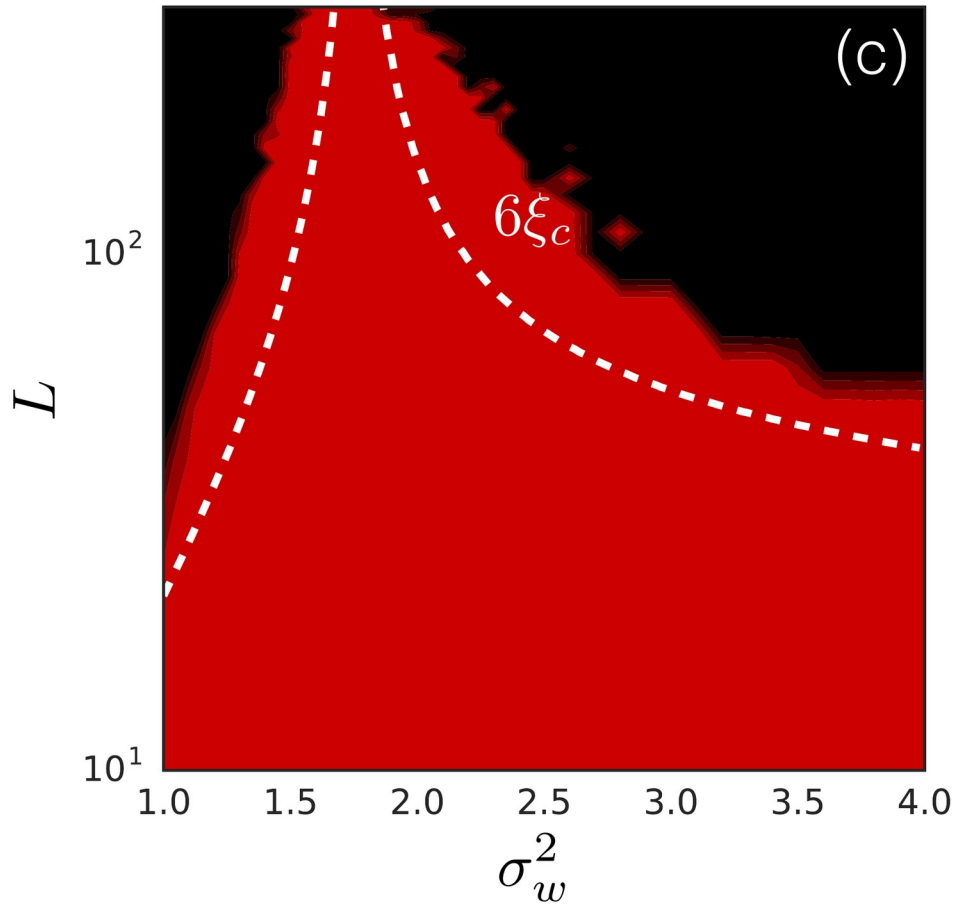
Initialization in a deeper network



- 30-layer ReLU network.
- Xavier initialization stalls in this experiment.
- The paper also reports diminishing gradients.

Source: He et al. (2015)

Initialization and trainable depth



- Horizontal: initialization scale. Vertical: number of layers.
- Digit-classification accuracy: red is high, black is low.
- The useful scale range narrows in deeper networks.

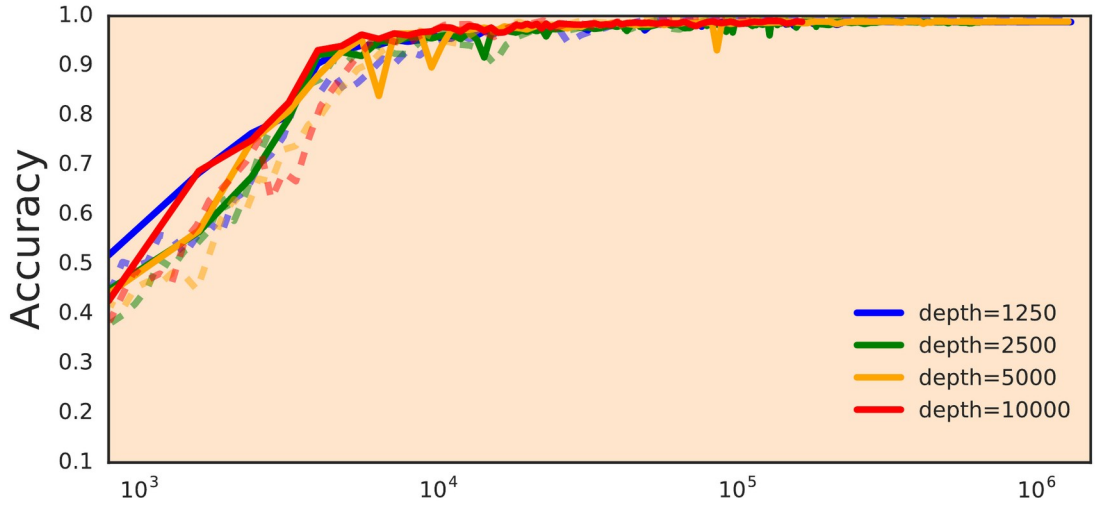
Source: Deep Information Propagation

Training 10,000-layer networks

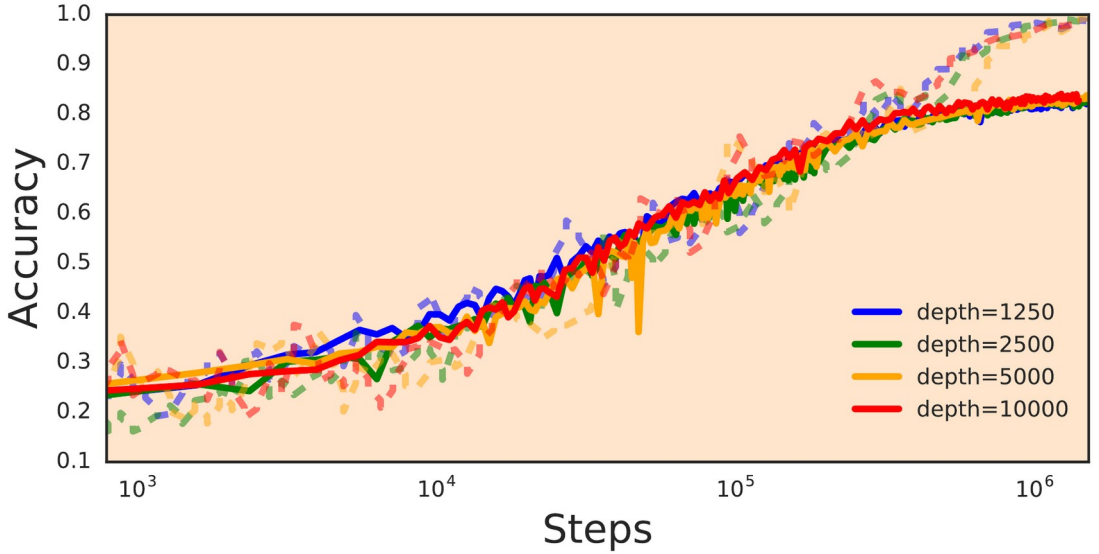
With carefully chosen initialization, even 10,000-layer networks can learn.

Handwritten digits

Solid: test accuracy · dashed: training accuracy



Small photographs



Source: Xiao et al. (2018)

What initialization controls

- Break symmetry between hidden units.
- Keep activations and gradients at useful initial scales.
- Training changes the weights: good initial scale is a starting condition.

Initialization in PyTorch

```
hidden = nn.Linear(input_dim, hidden_dim)
nn.init.kaiming_normal_(hidden.weight, nonlinearity="relu")
nn.init.zeros_(hidden.bias)

output = nn.Linear(hidden_dim, num_classes)
nn.init.xavier_normal_(output.weight)  # a linear output layer
nn.init.zeros_(output.bias)
```

Lecture outline

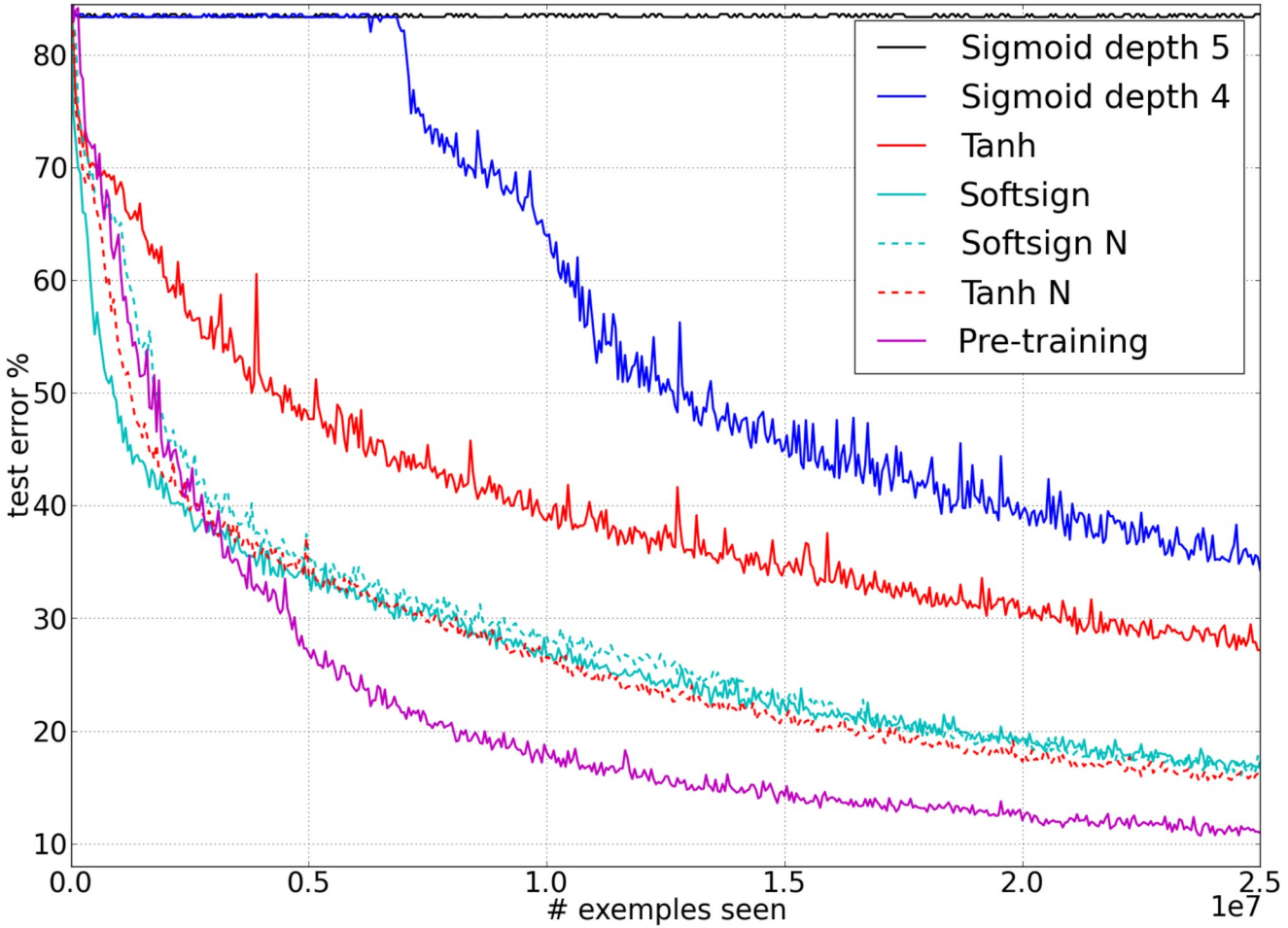
01 Initialization

02 Normalization

03 Schedules and batches

04 Clipping and stability

Training deep networks in the early 2010s



Even a few hidden layers could be difficult to train.

Five sigmoid layers

Almost no improvement

Four sigmoid layers

Long initial plateau

Understanding the difficulty of training deep feedforward neural networks (2010)

Input scale and gradient scale

Same measurement

Metres

Millimetres

Different numerical scales

$$\frac{\partial L}{\partial w_j} = \underbrace{\frac{\partial L}{\partial z}}_{\text{arriving gradient}} \underbrace{x_j}_{\text{input value}}$$

Input standardization

$$\widetilde{x}_j = \frac{x_j - \mu_j}{\sqrt{v_j + \varepsilon}}$$

Subtract the training mean; divide by the training scale.

μ : training mean v: training variance ε : small positive constant

Training inputs

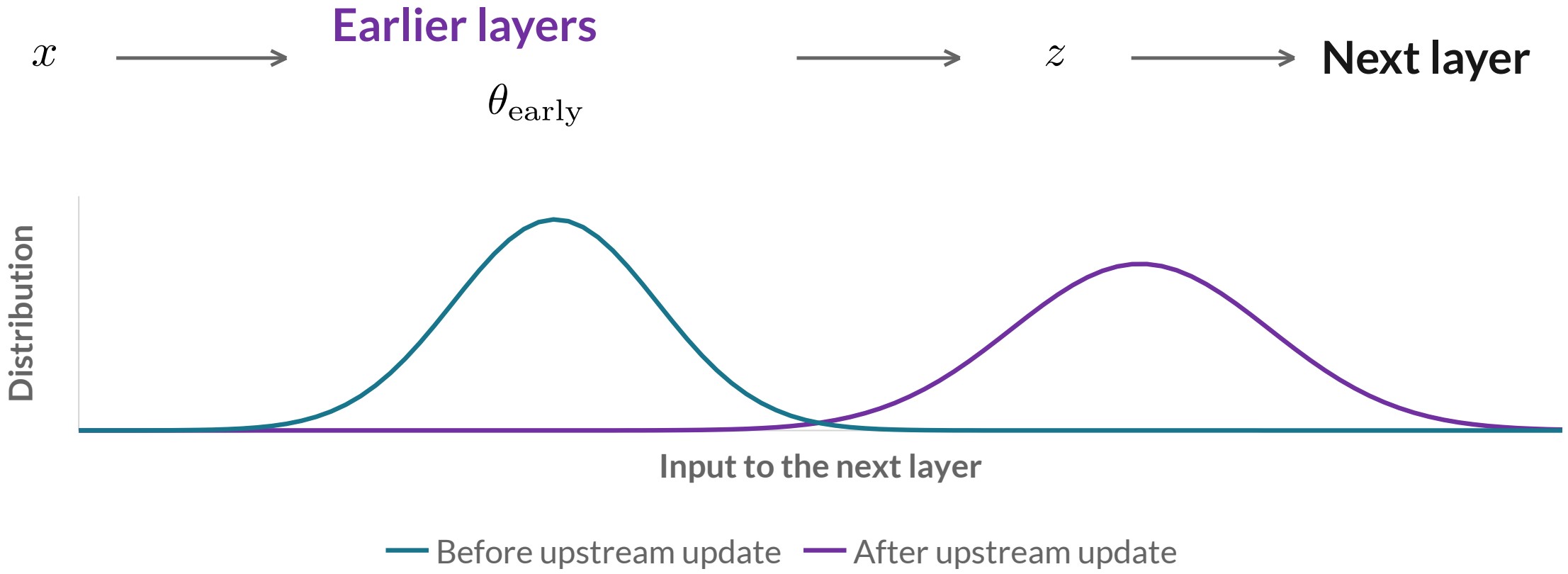
Estimate mean and variance

Validation / test inputs

Reuse the stored estimates

Changing inputs inside the network

The training data stay fixed. Inputs to later layers change.



Normalizing hidden activations

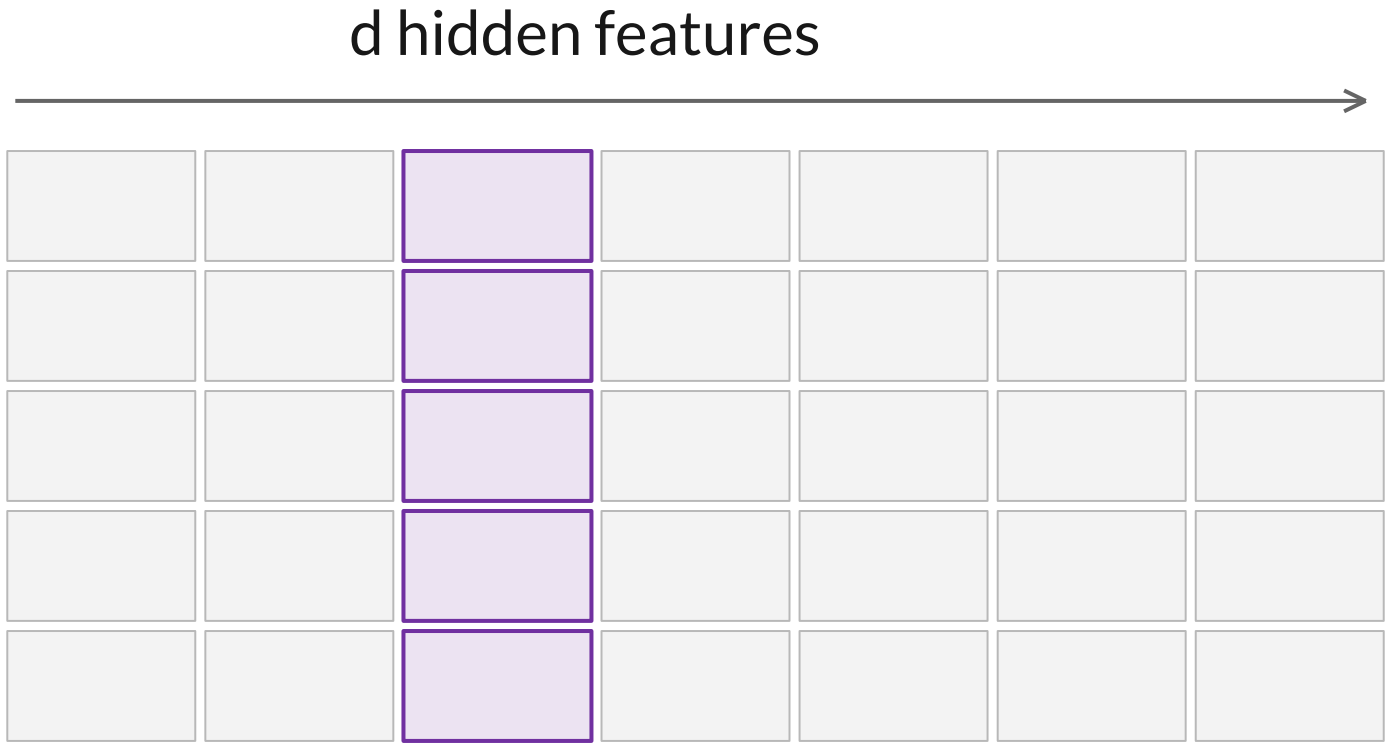
Initialization sets the starting scale.

Normalization recomputes a reference center and scale during training.

$$z \longmapsto \widehat{z} = \frac{z - \text{mean}}{\sqrt{\text{variance} + \varepsilon}}$$

Choose which values share the mean and variance.

One feature across a minibatch



$Z \in \mathbb{R}^{m \times d}$

One cell: one feature value for one example

One mean and variance for the highlighted column

Batch statistics

Batch mean

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} z_i$$

Batch variance

$$v_B = \frac{1}{m}\sum_{i=1}^{m}(z_i - \mu_B)^2$$

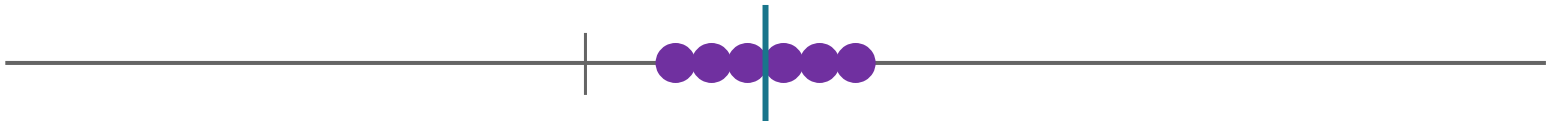
z_i: value of this feature for example i

m: number of examples B: current minibatch

Centering and rescaling

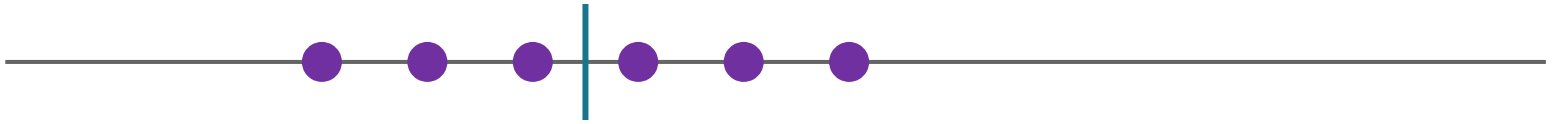
$$\widehat{z}_i = \frac{z_i - \mu_B}{\sqrt{v_B + \varepsilon}}$$

Before normalization



batch mean μB

After normalization



center ≈ 0

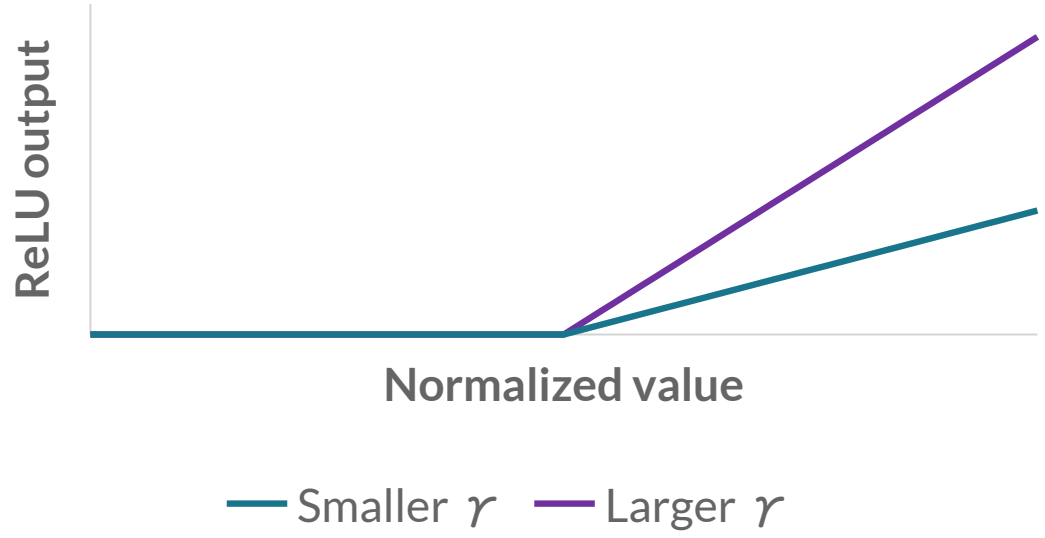
Same ordering; a new center and scale. ε prevents division by zero.

Learned scale and shift

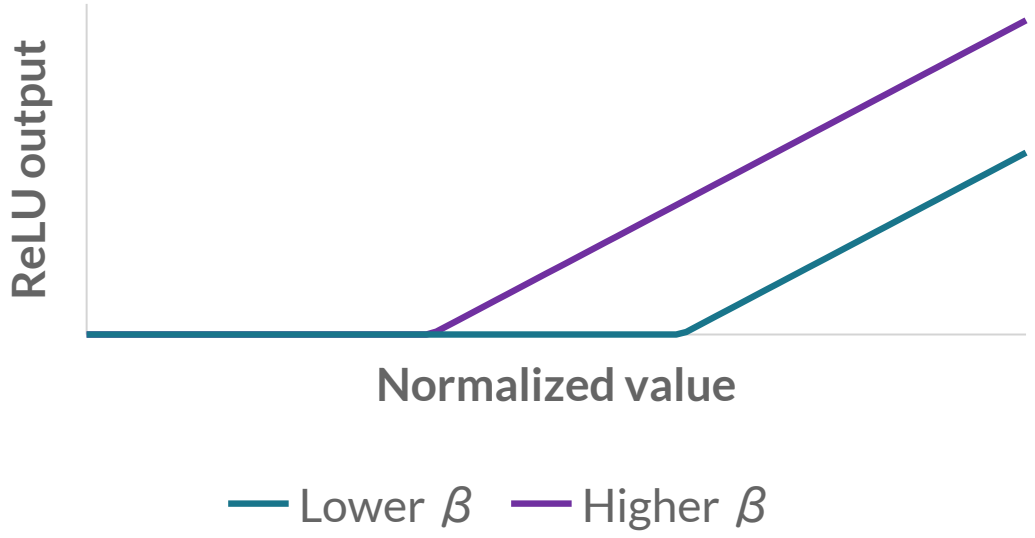
Learn a useful scale and threshold for the following ReLU.

$$u_{ij} = \gamma_j \widehat{z}_{ij} + \beta_j$$

γ_j: learned scale



β_j: learned shift



One learned scale and shift per feature, shared across examples.

Batch normalization

$$u_{ij} = \gamma_j \frac{z_{ij} - \mu_{B,j}}{\sqrt{v_{B,j} + \varepsilon}} + \beta_j$$

For one feature: batch statistics, then learned scale and shift.

Linear → **BatchNorm** → **ReLU**

$$z = Wx + b \qquad u = \mathrm{BN}(z) \qquad h = \mathrm{ReLU}(u)$$

BatchNorm in training and evaluation

Training

$$y_i = \gamma \frac{z_i - \mu_B}{\sqrt{v_B + \epsilon}} + \beta$$

Current minibatch statistics

Update the running estimates

Evaluation

$$y_i = \gamma \frac{z_i - \mu_{\text{run}}}{\sqrt{v_{\text{run}} + \epsilon}} + \beta$$

Stored running estimates

Independent of other examples

$$\gamma, \beta: \text{ same learned scale and shift in both modes}$$

Usual BatchNorm configuration: running statistics are tracked.

Parameters and running statistics

Learned through gradients

$$\gamma_j, \ \beta_j$$

Scale and shift

Updated from observations

$$\mu_{\text{run},j}, \ v_{\text{run},j}$$

Running mean and variance

$$\mu_{\text{run},j} \leftarrow (1-\rho)\mu_{\text{run},j} + \rho\mu_{B,j}$$

ρ sets how much weight the newest batch receives.

BatchNorm at inference

Fixed statistics turn BatchNorm into a scale and a shift.

$$u_j = \gamma_j \frac{z_j - \mu_{\text{run},j}}{\sqrt{v_{\text{run},j} + \varepsilon}} + \beta_j = s_j z_j + t_j$$

$$s_j = \frac{\gamma_j}{\sqrt{v_{\text{run},j} + \varepsilon}}, \qquad t_j = \beta_j - s_j \mu_{\text{run},j}$$

s: fixed scale per feature t: fixed shift per feature

Folding BatchNorm into a linear layer

Linear followed directly by BatchNorm, in evaluation mode

Original computation

$$z = Wx + b, \qquad u = s \odot z + t$$

Fold each output row

$$W'_{j,:} = s_j W_{j,:}, \qquad b'_j = s_j b_j + t_j$$

One linear layer
with bias

$$u = W'x + b'$$

Bias before BatchNorm

A shared bias cancels when the batch mean is subtracted.

$$z_i = u_i + b, \qquad \mu_B(z) = \mu_B(u) + b$$

$$z_i - \mu_B(z) = u_i - \mu_B(u)$$

```
nn.Sequential(
    nn.Linear(input_dim, channels, bias=False),
    nn.BatchNorm1d(channels),  # includes learned shift beta
    nn.ReLU(),
)
```

BatchNorm in an MLP

```
model = nn.Sequential(
    nn.Linear(input_dim, hidden_dim, bias=False),
    nn.BatchNorm1d(hidden_dim),  # one scale and shift per feature
    nn.ReLU(),
    nn.Linear(hidden_dim, num_classes),
)
```

Modes and gradient recording

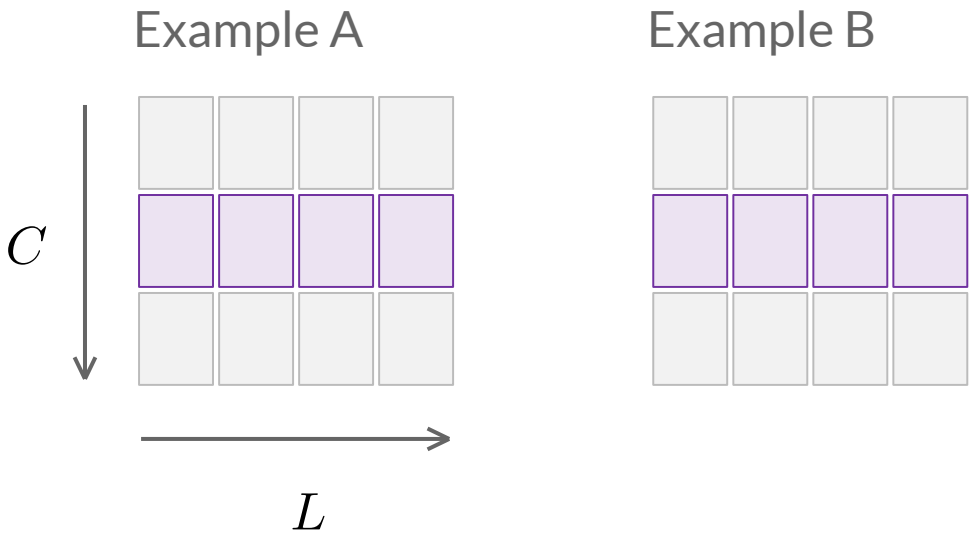
```
model.train()              # use batch statistics
pred = model(x)            # update running estimates

model.eval()               # use stored statistics
with torch.no_grad():      # skip gradient recording
    pred = model(x)        # evaluation forward pass
```

train() / eval()	Select module behavior
no_grad()	Control gradient recording

Examples, channels and positions

Sequence-like input

(N, C, L)

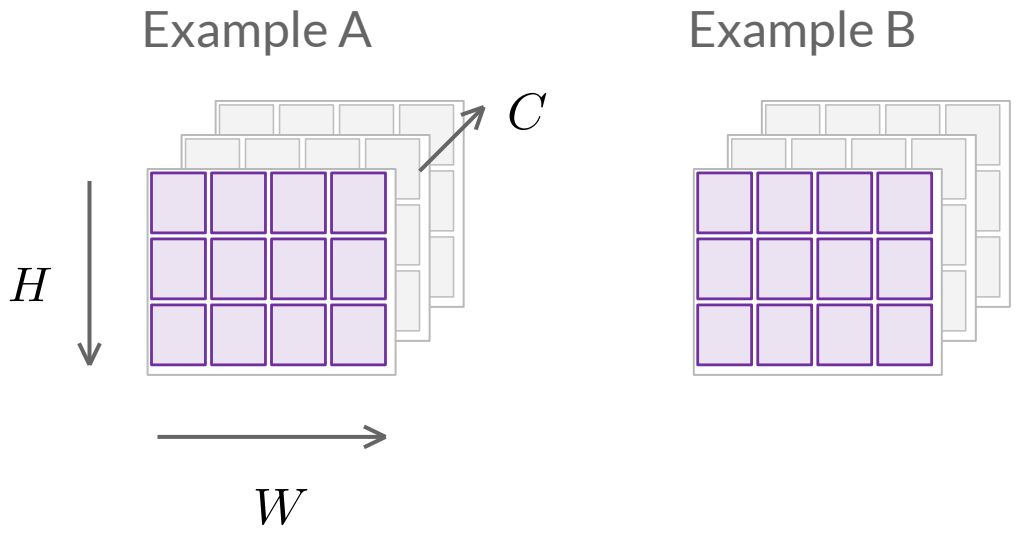


N examples

C channels at each position

Image-like input

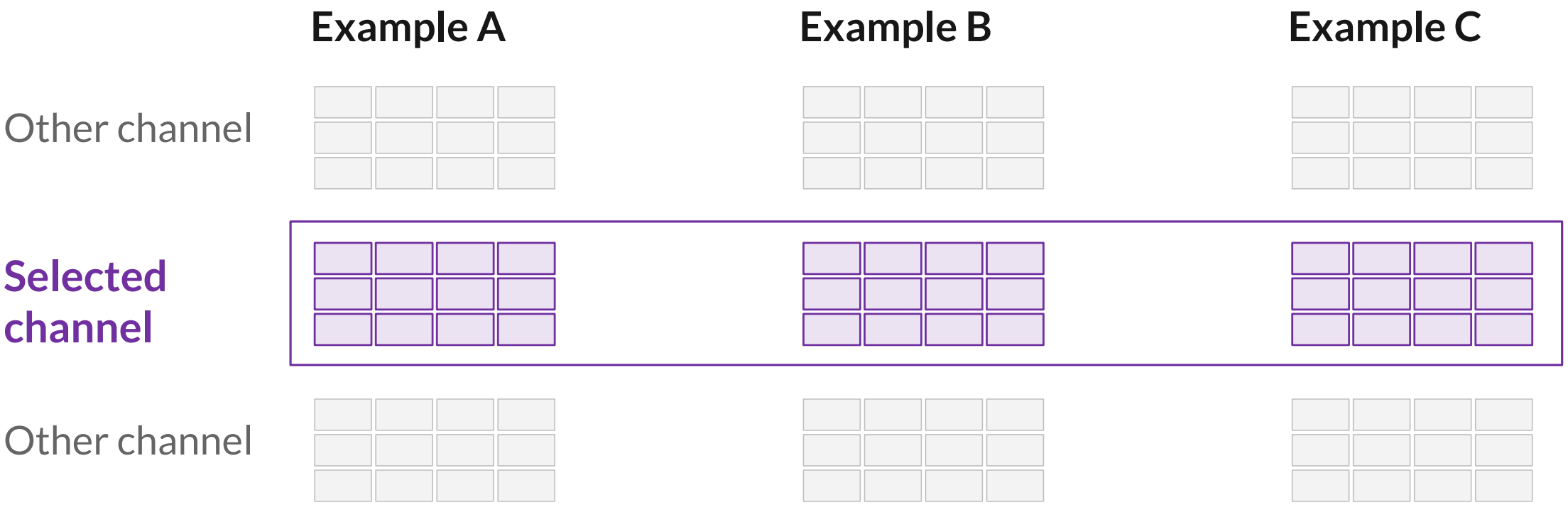
(N, C, H, W)



N examples

C grids, each with H × W positions

BatchNorm across spatial positions



$\text{mean}_{N,H,W}$, $\text{variance}_{N,H,W}$

One learned pair per channel

γ_c, β_c

BatchNorm shapes in PyTorch

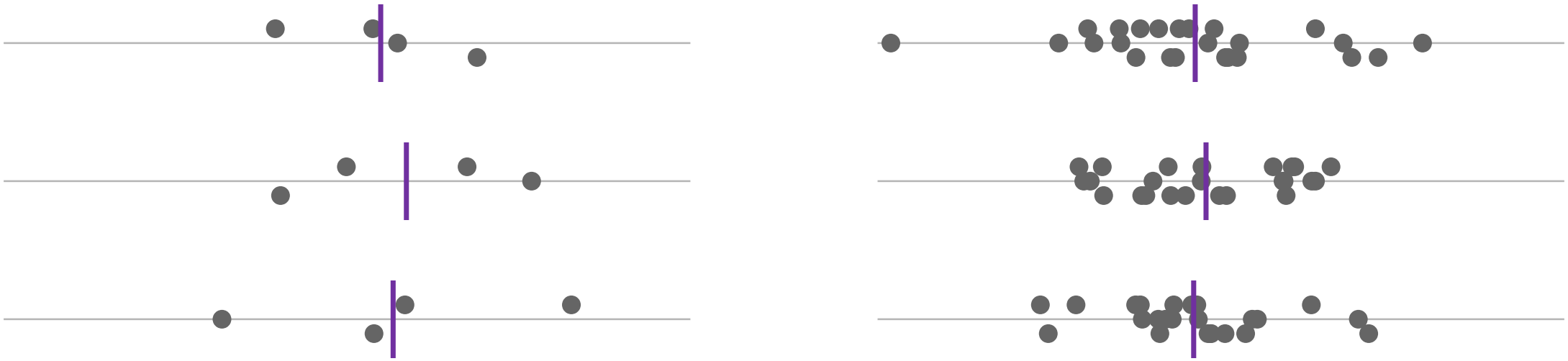
```
bn = nn.BatchNorm1d(C)
# Dense features: [N, C]
# Sequence features: [N, C, L]

bn = nn.BatchNorm2d(C)
# Image-like features: [N, C, H, W]
# One mean and variance over N, H, W per channel
```

C counts channels; N counts examples.

Batch size and BatchNorm

Smaller batches

Larger batches



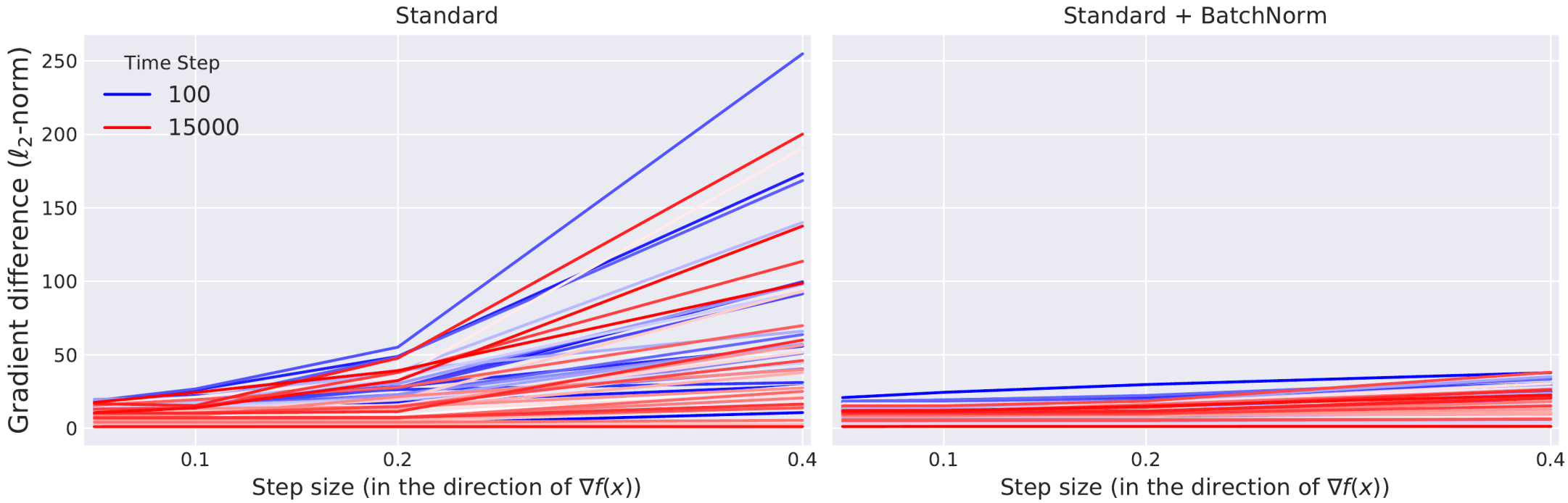
Batch mean

Repeated draws from the same distribution

With fewer values, the estimated mean fluctuates more.

BatchNorm and gradient stability

Measure how much the gradient changes after a trial step.



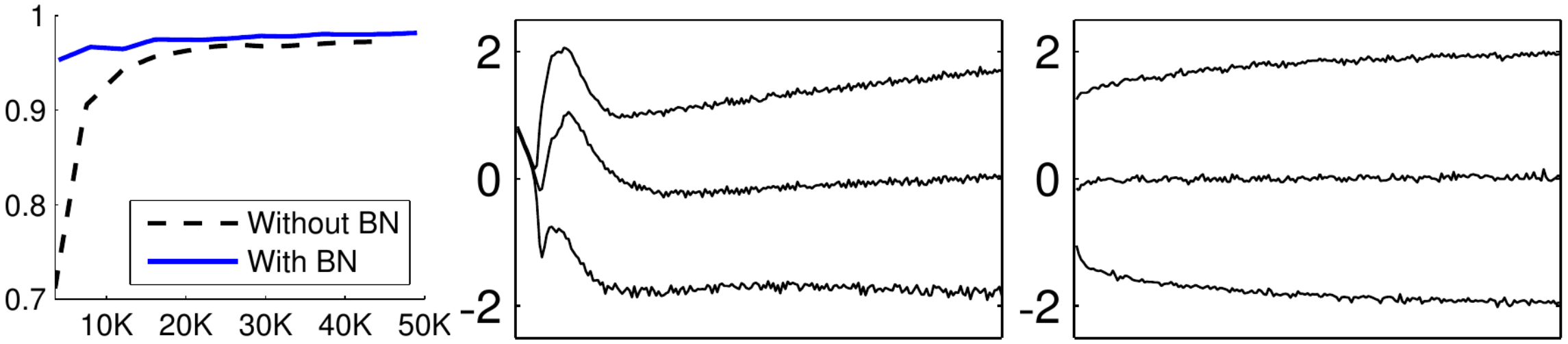
With BatchNorm, gradients change less in this comparison.

Source: How Does Batch Normalization Help Optimization?

BatchNorm in a digit classifier

Test accuracy and sigmoid-input percentiles during training.

Test accuracy **Without BN** **With BN**



Training steps

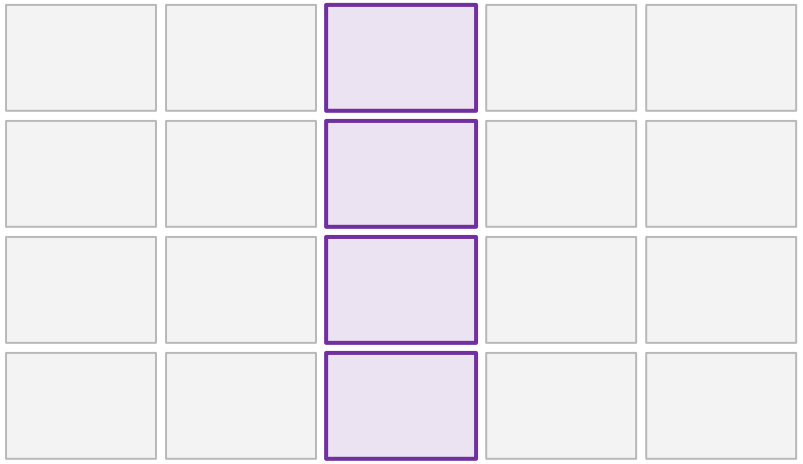
Input percentiles over training

Source: Batch Normalization

Normalization within each example

Use each example's own features when batch statistics are unreliable.

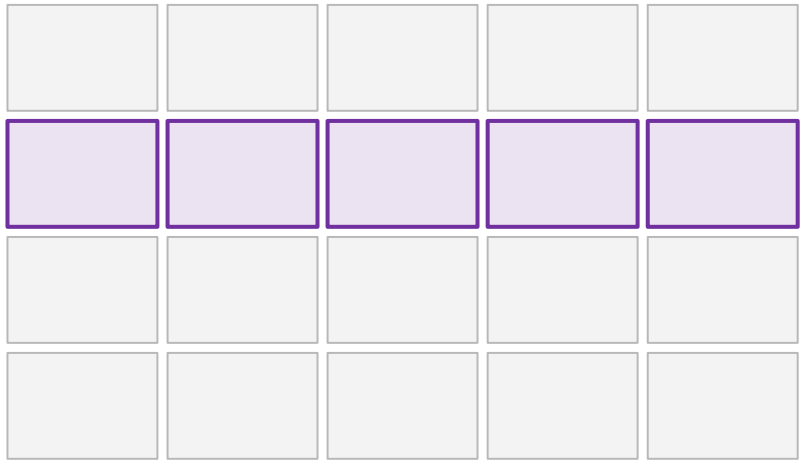
BatchNorm



Use the current batch

Rows: examples

LayerNorm

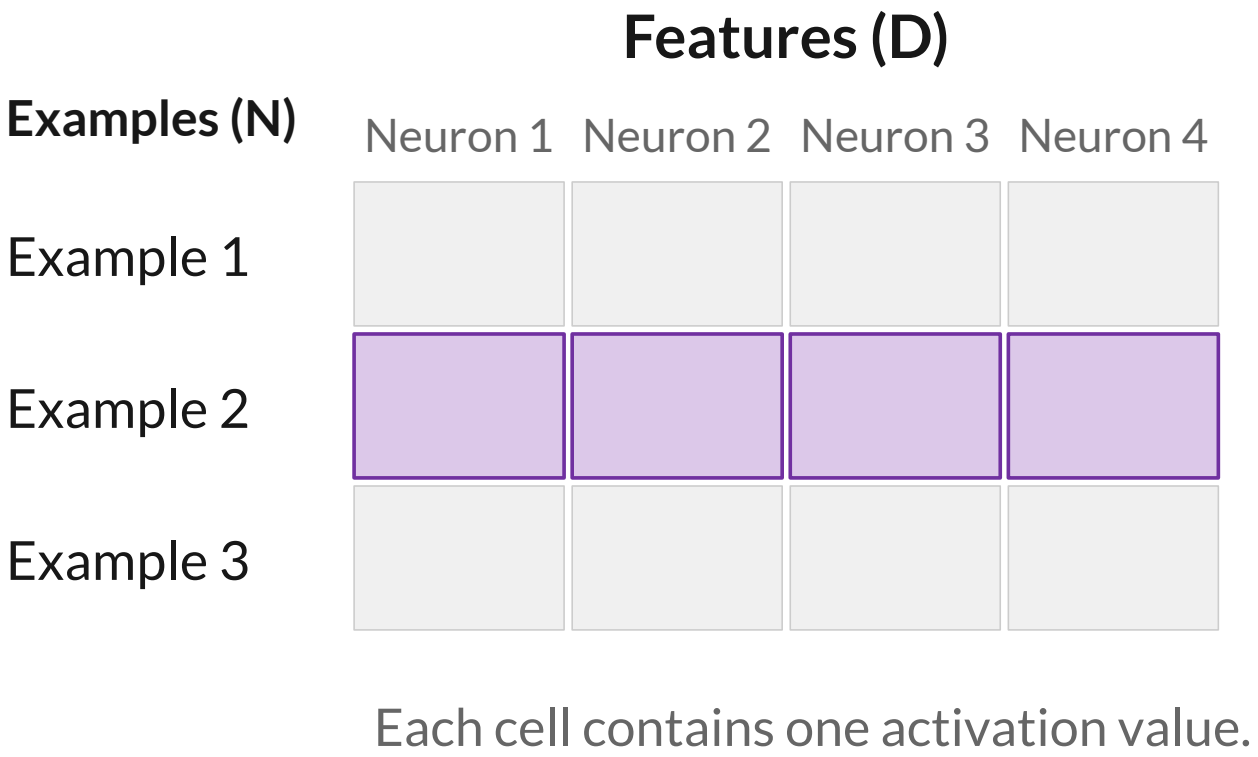


Use this example's features

Columns: features

LayerNorm groups

Dense-layer activations: one feature vector for each example.



All features of one example

These highlighted values share one mean and variance.

Repeat for every example.

LayerNorm computation

One example with D features in the normalization group

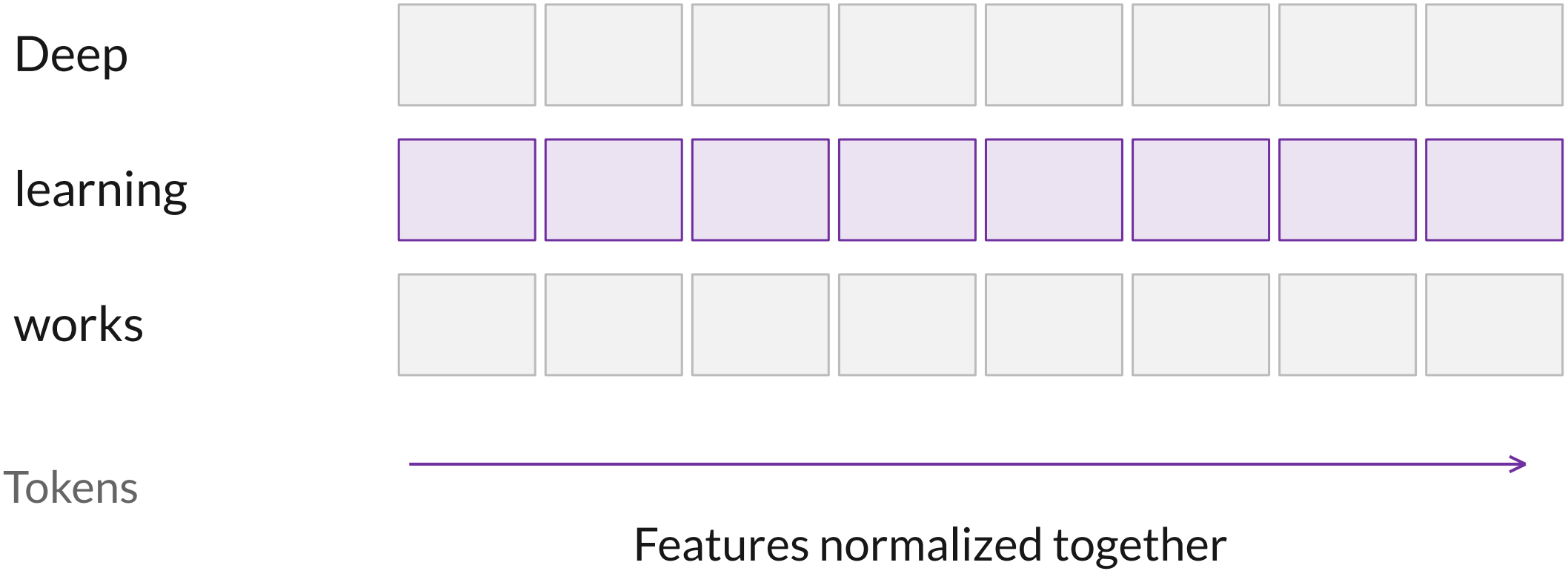
$$\mu = \frac{1}{D}\sum_{j=1}^{D} x_j, \qquad v = \frac{1}{D}\sum_{j=1}^{D}(x_j - \mu)^2$$

$$y_j = \gamma_j \frac{x_j - \mu}{\sqrt{v + \varepsilon}} + \beta_j$$

Same centering and scaling, using statistics from one example.

LayerNorm on token features

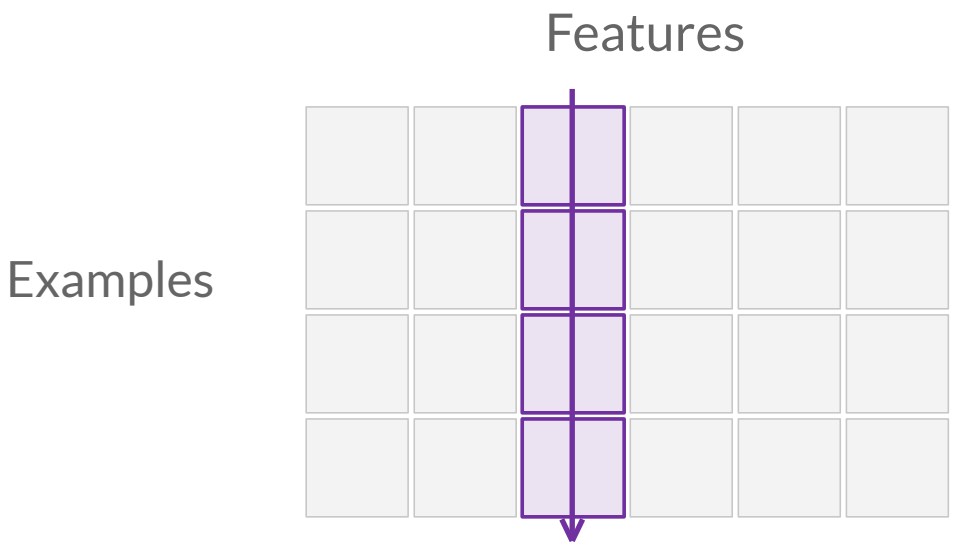
One example



Each token (one sequence item) uses its own statistics.

BatchNorm and LayerNorm

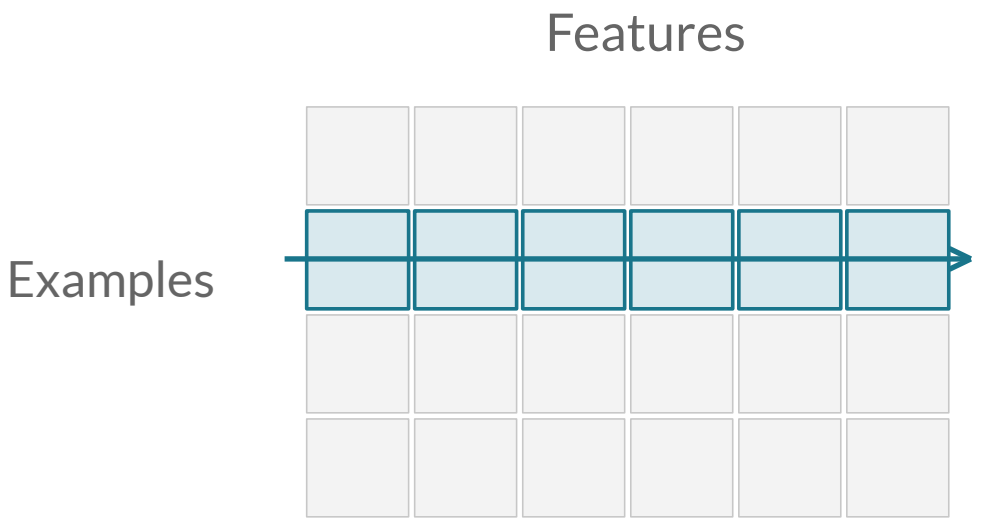
BatchNorm



One feature across examples

Training: current batch statistics
Evaluation: running estimates

LayerNorm



Features within one example

Current vector statistics
in both modes

RMSNorm

Root mean square

$$r = \sqrt{\frac{1}{D}\sum_{j=1}^{D} x_j^2 + \varepsilon}$$

Scale each feature

$$y_j = \gamma_j \frac{x_j}{r}$$

D: features in this vector γ: learned featurewise scale

Less computation

Omit mean subtraction.
Keep control of magnitude.

Language models

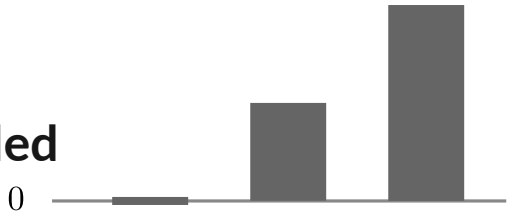
Can retain similar quality.
Gains depend on the model, implementation and hardware.

Centering and RMS scaling

Input

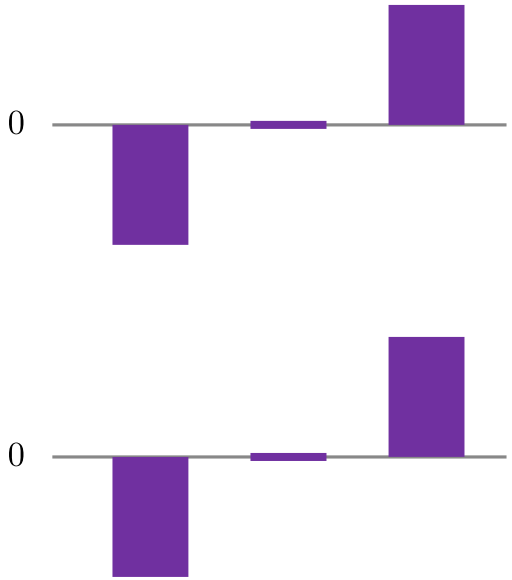
Original

Shared offset added



Constructed vectors

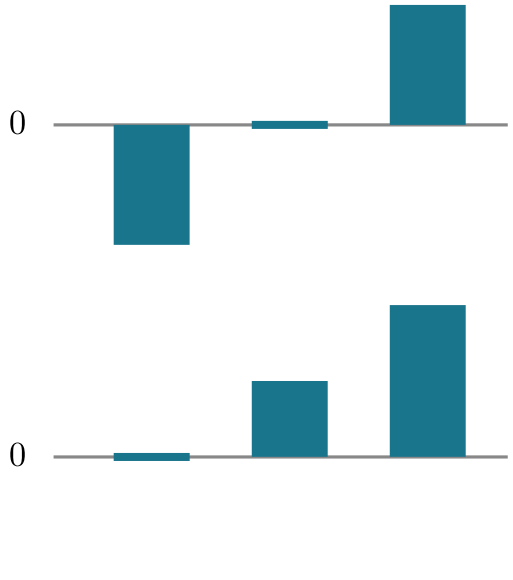
LayerNorm



Same output

$\gamma = 1, \quad \beta = 0;$

RMSNorm



Different outputs

ϵ omitted

LayerNorm and RMSNorm in PyTorch

```
# One feature vector at each sequence position
# x has shape [N, L, D]

ln = nn.LayerNorm(D)
rms = nn.RMSNorm(D)

y_ln = ln(x)    # normalize the last axis
y_rms = rms(x)  # same group; omit centering
```

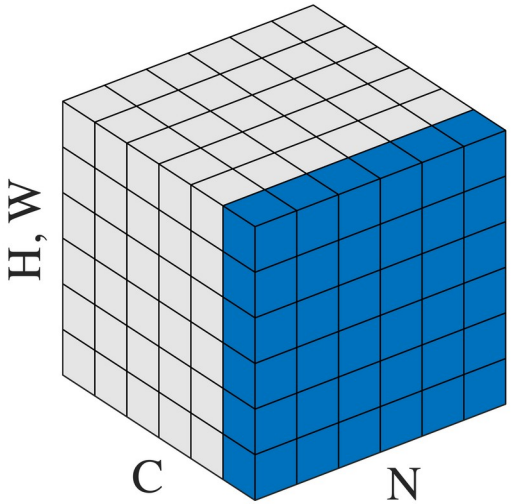
N: examples L: sequence positions D: features

Normalization groups on spatial tensors

A channel is one feature measured at each position.

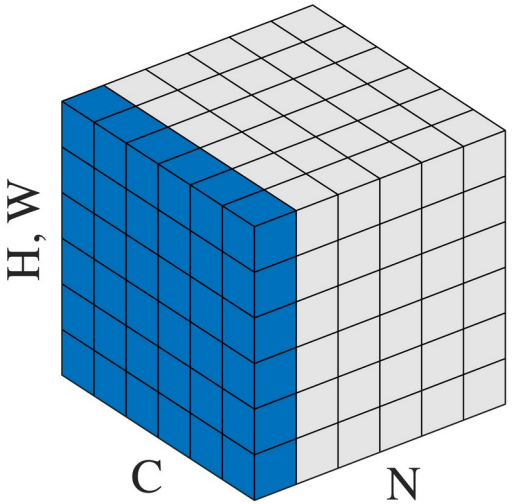
N: examples in the batch C: feature channels H, W: positions

BatchNorm



One channel across examples and positions.

LayerNorm (shown here)



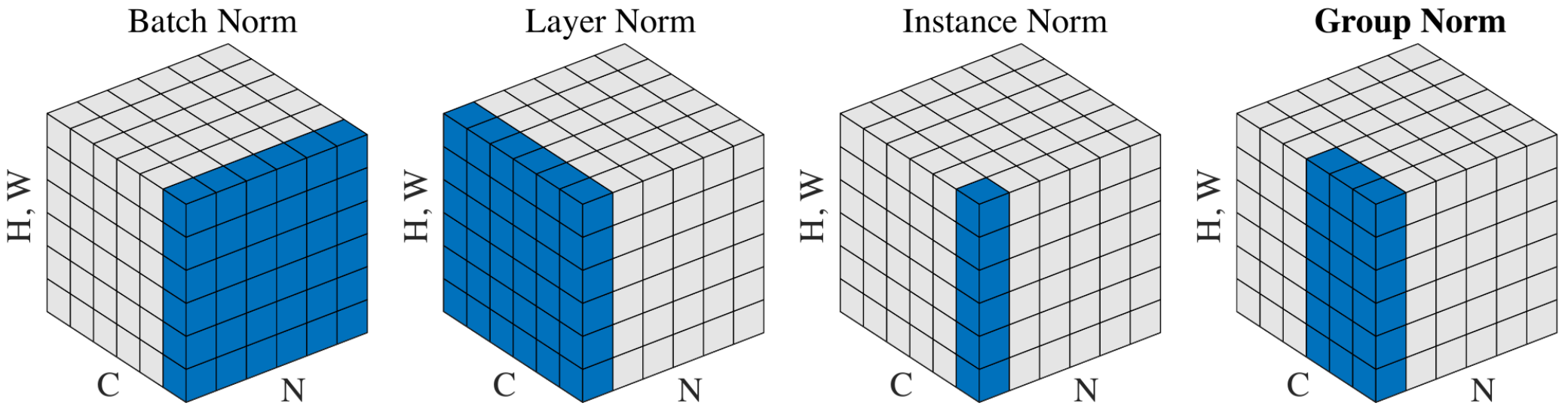
One example across channels and positions.

Source: Group Normalization

Normalization groups

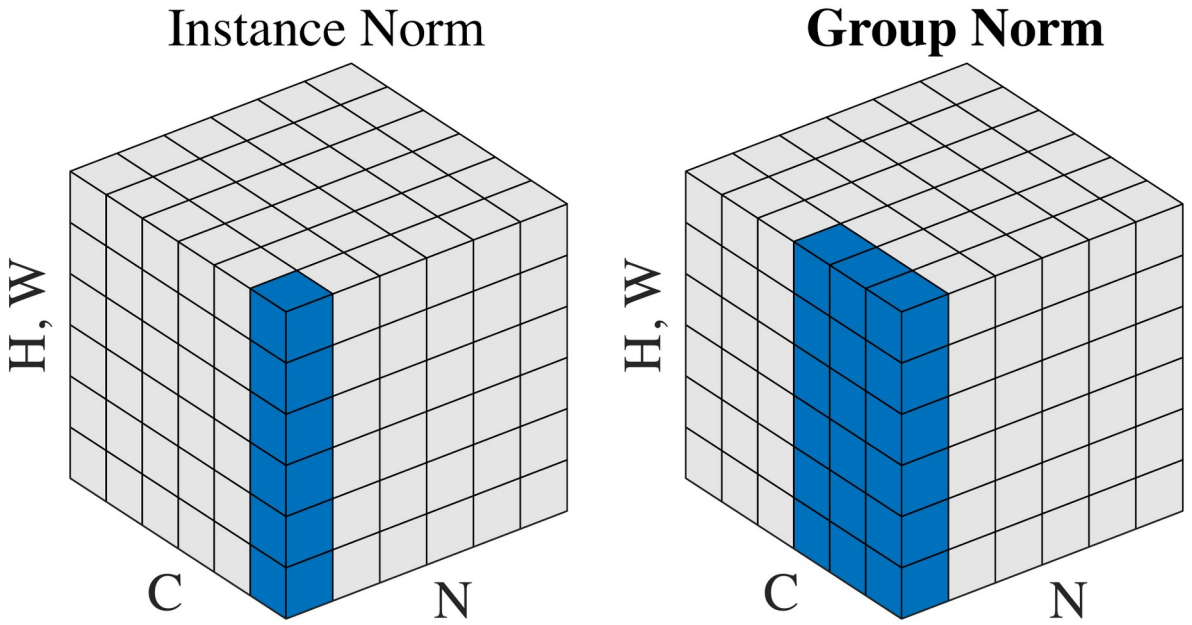
Blue cells share one mean and variance.

N: examples C: channels H, W: positions within an example



Source: Group Normalization

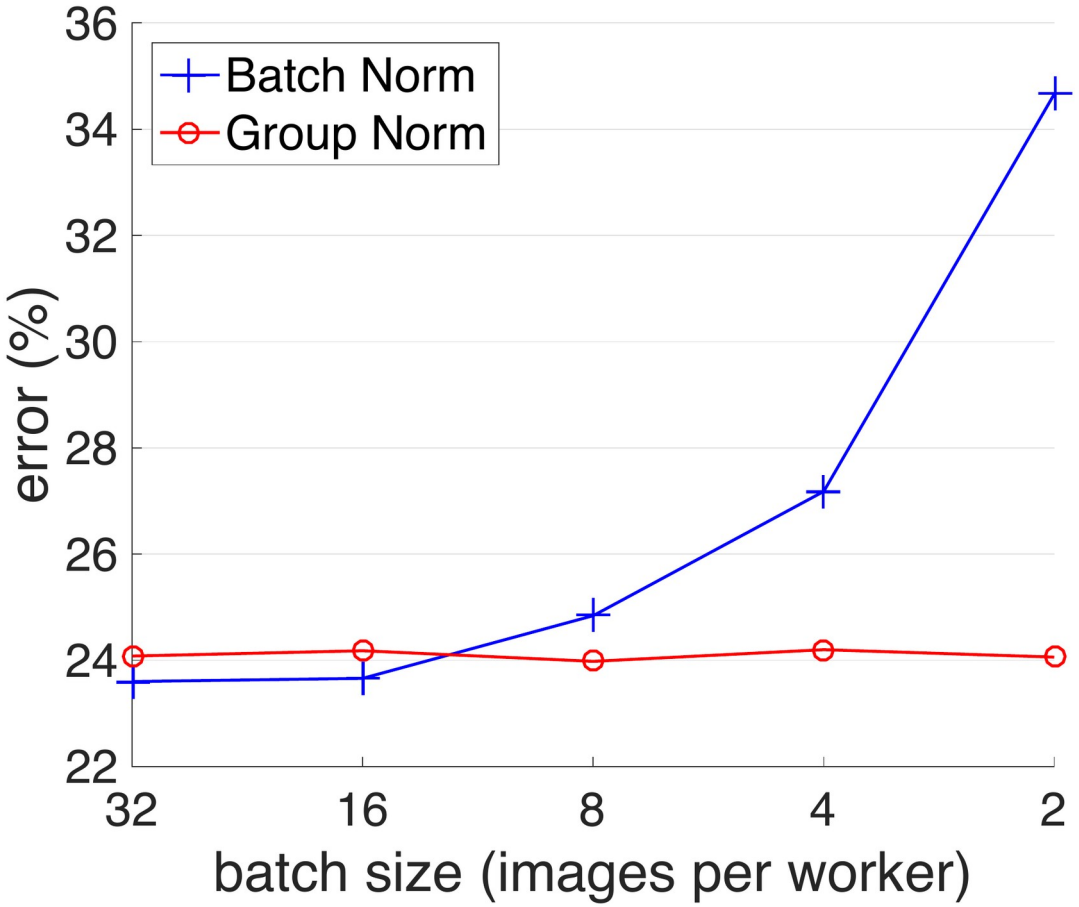
Instance and group normalization



- InstanceNorm: one channel of one example; removes its mean and contrast scale.
- GroupNorm: pool several channels within an example.

Source: Group Normalization

Normalization with small batches

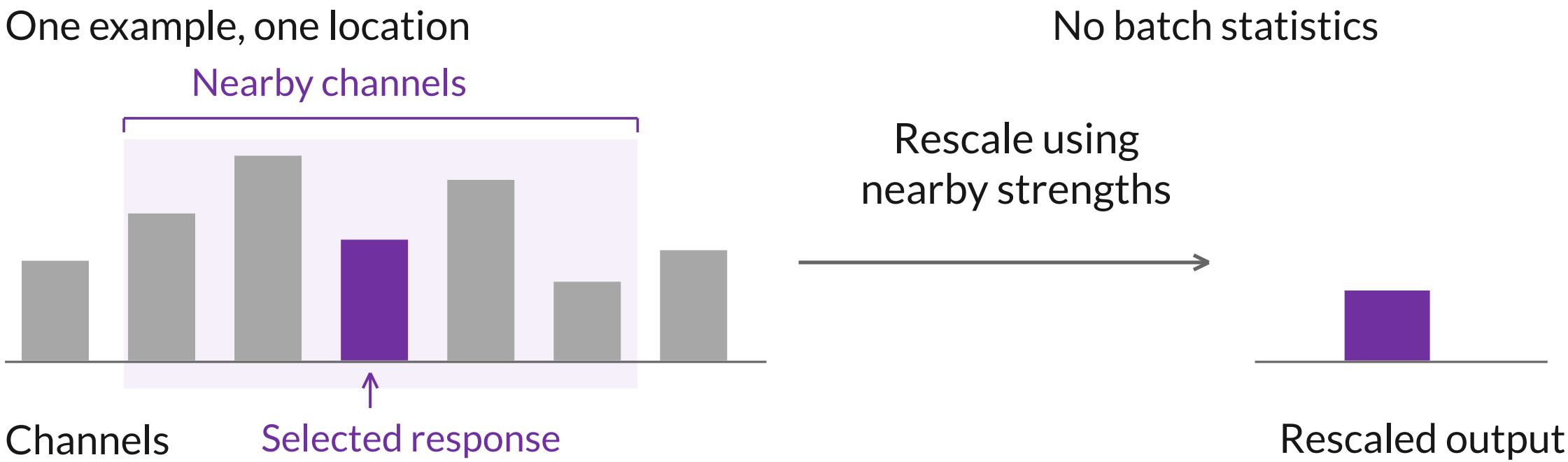


- Image-classification validation error.
  A worker is one training device.
- GroupNorm stays stable as examples per worker decrease.

Source: Group Normalization

Local response normalization (LRN)

Used in AlexNet (2012), an early image-recognition network.



Now mainly historical and rarely used in modern architectures.

Lecture outline

01 Initialization

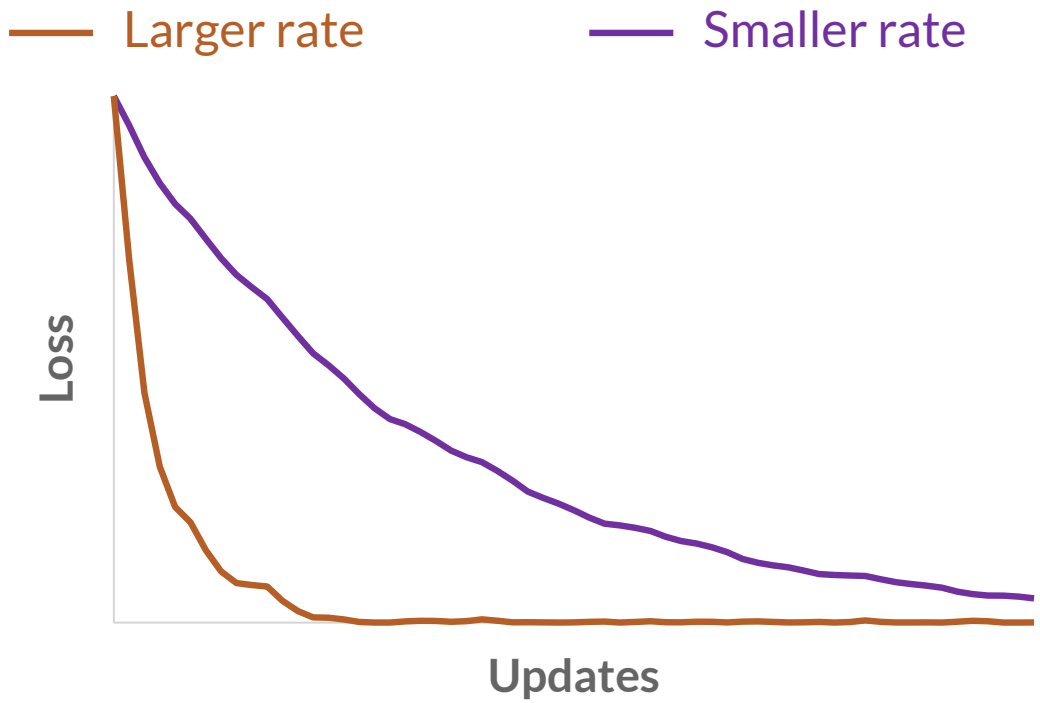
02 Normalization

03 Schedules and batches

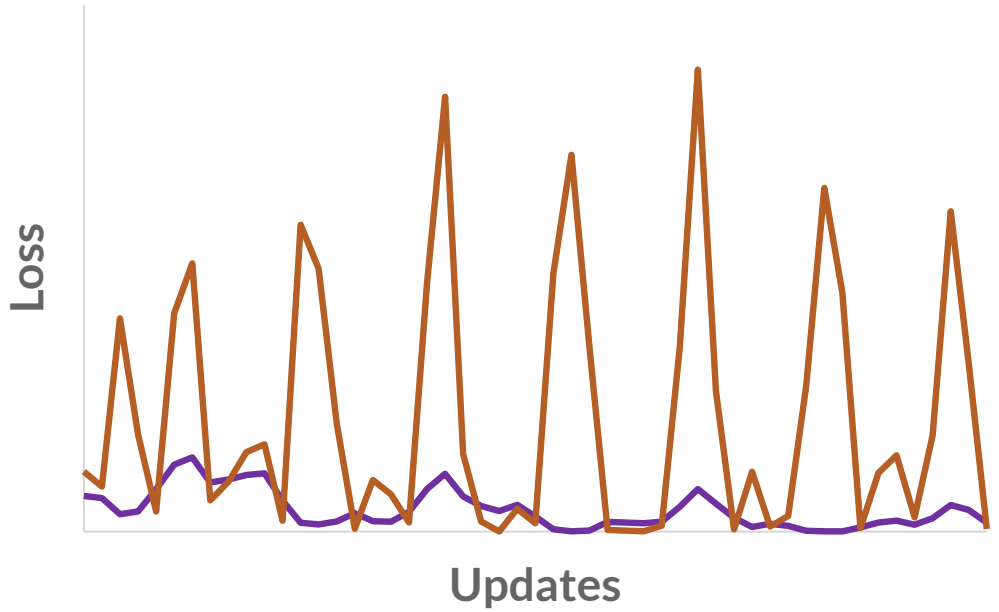
04 Clipping and stability

Learning rate during training

Early progress



Later updates: expanded scale

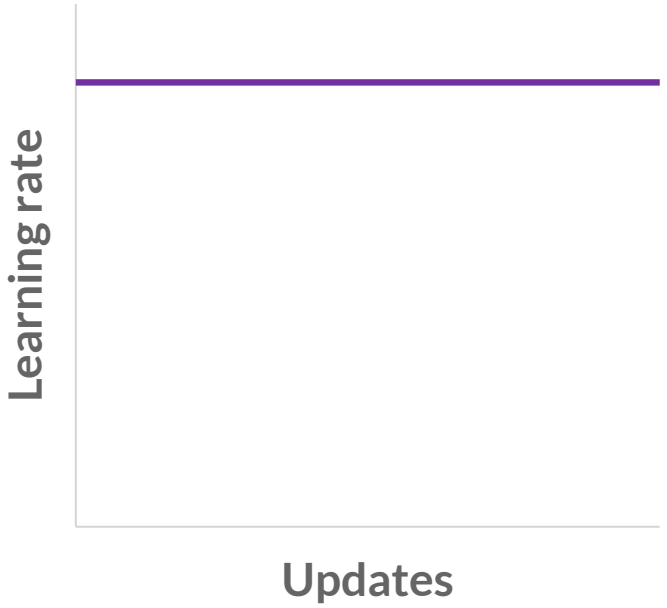


$$L(\theta) = \tfrac{1}{2}\theta^2$$

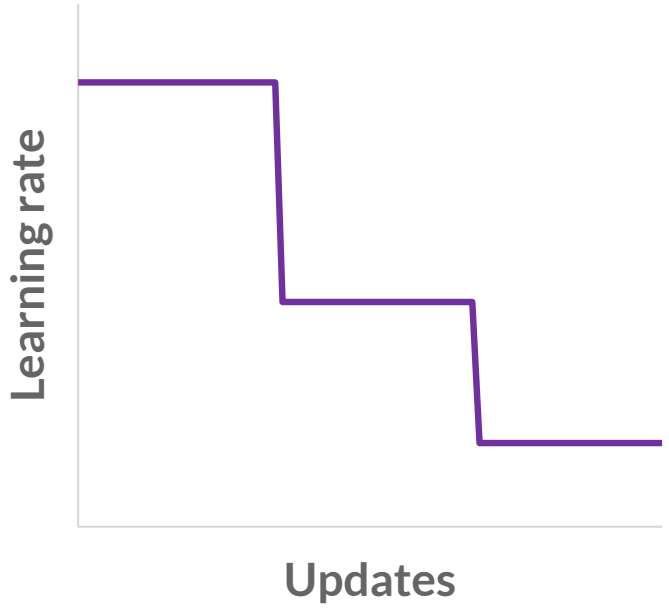
Constructed noisy-gradient example; the two runs use the same perturbations.

Learning-rate schedules

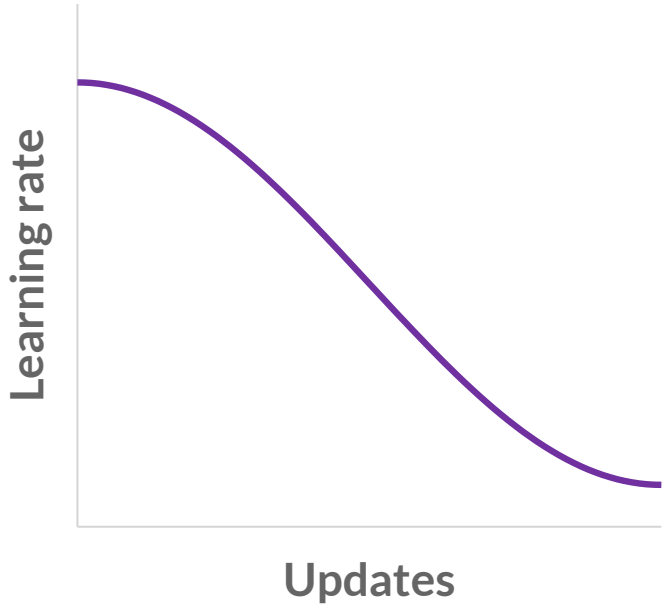
Constant



Step decay

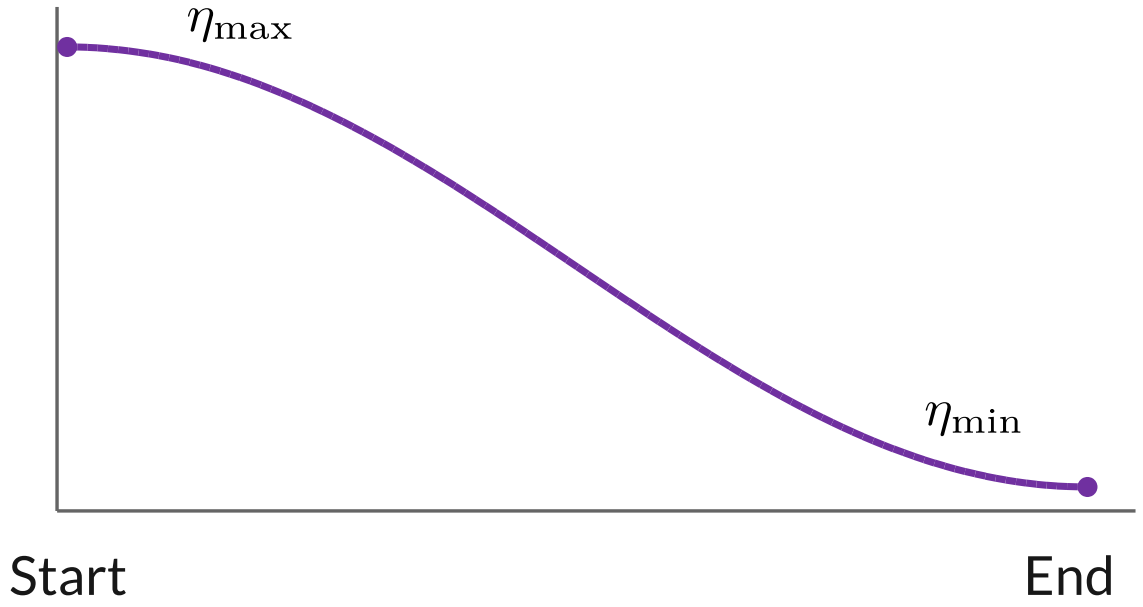


Cosine decay



Cosine decay

Learning rate



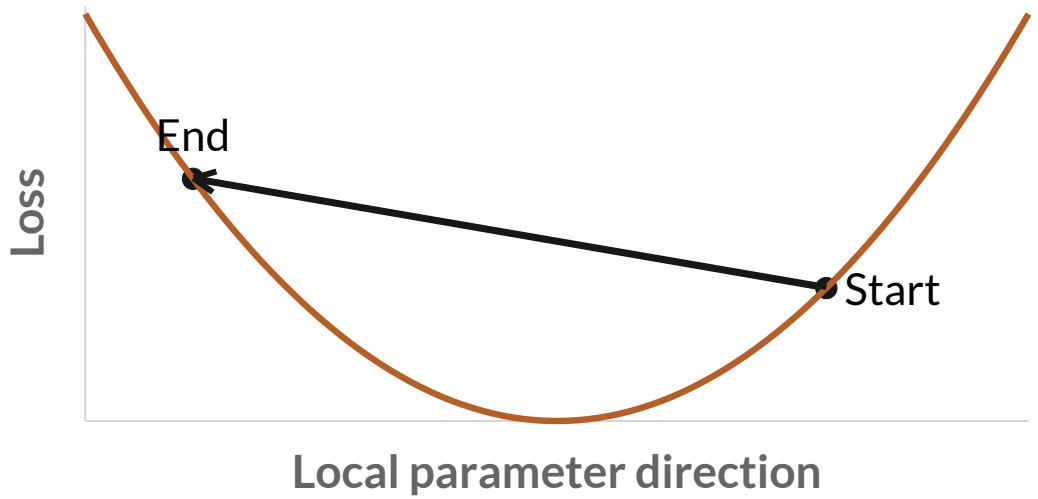
η_t current learning rate

T total decay updates

$$\eta_t = \eta_{\min} + \frac{\eta_{\max} - \eta_{\min}}{2}\left(1 + \cos\frac{\pi t}{T}\right)$$

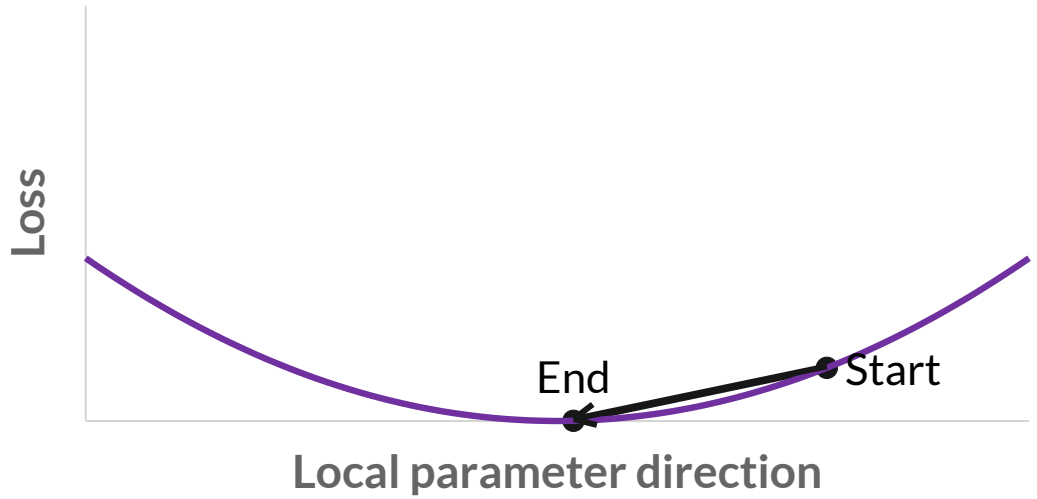
Warmup and local curvature

Sharper region



The step overshoots and raises the loss.

Flatter region



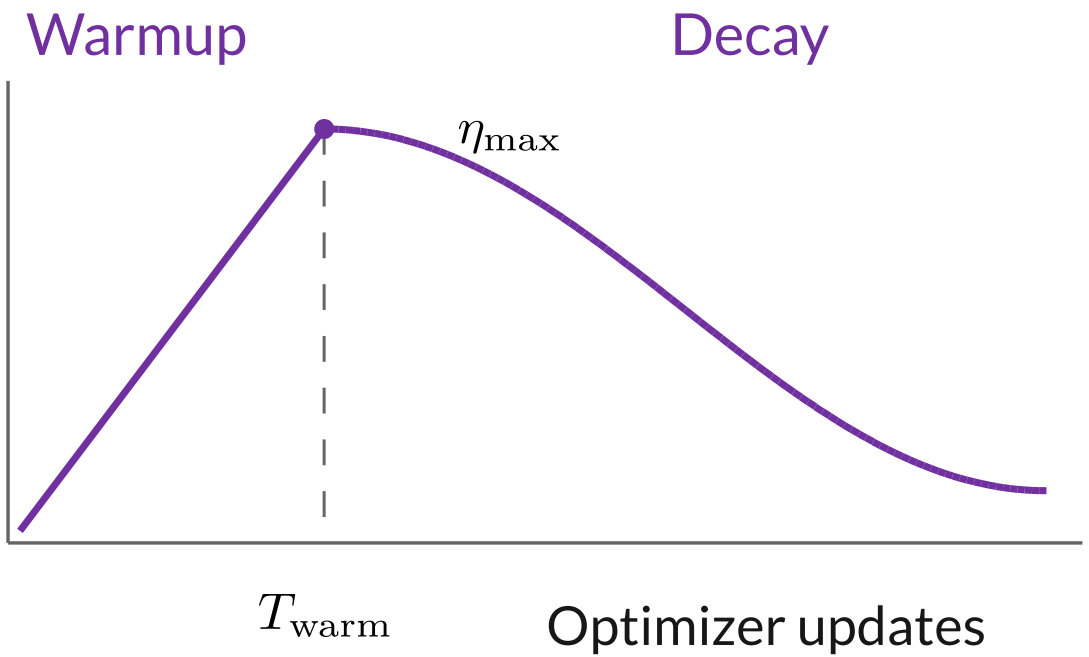
The step lowers the local loss.

Same learning rate in both regions

Small early updates can reach regions that tolerate a larger learning rate.

Warmup followed by decay

Learning rate



$\eta_{\max}$ target rate

T_{warm} warmup duration (in updates)

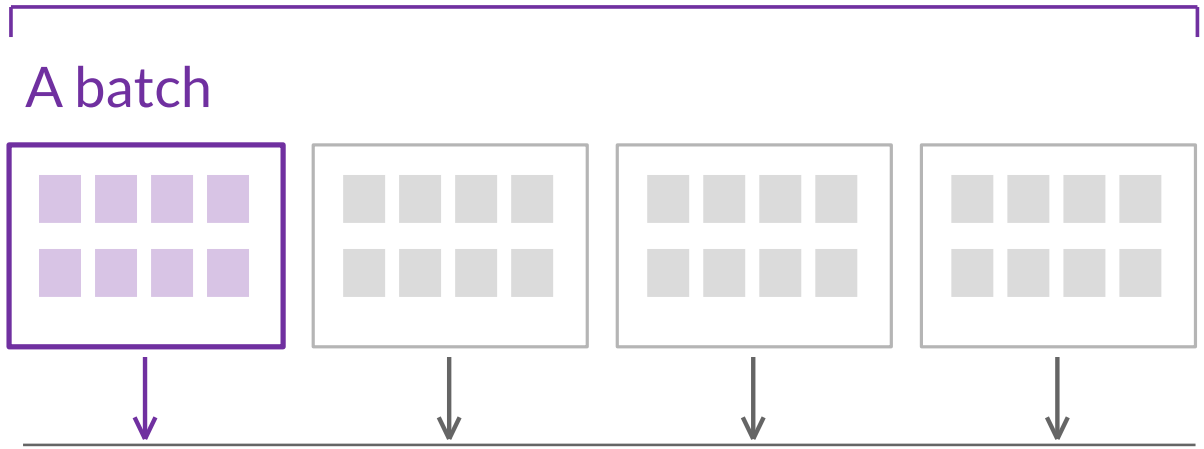
$$\eta_t = \eta_{\max} \frac{t}{T_{\text{warm}}}, \qquad 0 \leq t \leq T_{\text{warm}}$$

Learning-rate scheduling in PyTorch

```
# Cosine decay only; rate starts in the optimizer
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(
    optimizer, T_max=total_updates, eta_min=minimum_lr
)

for inputs, targets in loader:
    optimizer.zero_grad()
    loss = criterion(model(inputs), targets)
    loss.backward()
    optimizer.step()  # update parameters
    scheduler.step()  # advance the schedule
```

Batches, updates and epochs

One epoch: one pass through the training set



N number of training examples

B examples used for one update

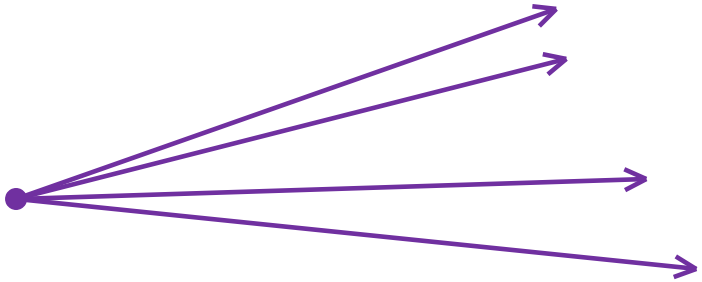
$$\text{updates per epoch} \approx \frac{N}{B}$$

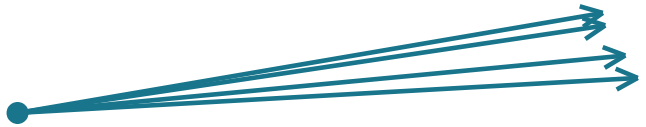
Choosing a batch size

Smaller batch

Larger batch

Gradient estimates from different batches, at fixed parameters





More variable estimates

- Less activation memory
- More updates per epoch

Less variable estimates

- More activation memory
- Fewer updates per epoch
- Can improve GPU use

Linear learning-rate scaling for SGD

SGD heuristic when gradients change slowly

Several smaller batches

Several updates

One combined batch

One update

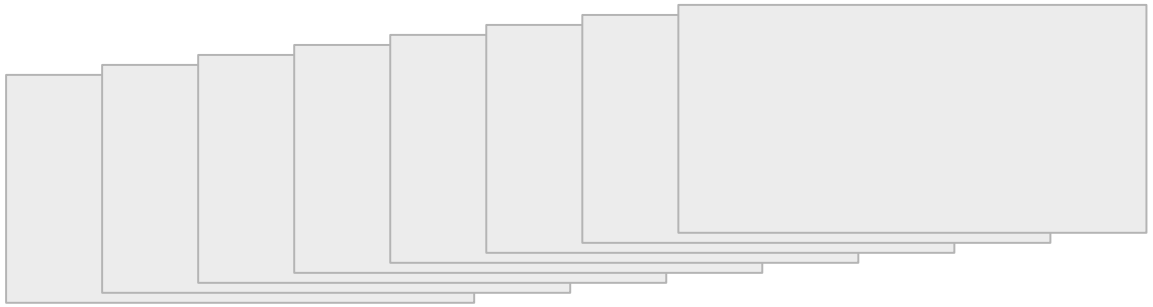
$$B \longrightarrow kB, \qquad \eta \longrightarrow k\eta$$

k batch-size multiplier

Curvature still limits the step size.

An effective batch under a memory limit

All examples at once



Many activations kept in memory

Microbatches in sequence

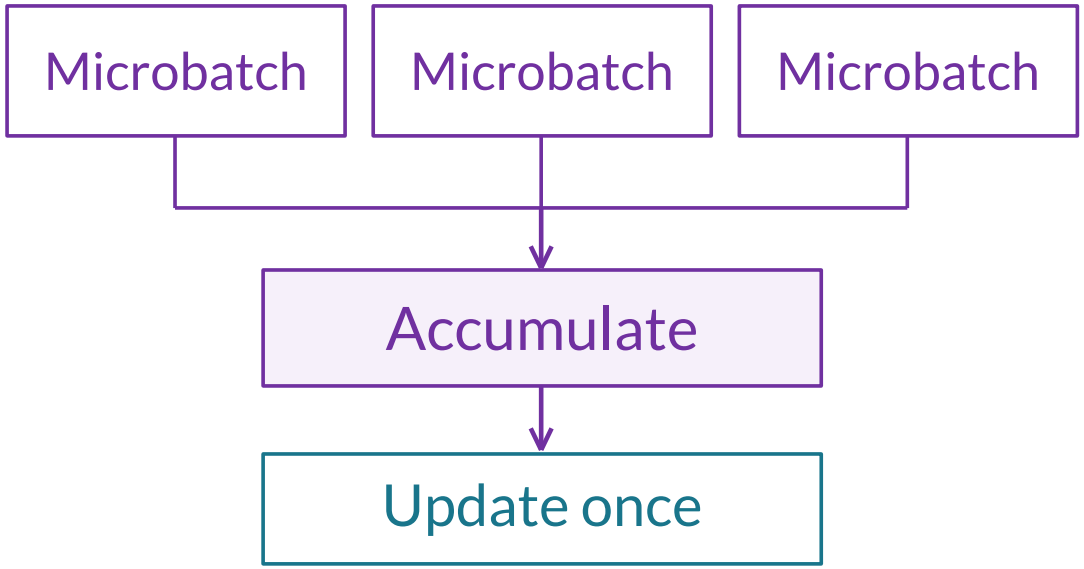
Keep only one microbatch's activations at a time

Accumulated gradients remain until the parameter update.

Gradient accumulation

Equal-sized microbatches; the parameters stay fixed

$$g = \nabla_\theta \left[\frac{1}{K} \sum_{k=1}^{K} L_k(\theta) \right] = \frac{1}{K} \sum_{k=1}^{K} \nabla_\theta L_k(\theta)$$



K number of microbatches

L_k mean loss of microbatch k

Accumulation and separate updates

Accumulation: fixed parameters

$$g_1(\theta), \quad g_2(\theta), \quad \ldots, \quad g_K(\theta) \longrightarrow \text{One update}$$

Separate updates: parameters change

$$g_1(\theta_0) \longrightarrow \theta_1 \longrightarrow g_2(\theta_1) \longrightarrow \theta_2$$

Gradient accumulation in PyTorch

```
# Equal-sized microbatches; criterion returns a mean
for microbatches in grouped_loader:
    optimizer.zero_grad()
    for inputs, targets in microbatches:
        loss = criterion(model(inputs), targets) / K
        loss.backward()  # add to the gradient buffers
    optimizer.step()     # one effective-batch update
    scheduler.step()     # advance once per update
```

K microbatches form one effective batch.

Accumulation with BatchNorm

Microbatch 1

Its own BN statistics

Microbatch 2

Its own BN statistics

Microbatch K

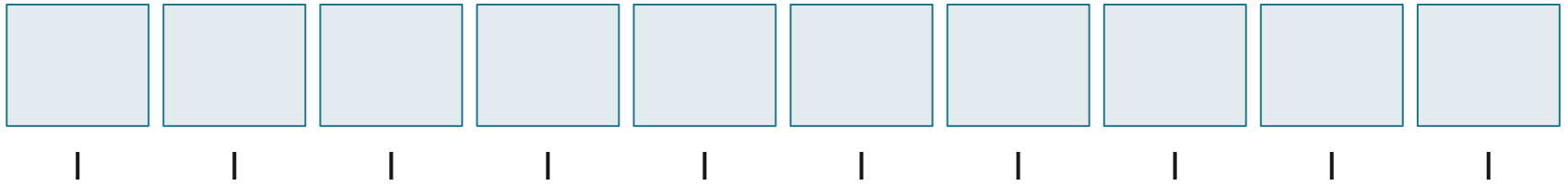
Its own BN statistics

Accumulation does not reproduce the normalization of one large batch.

The schedule's clock

Same data pass

Smaller batch



Larger batch

Each tick marks a parameter update.

Choose whether the schedule follows updates or data passes.

Lecture outline

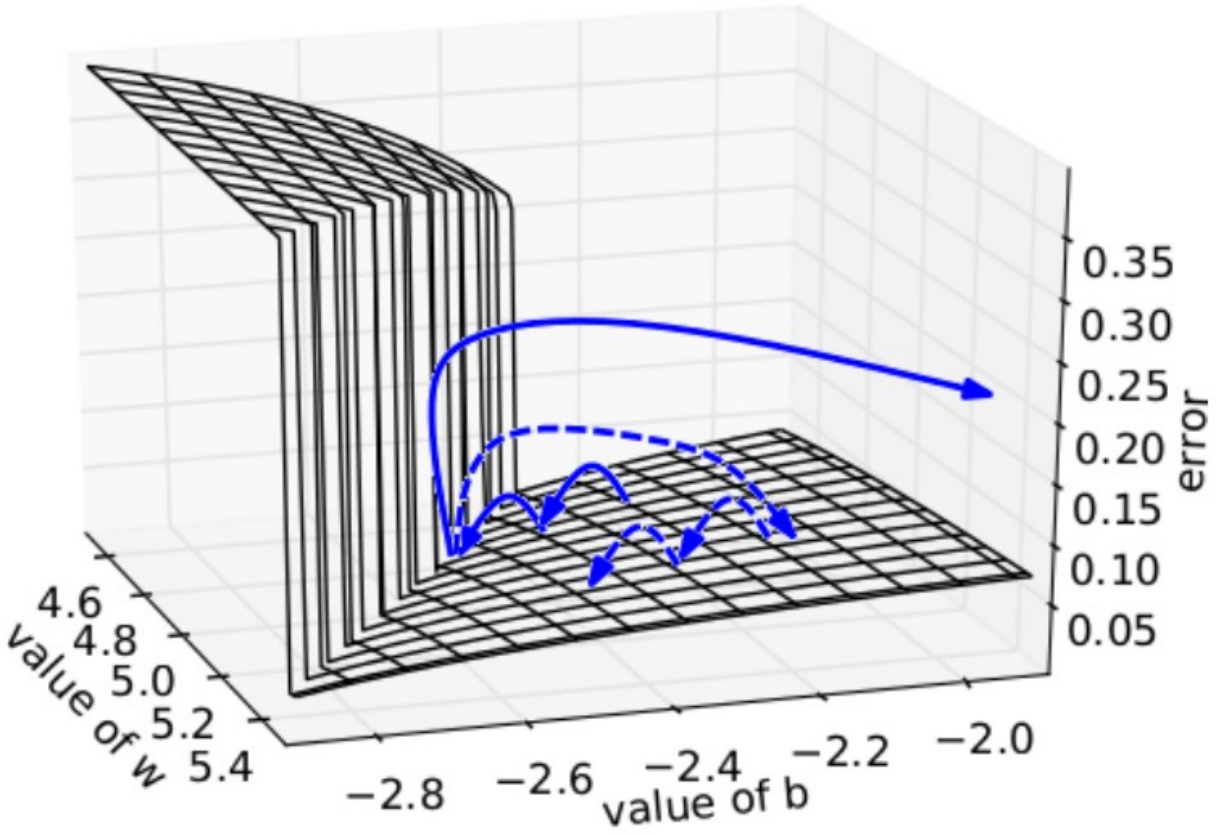
01 Initialization

02 Normalization

03 Schedules and batches

04 Clipping and stability

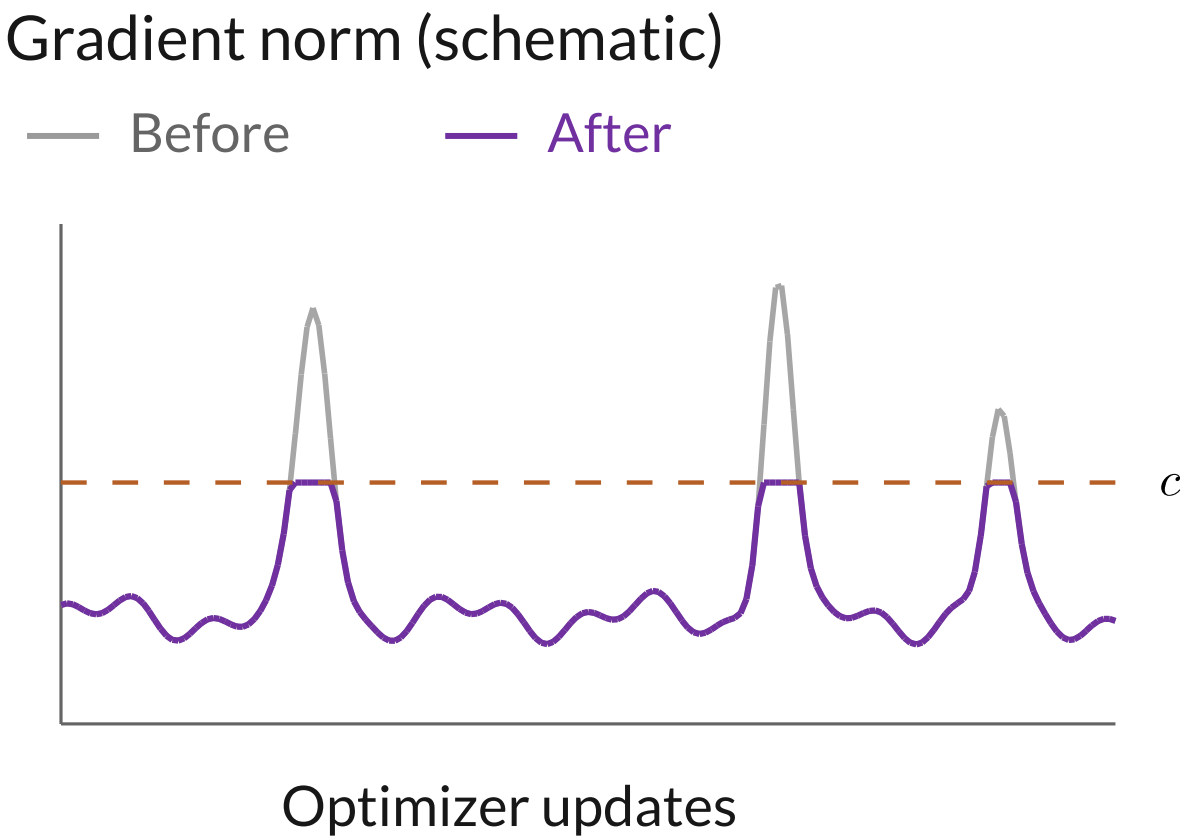
Steep cliffs in the loss surface



- A steep wall produces a large gradient.
- Large steps can erase earlier progress.
- Clipping limits an oversized gradient.

Figure: Pascanu et al. (2013)

Global gradient-norm clipping



$$\widetilde{g} = g \min\left(1, \frac{c}{\|g\|_2}\right)$$

All parameter gradients form one vector.

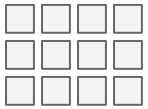
$\|g\|_2$ length of this vector

c clipping threshold

One scale factor for all gradients

Every gradient entry contributes to the global norm.

Stored gradients

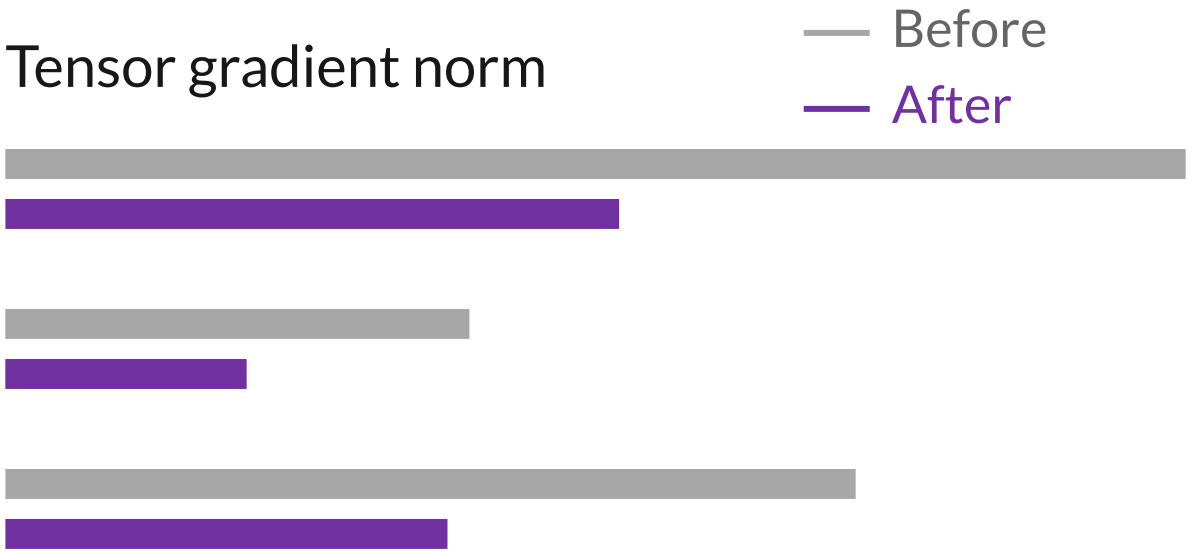


Weight matrix

Bias vector

Next weight matrix

Tensor gradient norm



$$\widetilde{g} = s\, g$$

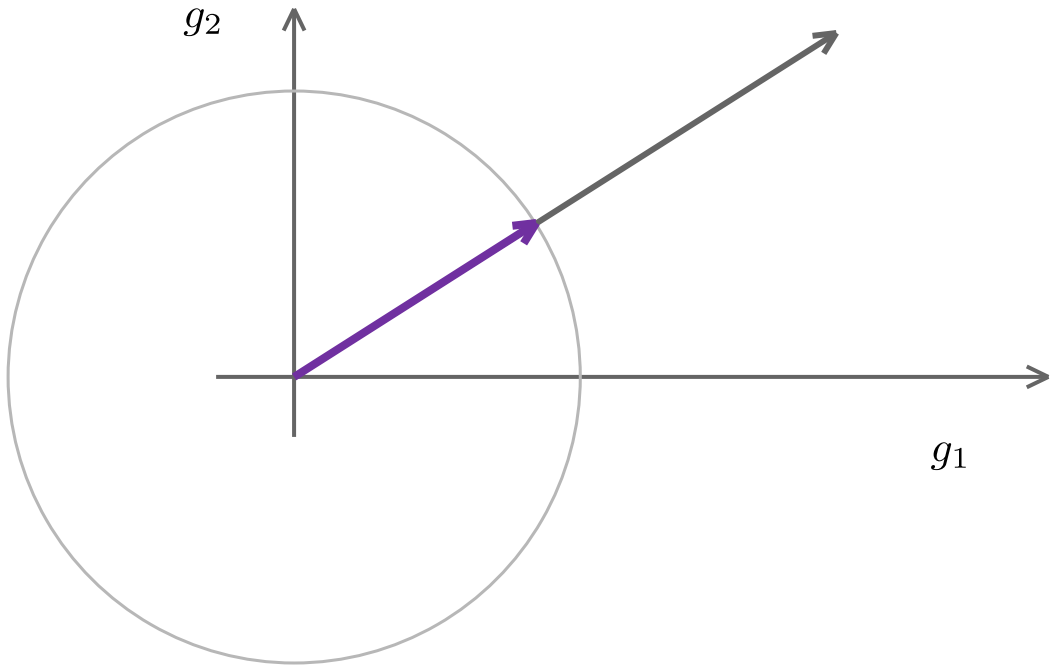
s one common scale factor

Norm clipping and value clipping

— Original gradient

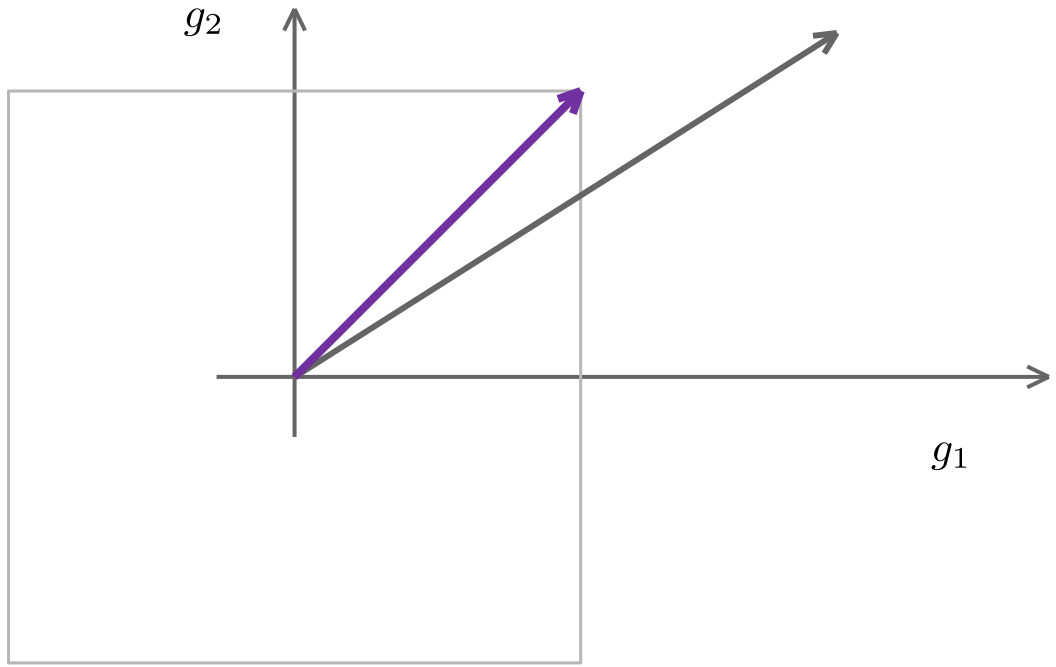
— Clipped gradient

Norm clipping



One common scale factor.

Value clipping



Components are capped separately.

Clipping in the training loop

```
# After all microbatch backward passes
torch.nn.utils.clip_grad_norm_(
    model.parameters(), max_norm
)
optimizer.step()  # use the completed, clipped gradient
scheduler.step()  # advance once per update
```

max_norm is the clipping threshold.

The training procedure

1	Initialize the parameters	**Initialization**
2	For each effective batch:	**Batch**
3	Clear gradients	
4	Forward and backward on each microbatch	**Normalization**
5	Clip the accumulated gradient if needed	**Clipping**
6	Update parameters with the optimizer	
7	Advance the learning-rate schedule	**Schedule**

Next: generalization and training practice

- Training fit and generalization
- Regularization and teacher supervision
- Diagnosing experiments
- Memory and activation checkpointing