DDA4220 · Fall 2026

Lecture 2

Deep Learning and Applications

Neural networks: representation and learning

Zhen Liu

CUHK-Shenzhen

Lecture agenda

01 Neurons, layers and networks

02 Capacity and activations

03 Predictions and losses

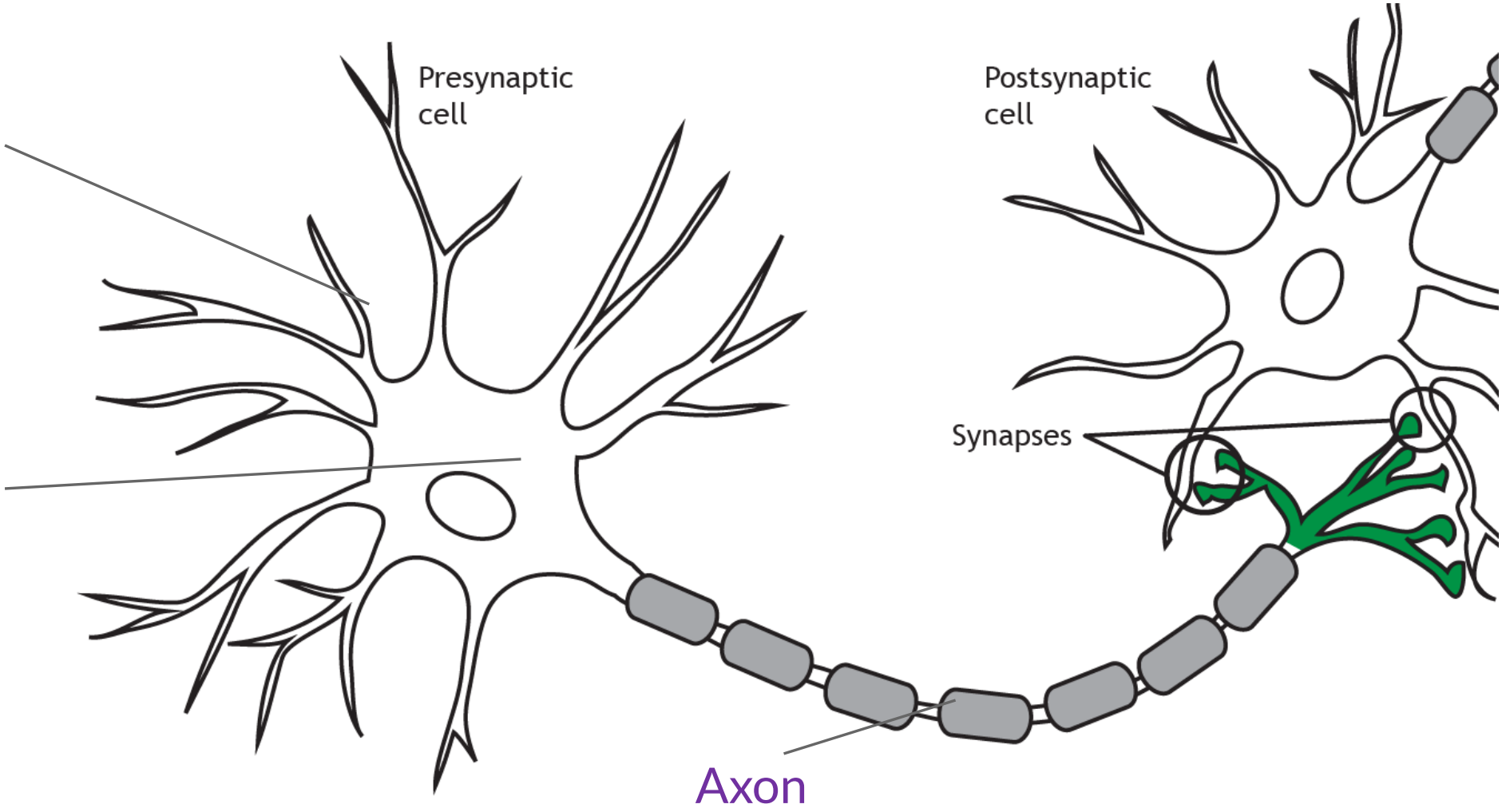
04 Learning and backpropagation

05 Training in practice

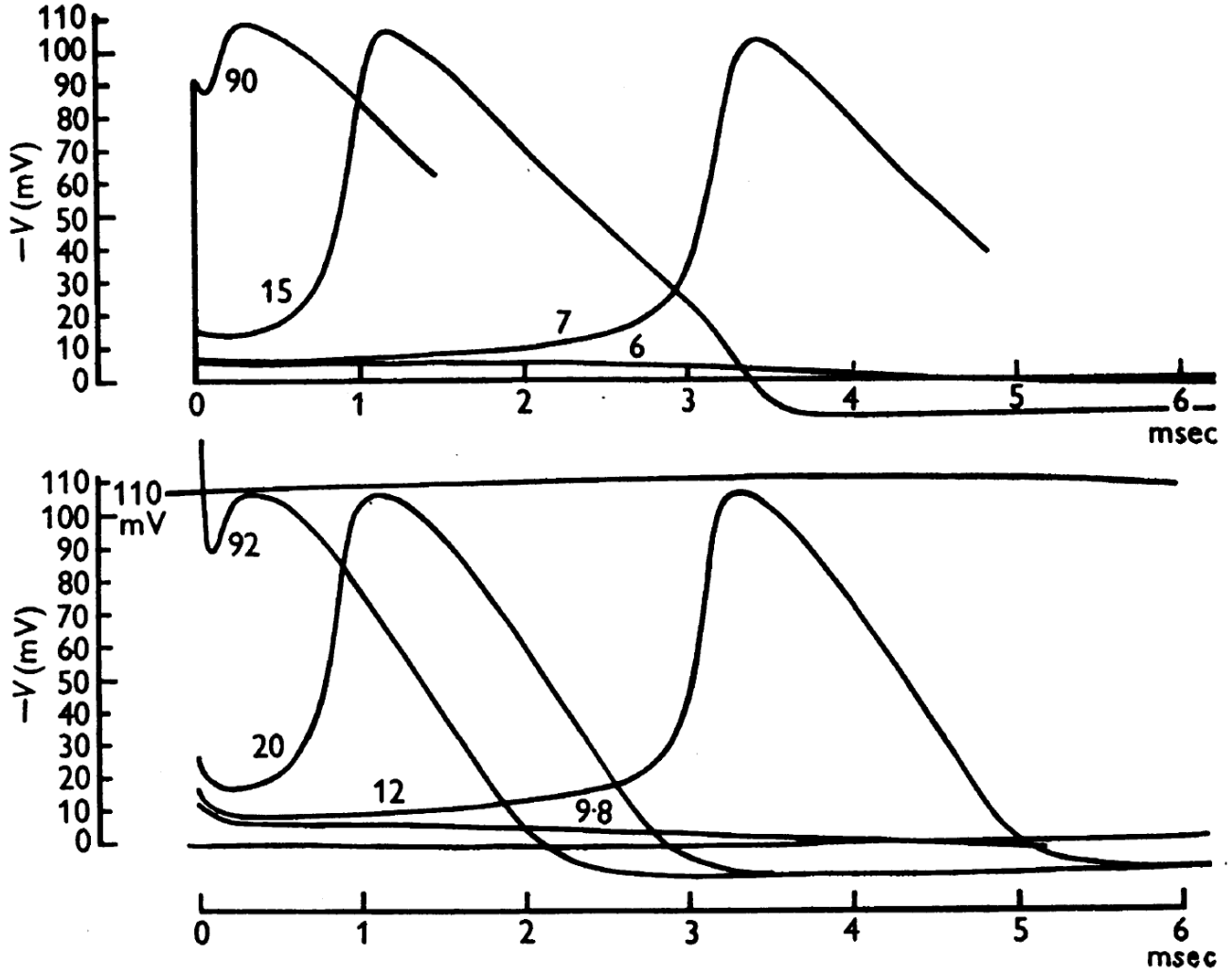
Biological neurons

Dendrites

Cell body



Hodgkin–Huxley model



1952

Calculated

Model equations

Recorded

Squid giant axon

Hodgkin–Huxley dynamics

Four coupled differential equations

$$C_m \frac{dV}{dt} = I(t) - \bar{g}_{\mathrm{Na}} m^3 h (V - E_{\mathrm{Na}}) - \bar{g}_{\mathrm{K}} n^4 (V - E_{\mathrm{K}}) - g_{\mathrm{L}} (V - E_{\mathrm{L}})$$

$$\frac{dm}{dt} = \alpha_m(V)(1-m) - \beta_m(V) m$$

$$\frac{dh}{dt} = \alpha_h(V)(1-h) - \beta_h(V) h$$

$$\frac{dn}{dt} = \alpha_n(V)(1-n) - \beta_n(V) n$$

V

Membrane voltage

m, h, n

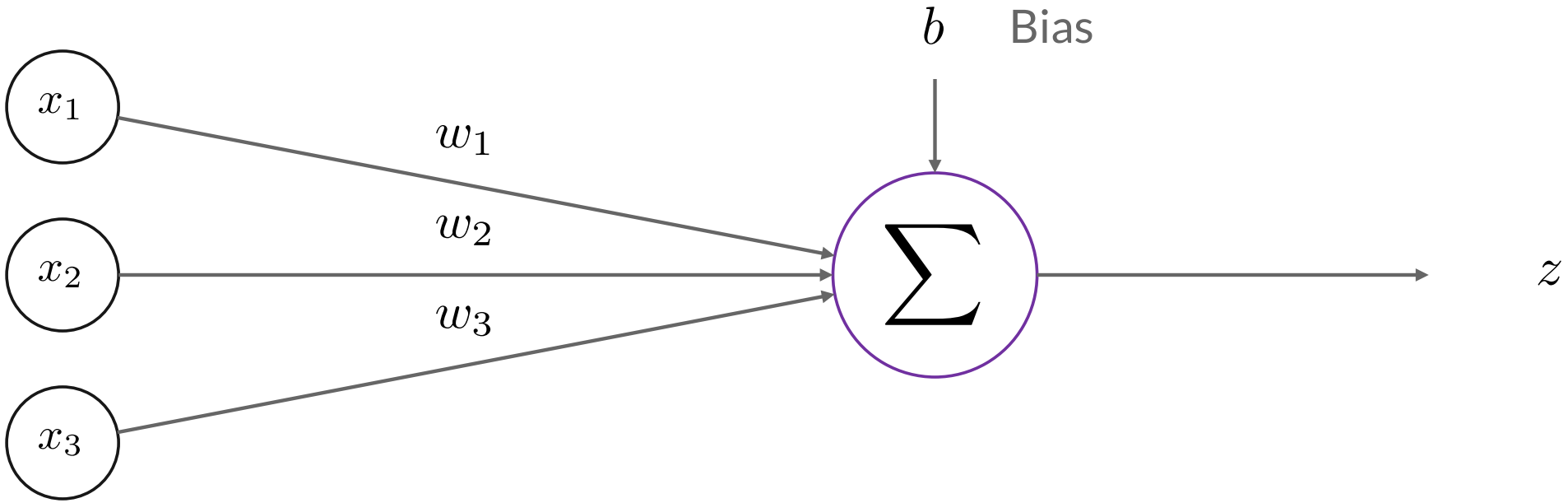
Channel gating

Biological inspiration and mathematical models

$$z = w^{\mathsf{T}} x + b, \qquad h = \phi(z)$$

- Biology motivated weighted connections and nonlinear responses.
- Our goal is to learn useful functions from data.
- These networks do not simulate ion currents or spike timing.

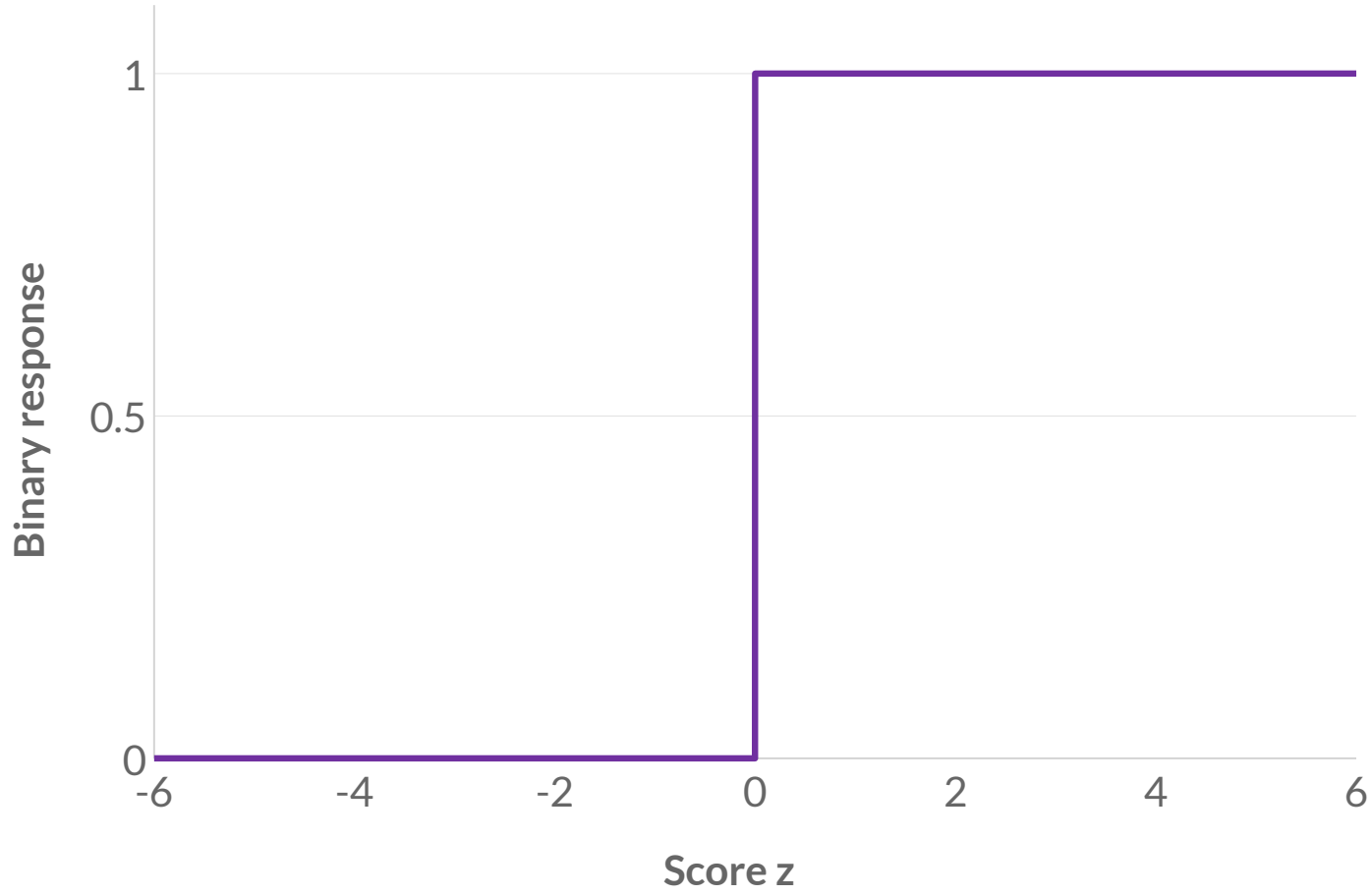
Weighted inputs



$$z = w^{\mathsf{T}} x + b = \sum_{j=1}^{d} w_j x_j + b$$

- Each weight scales its input's contribution.
- The bias adds an offset to the weighted sum.

Threshold neuron

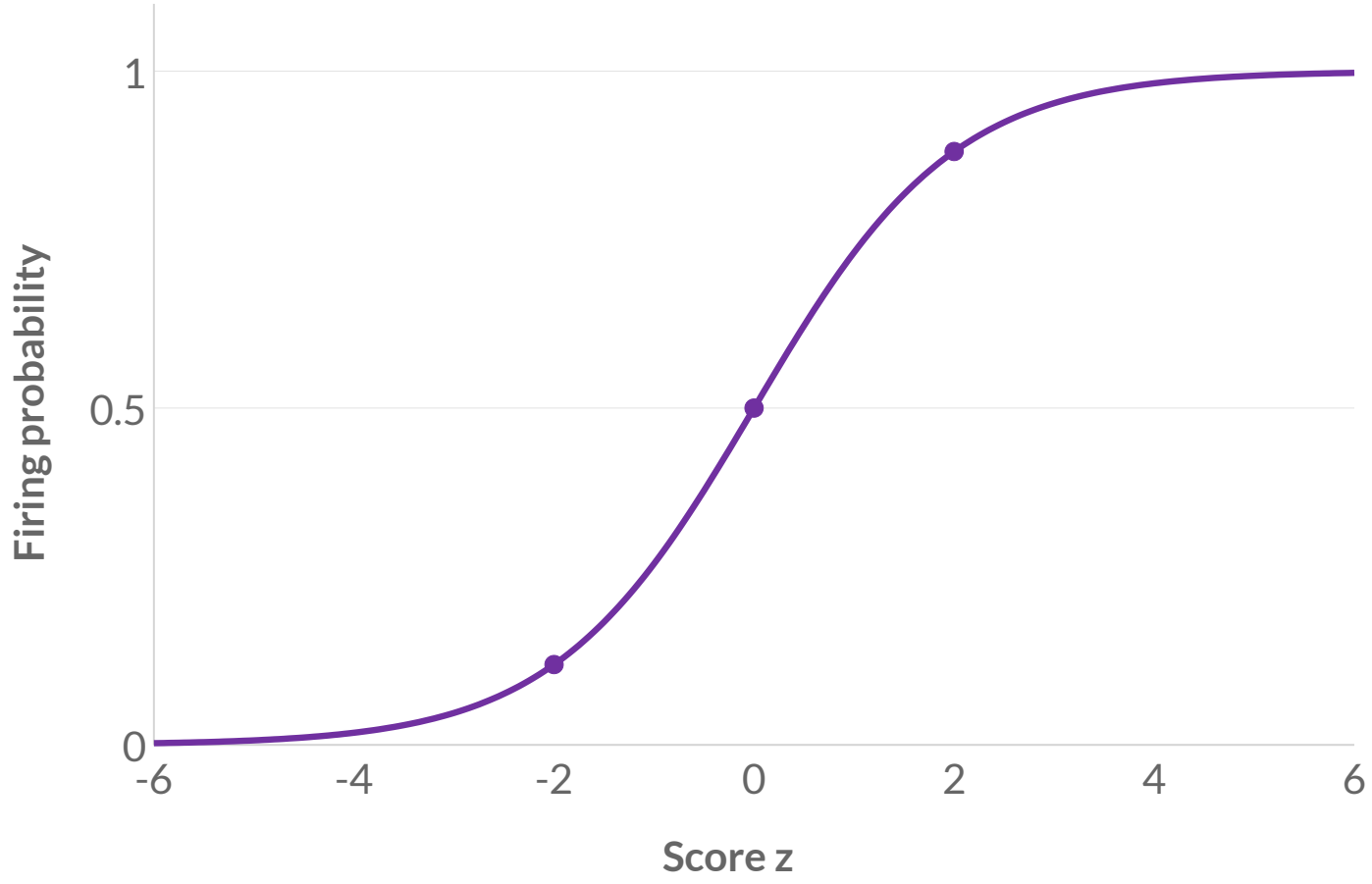


$$s = \mathbf{1}\{z \geq 0\}$$

$$z = \sum_i w_i x_i + b$$

- Active: 1
- Inactive: 0

Sigmoid firing probability



$$p = \sigma(z) = \frac{1}{1 + e^{-z}}$$

$$\Pr(s = 1 \mid x) = p$$

$$\sigma(-2) \approx 0.12$$

$$\sigma(0) = 0.50$$

$$\sigma(2) \approx 0.88$$

Binary and continuous activations

Stochastic binary unit

$$s \sim \text{Bernoulli}(p)$$

$$\mathbb{E}[s \mid x] = p$$

- Each sampled event is 0 or 1.

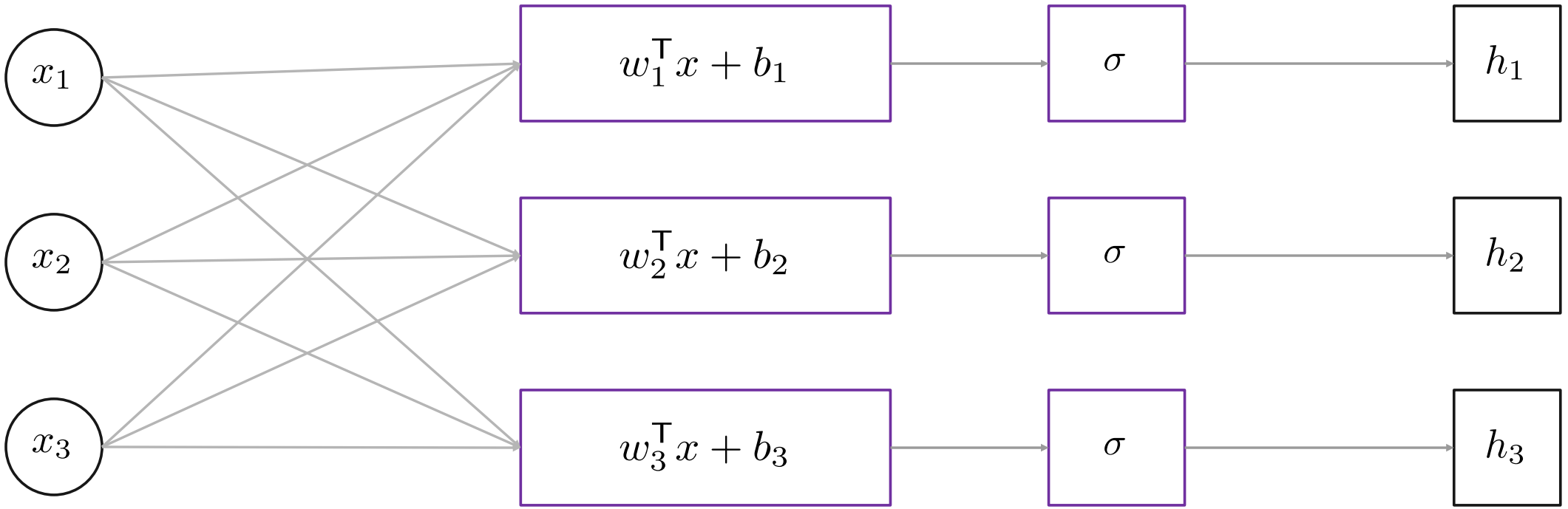
Ordinary sigmoid MLP

$$h = \sigma(z) = p$$

$$\text{For } p = 0.8: \quad h = 0.8$$

- Pass the continuous response.

A layer of neurons



$$h_j = \sigma(w_j^\top x + b_j), \qquad j = 1, 2, 3$$

- Same input; different weights and biases.

Layer notation

$$z_j = w_j^\top x + b_j$$

$$z = Wx + b, \qquad W = \begin{bmatrix} w_1^\top \\ w_2^\top \\ \vdots \\ w_m^\top \end{bmatrix}$$

- One row of W belongs to one output unit.
- An affine map includes the bias.

Vectors, matrices and dimensions

$$x \in \mathbb{R}^d, \quad W \in \mathbb{R}^{m \times d}, \quad Wx \in \mathbb{R}^m$$

$$W = \begin{bmatrix} 1 & -1 \\ 2 & 0 \\ 0 & 1 \end{bmatrix}, \quad x = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \quad Wx = \begin{bmatrix} 1 \\ 4 \\ 1 \end{bmatrix}$$

- A column is one example; coordinates are its features.
- Rows of W specify different weighted sums.

Elementwise activation

$$h = \sigma(z) = \begin{bmatrix} \sigma(z_1) \\ \sigma(z_2) \\ \sigma(z_3) \end{bmatrix}$$

$$\begin{bmatrix} -2 \\ 0 \\ 2 \end{bmatrix} \xrightarrow{\sigma} \begin{bmatrix} 0.12 \\ 0.50 \\ 0.88 \end{bmatrix}$$

- Apply the scalar function separately to each entry.

Matrix products and elementwise products

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \qquad B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

Matrix product

$$AB = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

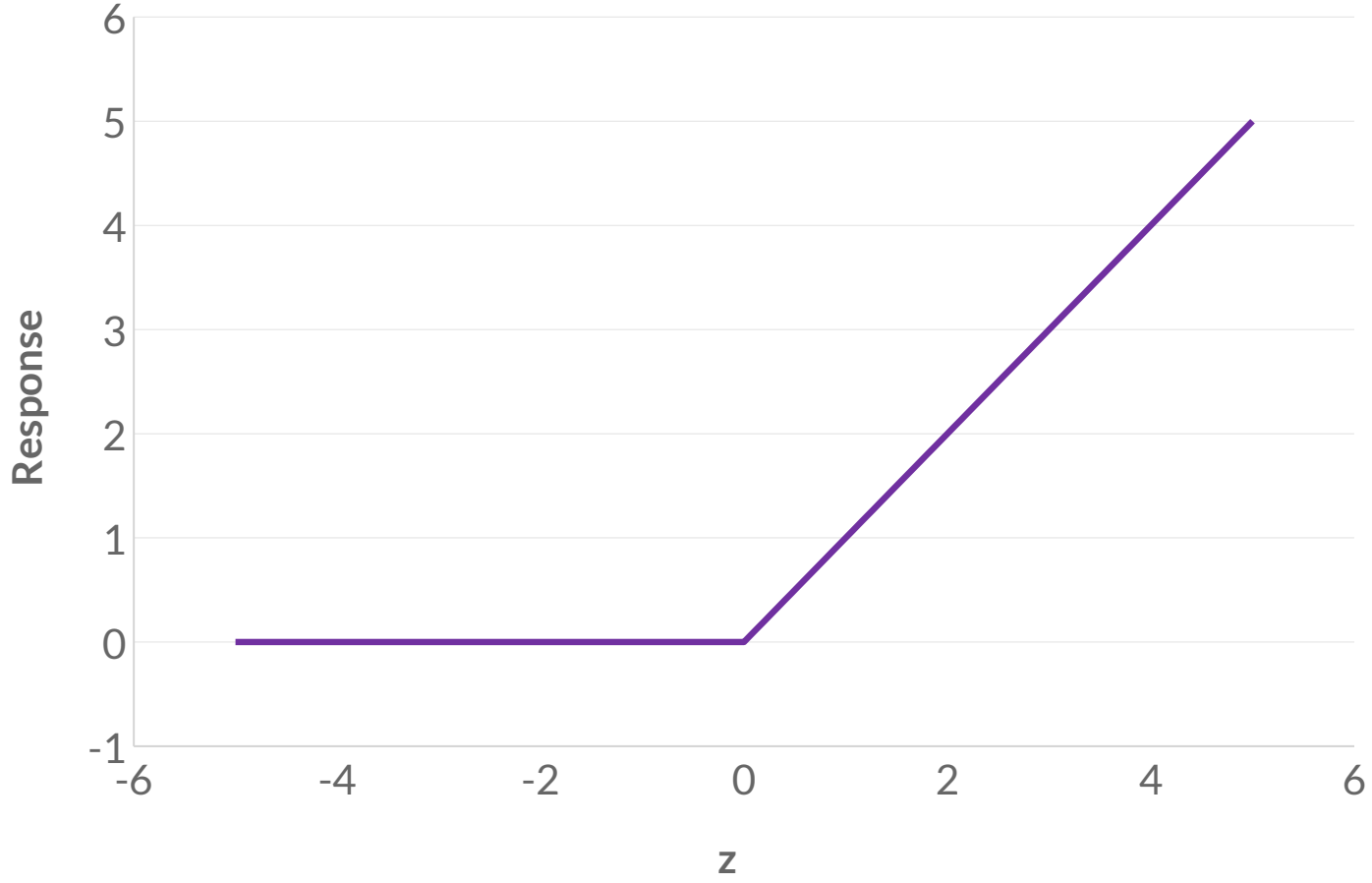
- Combines rows with columns.

Elementwise product

$$A \odot B = \begin{bmatrix} 5 & 12 \\ 21 & 32 \end{bmatrix}$$

- Multiplies matching entries.

ReLU activation



$$h = \phi(z)$$

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

- Negative input $\rightarrow 0$
- Positive input $\rightarrow$ unchanged response

Batched computation

$$X = [x_1 \; x_2 \; \cdots \; x_B] \in \mathbb{R}^{d \times B}$$

$$WX = [Wx_1 \; Wx_2 \; \cdots \; Wx_B] \in \mathbb{R}^{m \times B}$$

- The same weights are applied to every example.
- A batch is a collection of examples, not a new layer.

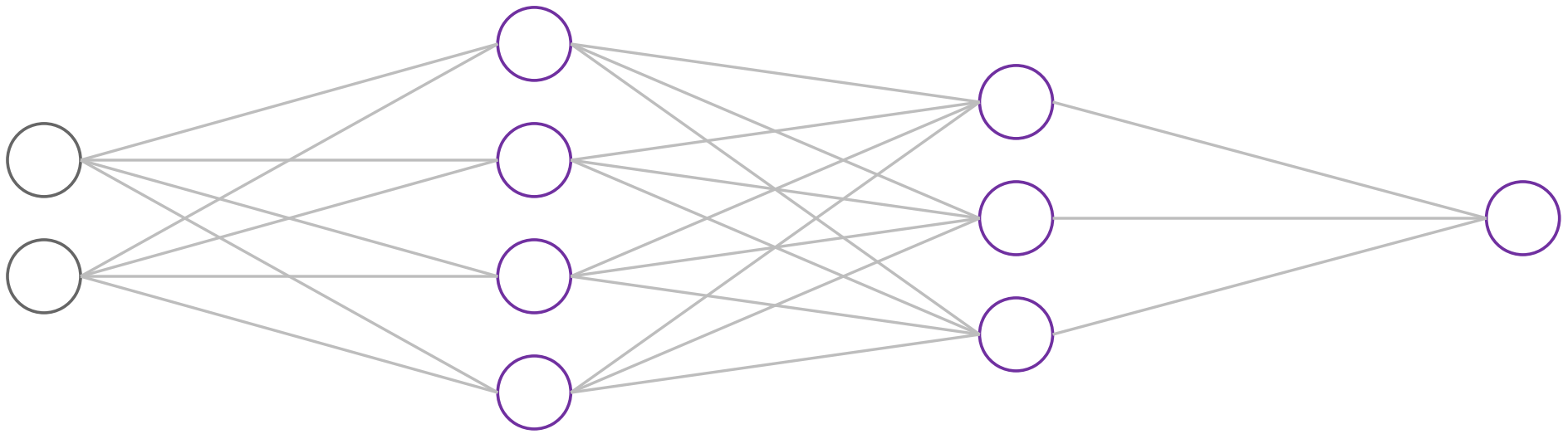
Biases and broadcasting

$$Z = WX + b\mathbf{1}_B^\top$$

$$b = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, \quad b\mathbf{1}_3^\top = \begin{bmatrix} 1 & 1 & 1 \\ -2 & -2 & -2 \end{bmatrix}$$

- Each output unit shares its bias across the batch.
- Broadcasting expresses the repeated addition.

Multilayer perceptron



Input | Hidden layer 1 | Hidden layer 2 | Output

- Hidden layers transform the representation.
- The output layer produces the task's prediction or scores.

MLP equations

$$h^{[1]} = \phi(W^{[1]}x + b^{[1]})$$

$$h^{[2]} = \phi(W^{[2]}h^{[1]} + b^{[2]})$$

$$s = W^{[3]}h^{[2]} + b^{[3]}$$

- Each layer uses the previous layer's responses.

An MLP in PyTorch

```
import torch
from torch import nn

model = nn.Sequential(
    nn.Linear(1, 32), nn.ReLU(),
    nn.Linear(32, 32), nn.ReLU(),
    nn.Linear(32, 1),
)
x = torch.tensor([[1.5], [2.0]])
prediction = model(x)
```

- Tensors store arrays of numbers.
- Two hidden layers, each of width 32
- One numerical output per example

A forward pass

Parameters and input

$$x = 1.5$$

$$w = (1, 1, 1)^\mathsf{T}$$

$$b = (0, -1, -2)^\mathsf{T}$$

$$v = (1, -1, 1)^\mathsf{T}, \quad c = 0$$

Computed values

$$z = wx + b = (1.5, 0.5, -0.5)^\mathsf{T}$$

$$h = \mathrm{ReLU}(z) = (1.5, 0.5, 0)^\mathsf{T}$$

$$\widehat{y} = v^\mathsf{T} h + c = 1$$

Function composition

$$f_\theta = f_L \circ f_{L-1} \circ \cdots \circ f_1$$

$$h^{[l]} = f_l(h^{[l-1]})$$

- Width: several responses computed in parallel
- Depth: responses transformed repeatedly

Lecture agenda

01 Neurons, layers and networks

02 Capacity and activations

03 Predictions and losses

04 Learning and backpropagation

05 Training in practice

Composition of affine layers

Without nonlinear activations

$$W_2(W_1x + b_1) + b_2$$

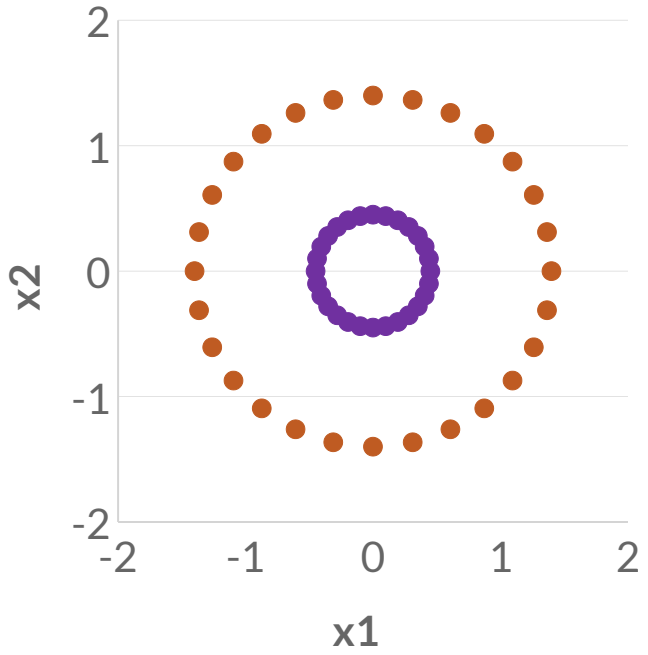
$$= (W_2W_1)x + (W_2b_1 + b_2)$$

Stacking affine maps still gives an affine map.

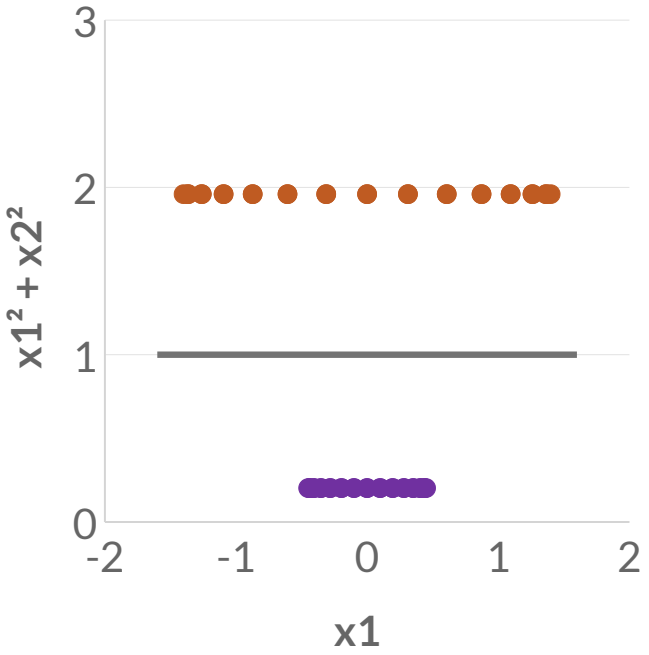
Rank Number of independent output directions.
A narrow linear layer can reduce this number.

Nonlinear feature maps

Input coordinates



Chosen nonlinear coordinates



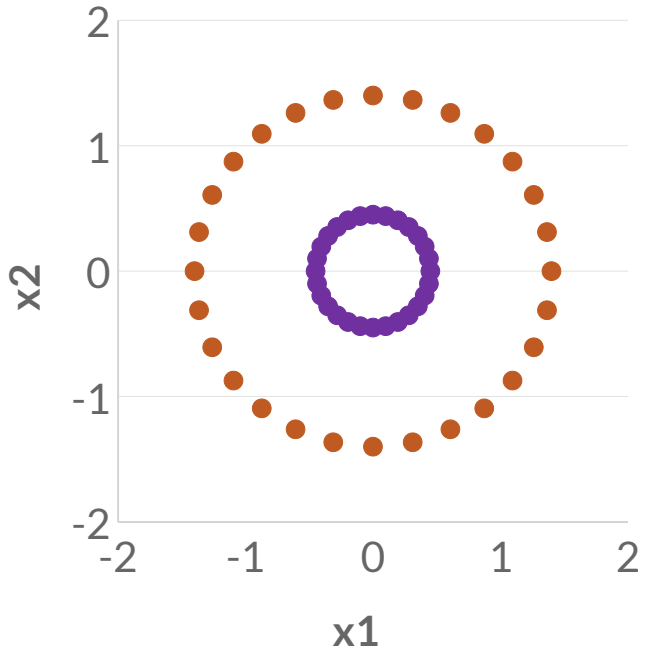
Inner class Outer class Linear boundary

$$s = w^\mathsf{T} x + b$$

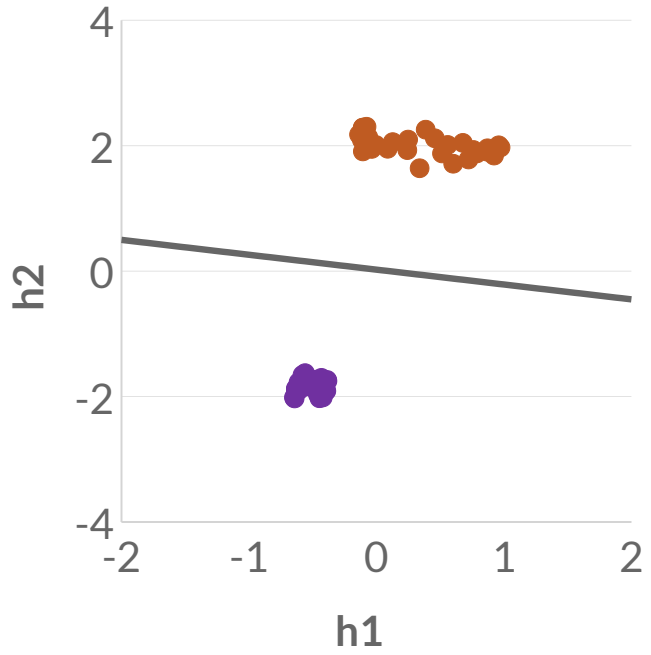
$$s = v^\mathsf{T} \phi(x) + c$$

Learned representations

Input coordinates



Two learned coordinates



Inner class Outer class Linear boundary

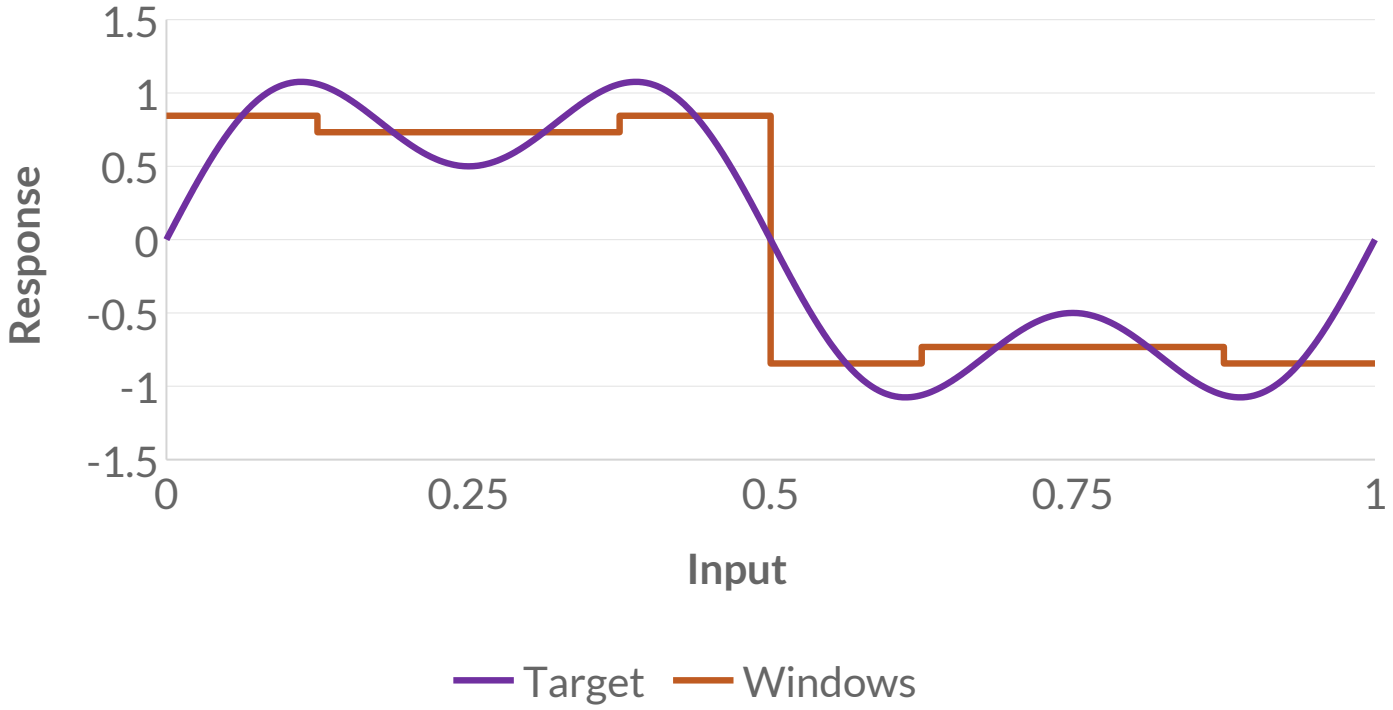
$$h_\theta(x) \in \mathbb{R}^2, \qquad s = v^\mathsf{T} h_\theta(x) + c$$

One hidden layer as an expansion

$$f_\theta(x) = c + \sum_{j=1}^{m} v_j \, \phi(w_j^\mathsf{T} x + b_j)$$

- Hidden units supply the functions being combined.
- Weights and biases can change those functions.

Local window functions

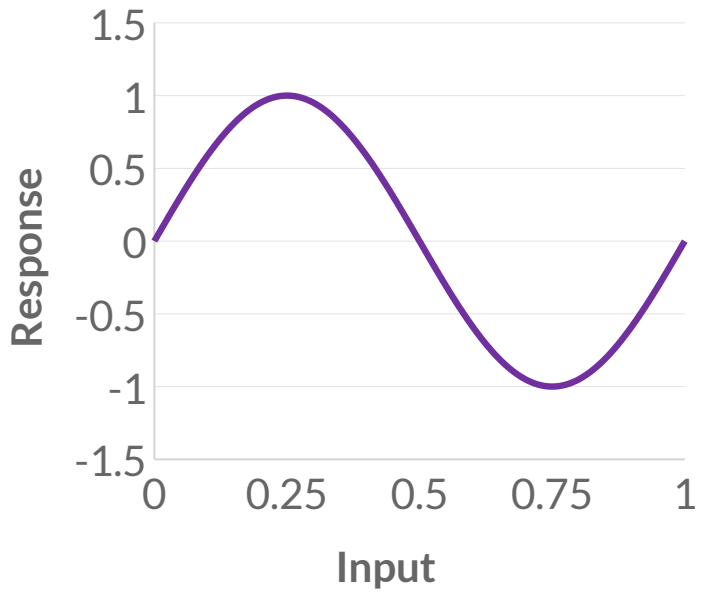


- Each window contributes on a local interval.
- More pieces can describe finer variation.

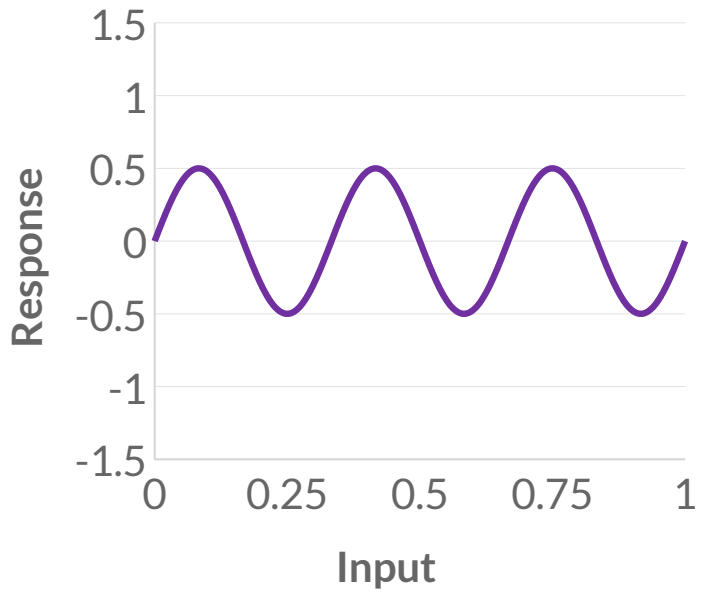
$$f(x) = \sum_j c_j \, \mathbf{1}\{t_j \leq x < t_{j+1}\}$$

A sinusoidal expansion

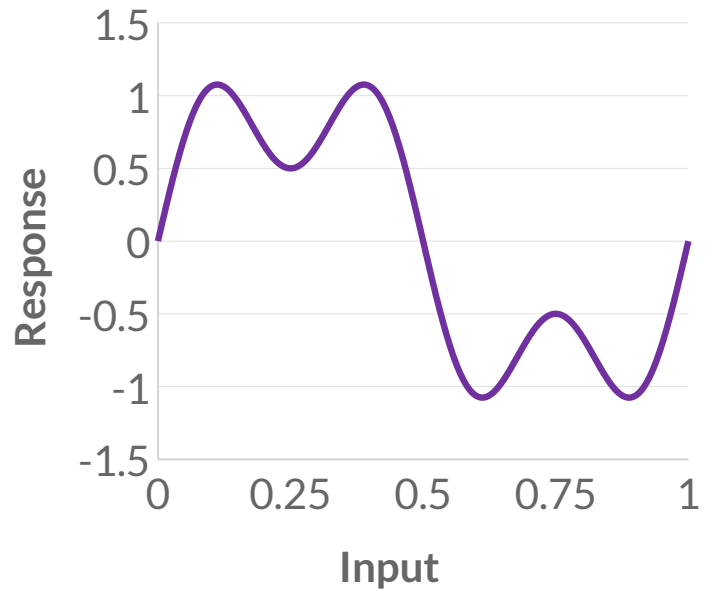
One cycle



Three cycles



Their sum

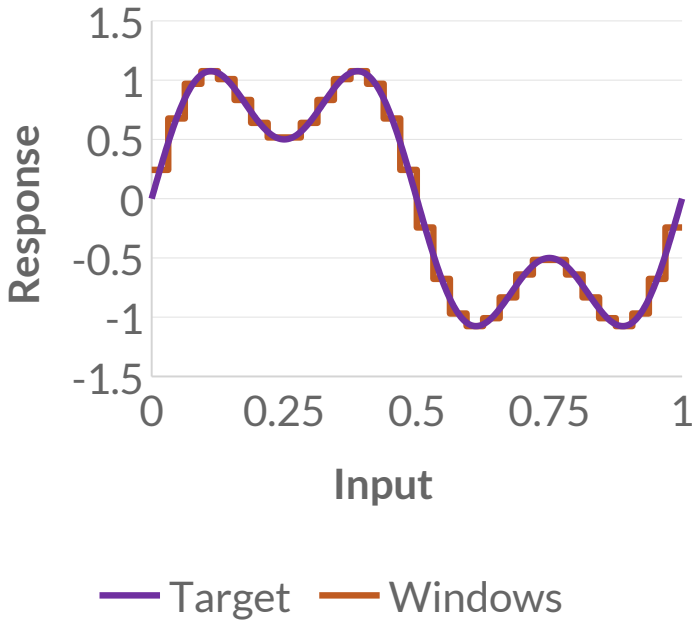


$$f(x) = \sin(2\pi x) + \tfrac{1}{2}\sin(6\pi x)$$

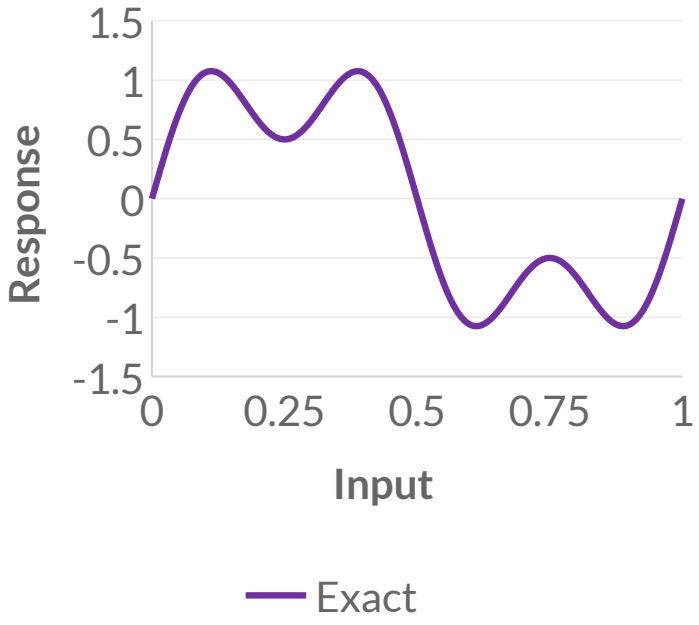
Basis choice and approximation

8 local windows

32 local windows

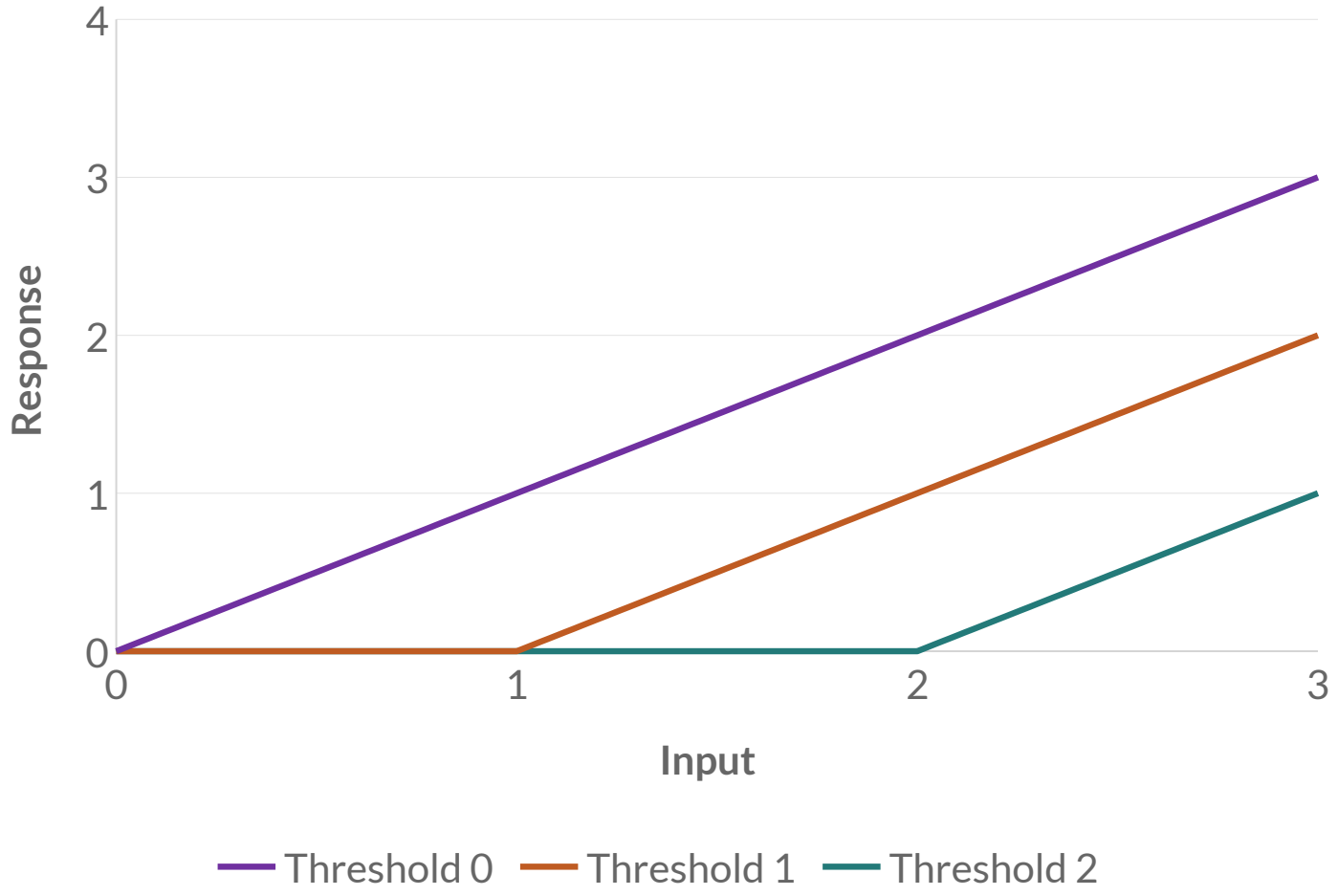


2 sinusoidal units



$$f(x) = \sin(2\pi x) + \tfrac{1}{2}\sin(6\pi x)$$

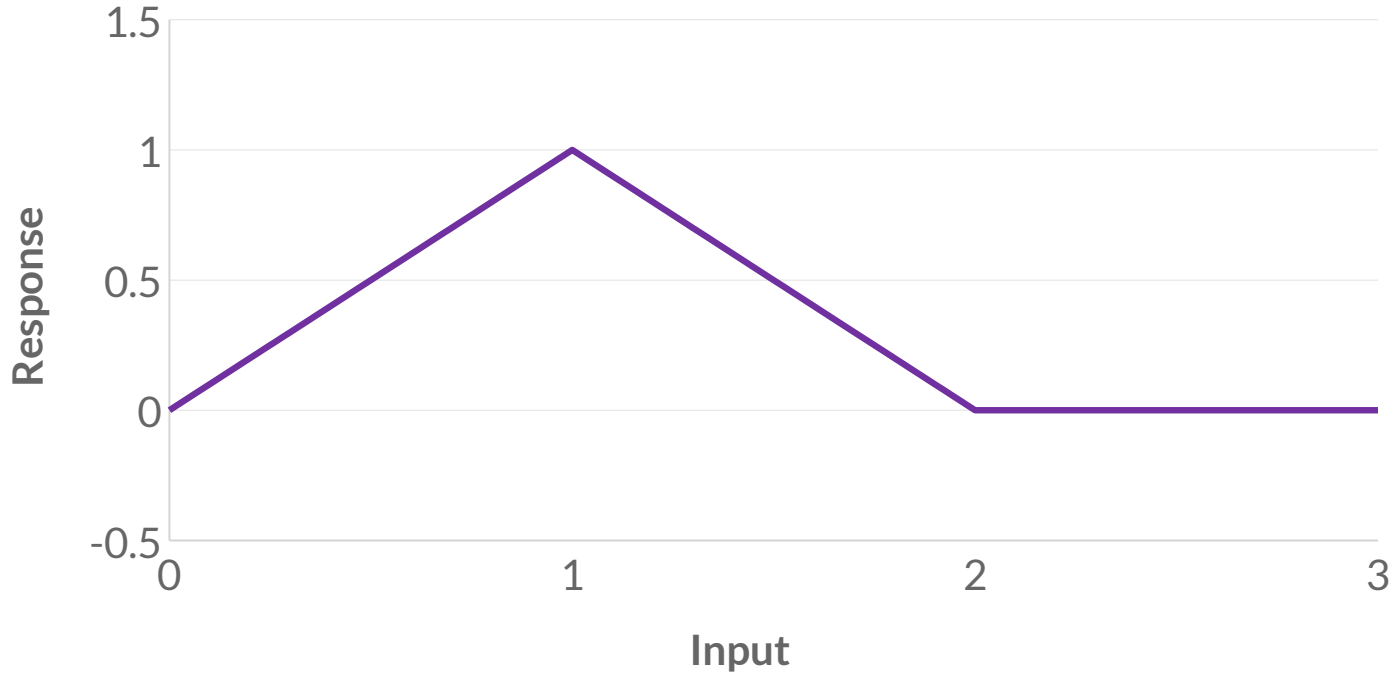
ReLU hinges



$$h_j(x) = \max(0, x - t_j)$$

- The bias sets the location of a bend.
- A weighted unit changes the slope after that point.

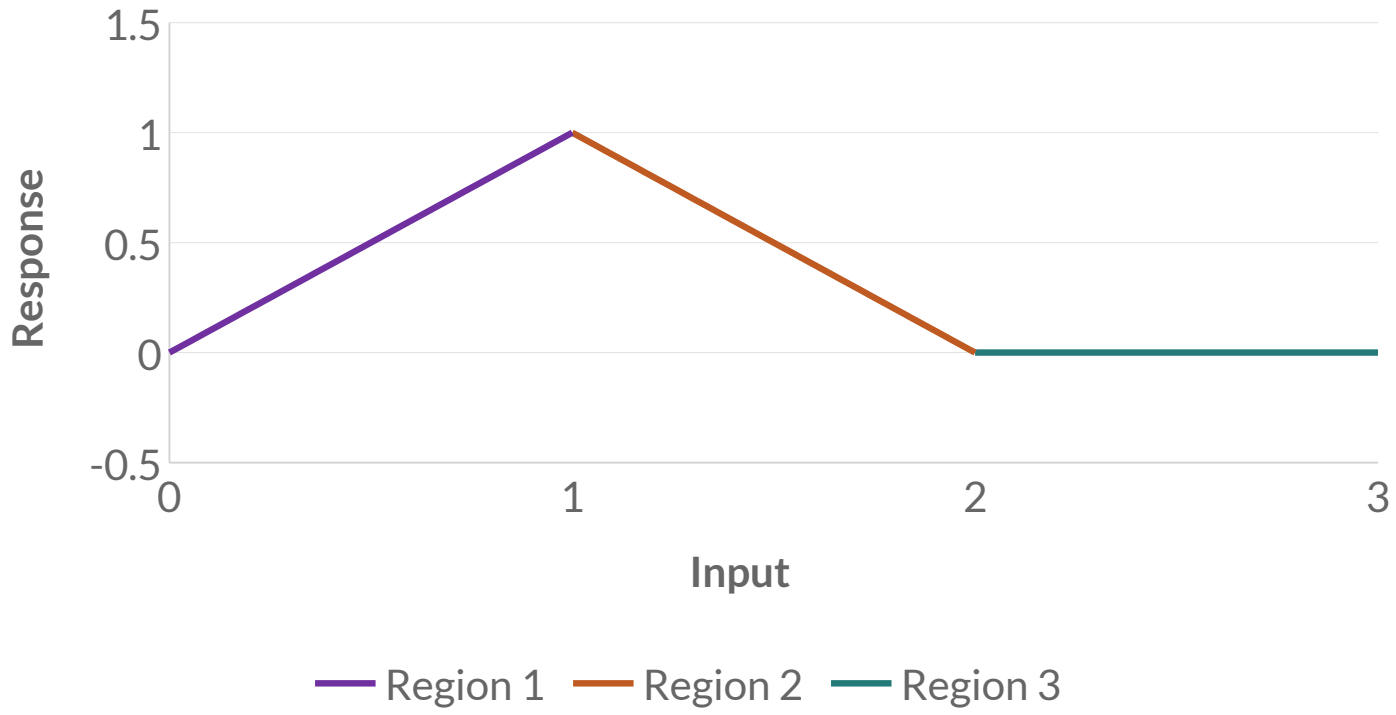
A triangular function



- Slope changes: +1, −2, +1

$$f(x) = \mathrm{ReLU}(x) - 2\,\mathrm{ReLU}(x-1) + \mathrm{ReLU}(x-2)$$

Piecewise-affine functions



- Fixing the active units fixes an affine map.
- Activation patterns define the regions.

$$f(x) = a_r^\mathsf{T} x + c_r \quad \text{within region } r$$

Universal approximation

- Each added ReLU can introduce another bend.
- More bends give a closer match to the target.

Universal approximation theorem

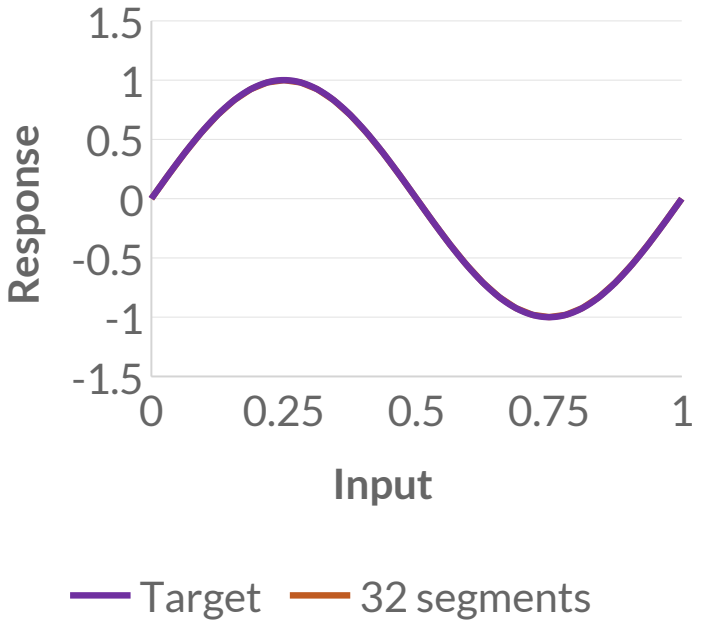
On a fixed, closed and bounded input region, a network with one hidden layer can approximate any continuous function as closely as desired, given enough hidden units.

Limits of the result

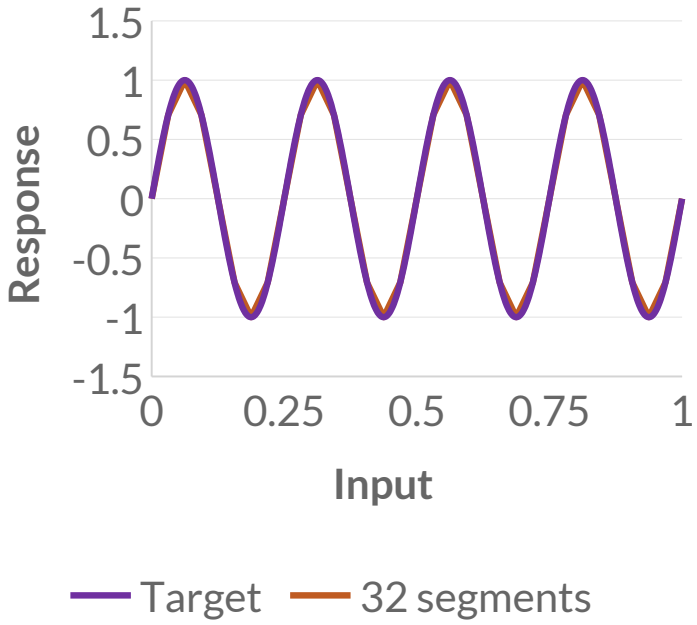
- The required network may be very large.
- The theorem does not tell us how to find its weights.

Approximation cost

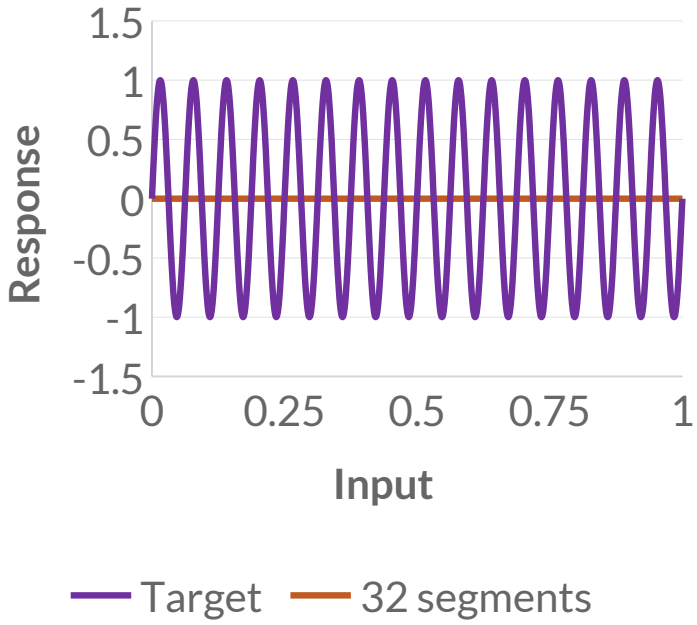
1 cycle



4 cycles

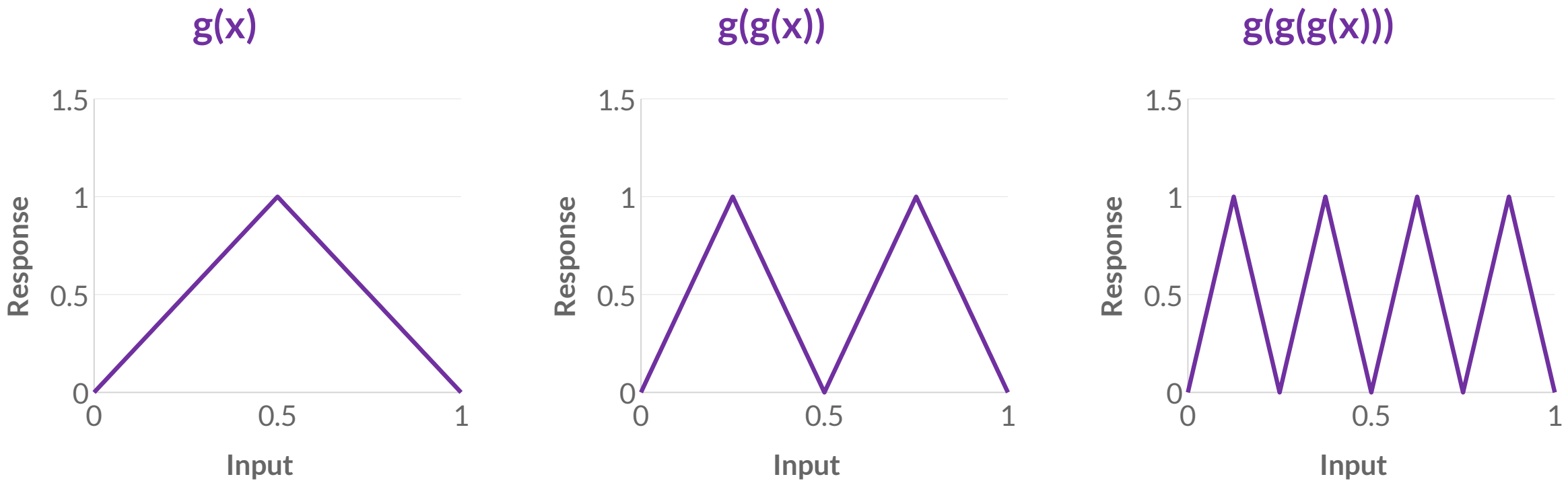


16 cycles



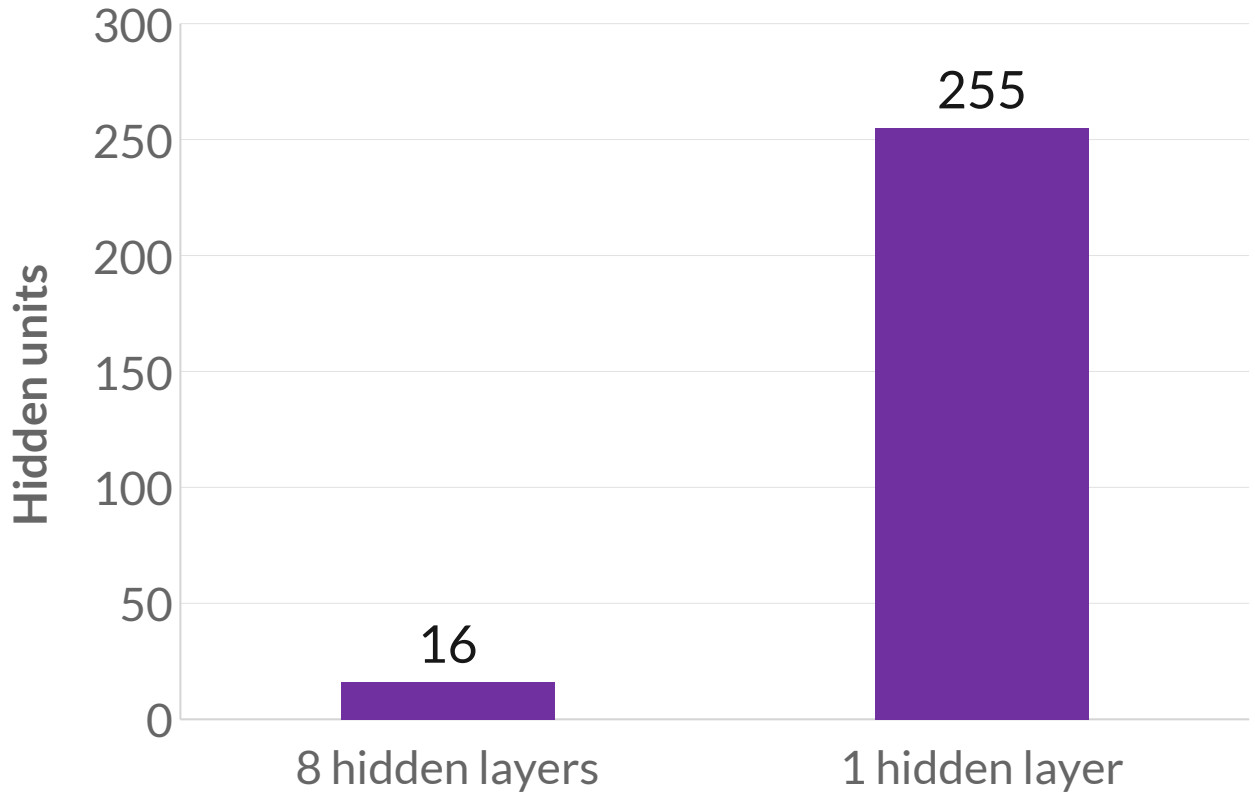
$$f(x) = \sin(2\pi\omega x), \qquad 32 \text{ linear segments in every panel}$$

Repeated composition



$$g(x) = 2\,\mathrm{ReLU}(x) - 4\,\mathrm{ReLU}(x - \tfrac{1}{2}), \quad x \in [0, 1]$$

Depth and width



Eightfold composition

Depth 8, width 2

16 hidden units
49 parameters

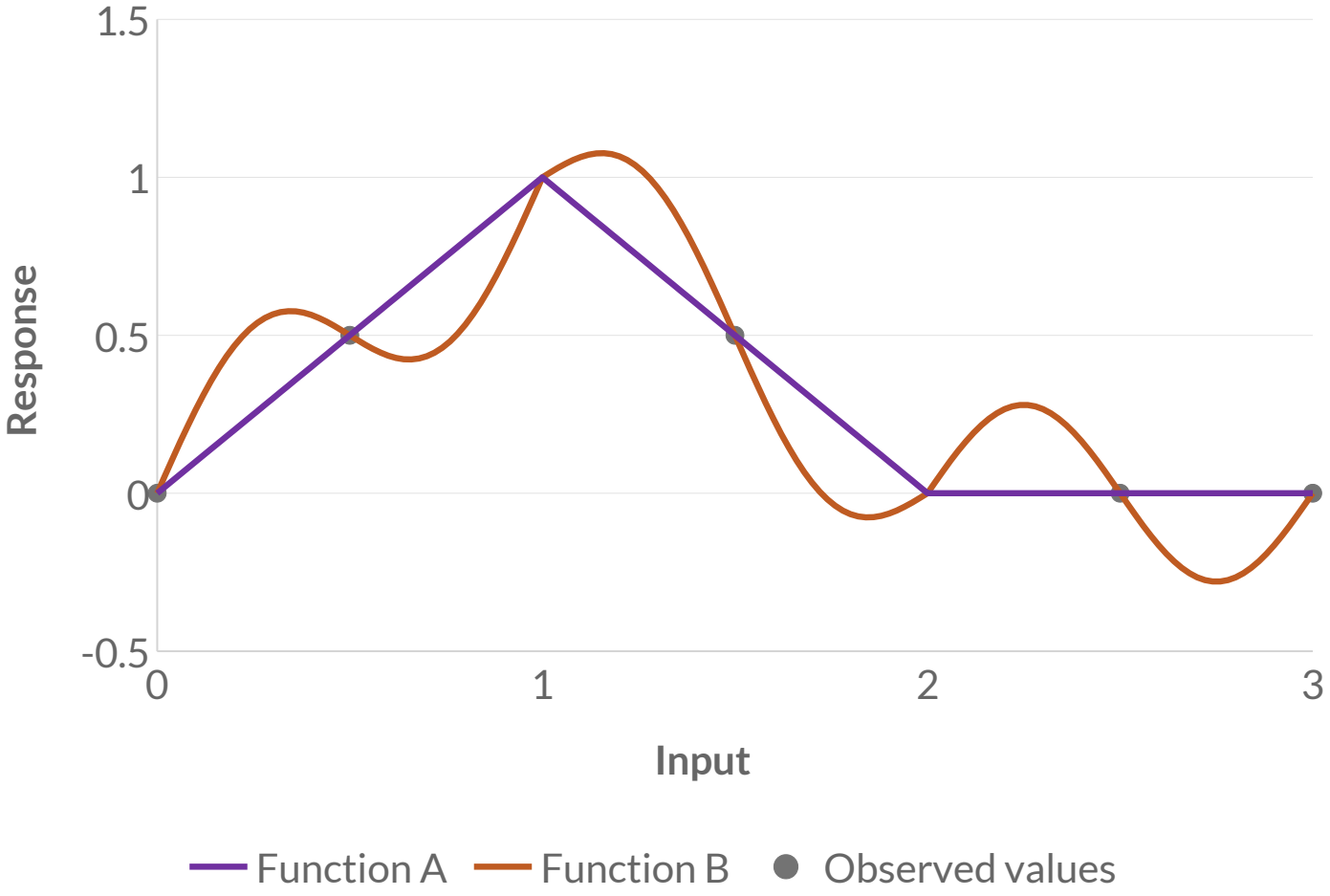
One hidden layer

At least 255 hidden units
At least 766 parameters

$$g^{\circ k} \text{ has } 2^k \text{ linear pieces on } [0,1]$$

$$\text{one hidden layer:} \quad m + 1 \geq 2^k$$

Interpolation and generalization



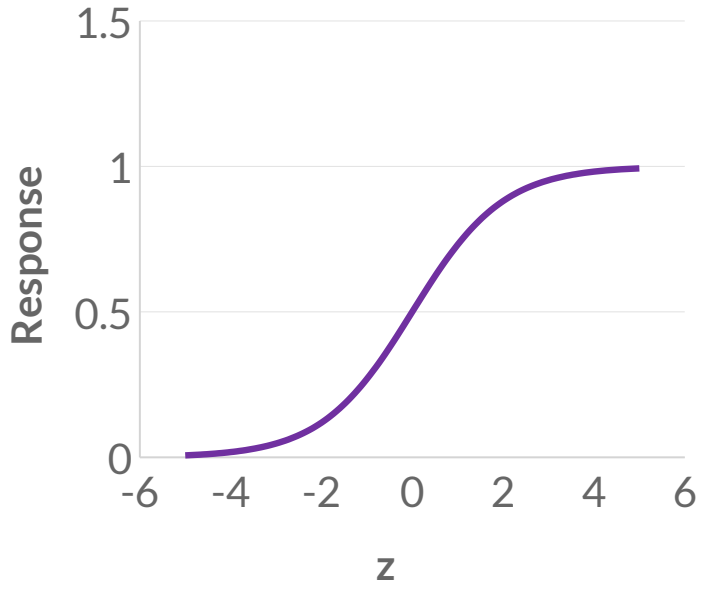
- Different functions can fit the same observations.
- Their predictions between observations can differ.

Capacity, optimization and generalization

- Capacity: does the model family contain a useful function?
- Optimization: can training find suitable parameters?
- Generalization: do the predictions work on unseen data?

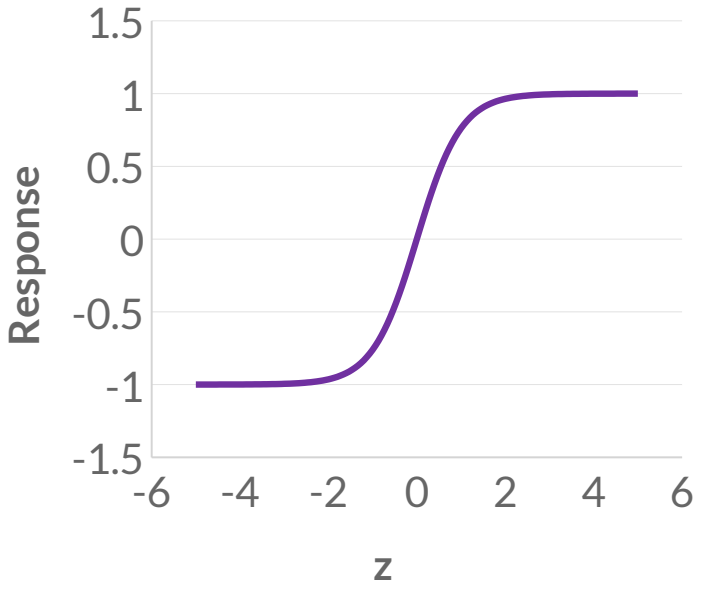
Hidden-layer activations

Sigmoid



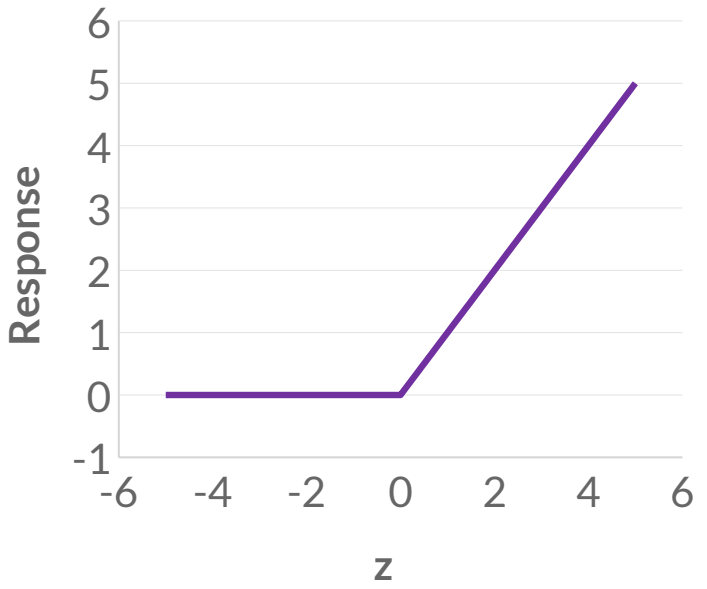
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Tanh



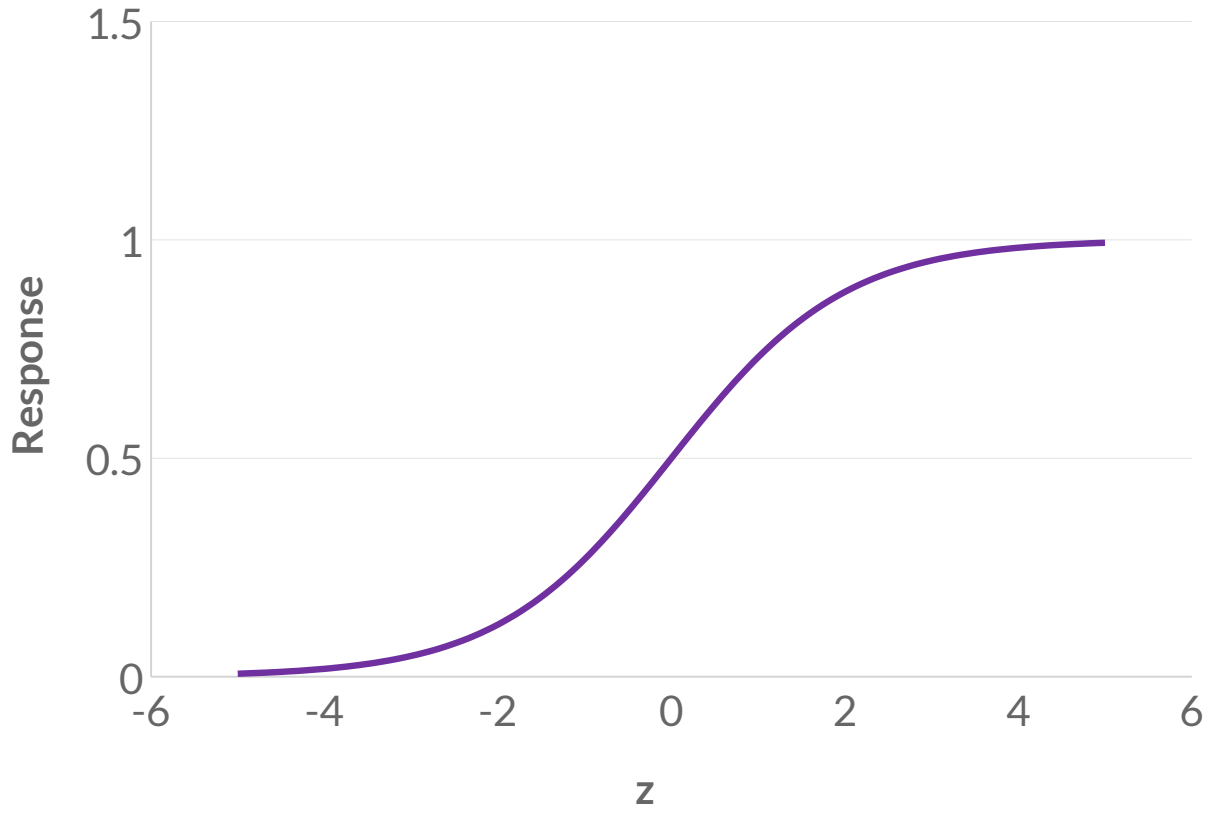
$$\tanh(z)$$

ReLU



$$\max(0, z)$$

Sigmoid



$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

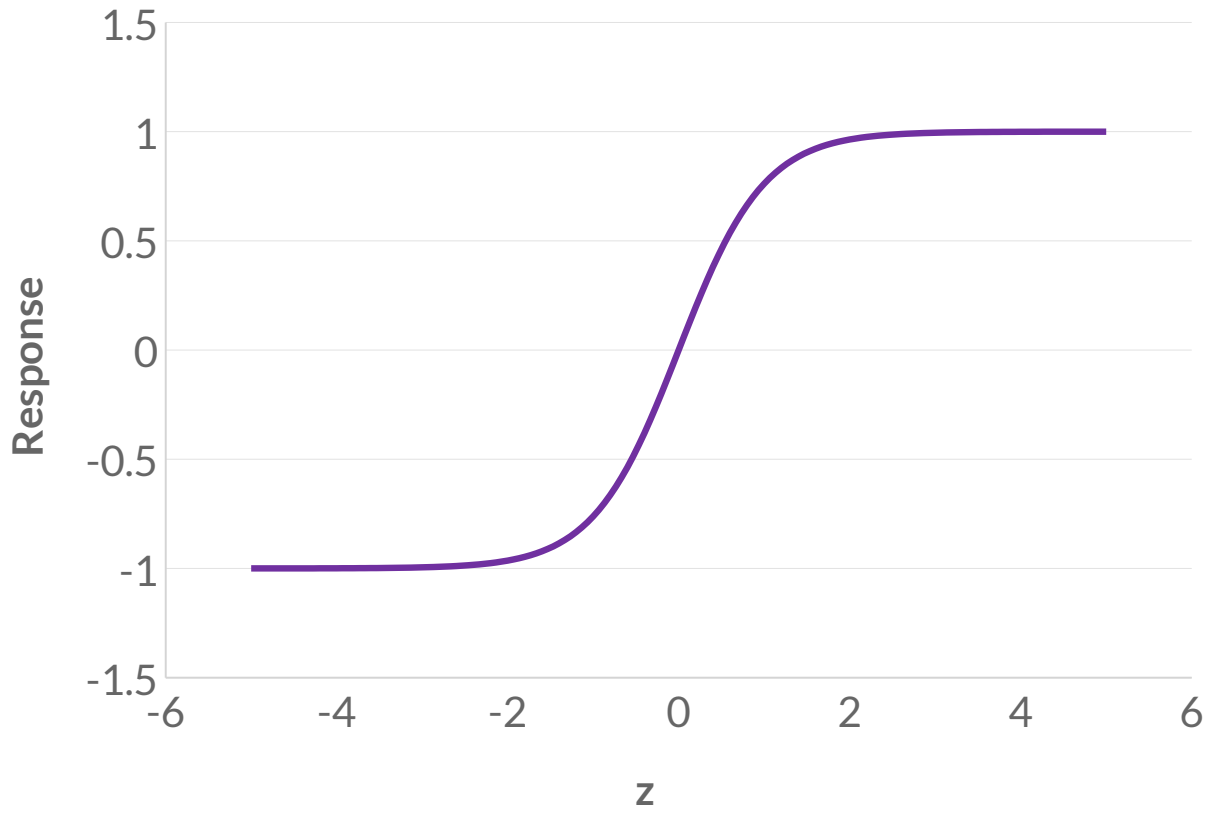
$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

- Smooth, with range (0, 1)
- Saturation gives small derivatives in both tails

Used for

Binary probabilities and gates

Tanh



$$\tanh(z) = 2\sigma(2z) - 1$$

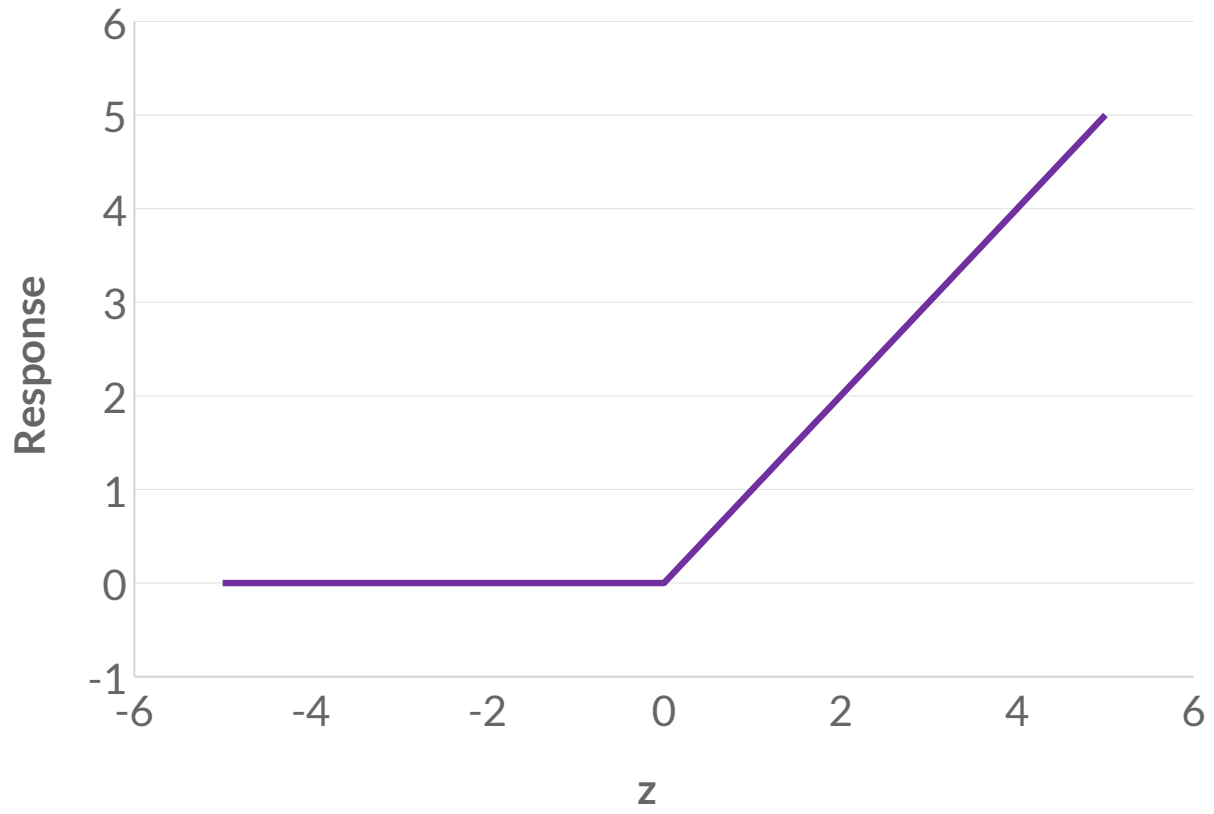
$$\tanh'(z) = 1 - \tanh^2(z)$$

- Signed range (−1, 1)
- Zero-centered, but saturating

Used for

Bounded outputs, recurrent states

ReLU



$$\mathrm{ReLU}(z) = \max(0, z)$$

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

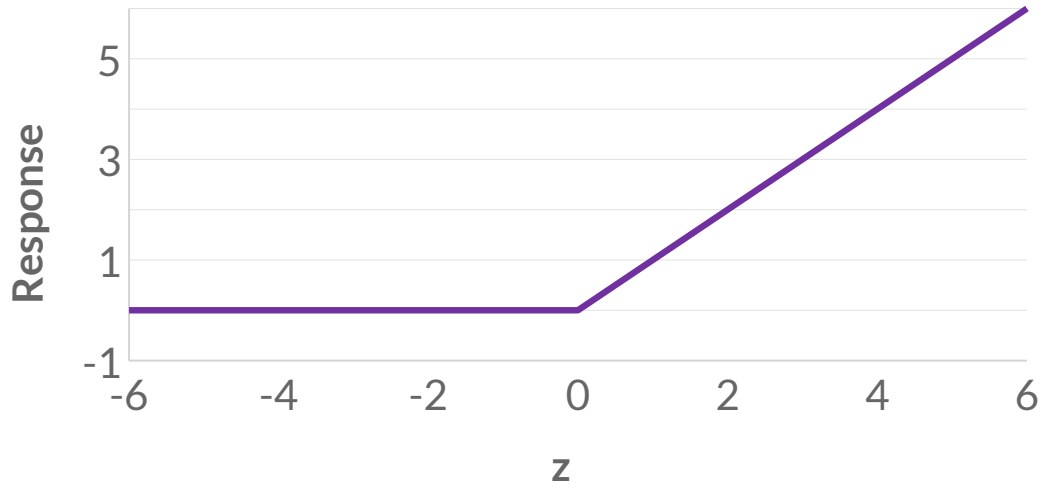
- Cheap; no saturation on the positive branch
- Zero output and local gradient for negative inputs

Used for

MLPs and convolutional networks

ReLU and Leaky ReLU

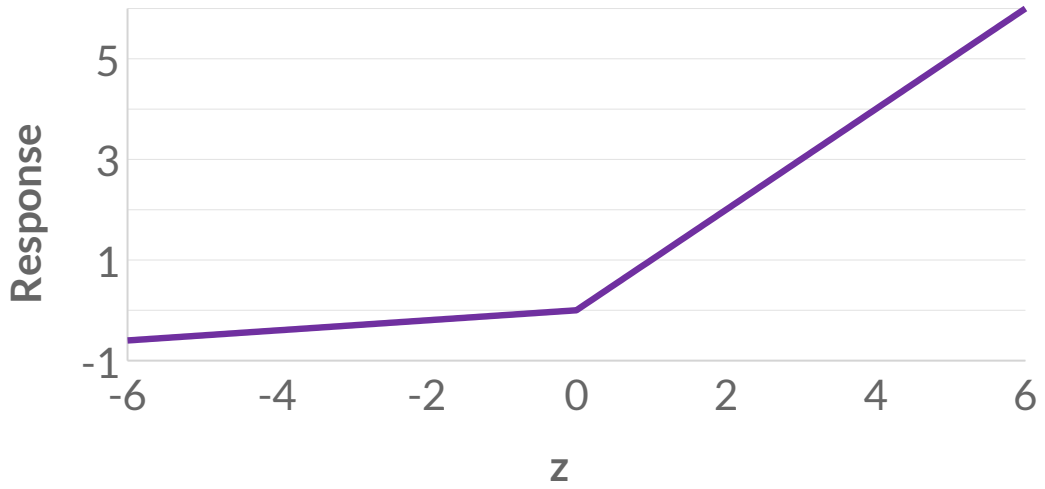
ReLU



$$\max(0, z)$$

- Zero response for negative inputs
- A unit may stay inactive across the data

Leaky ReLU

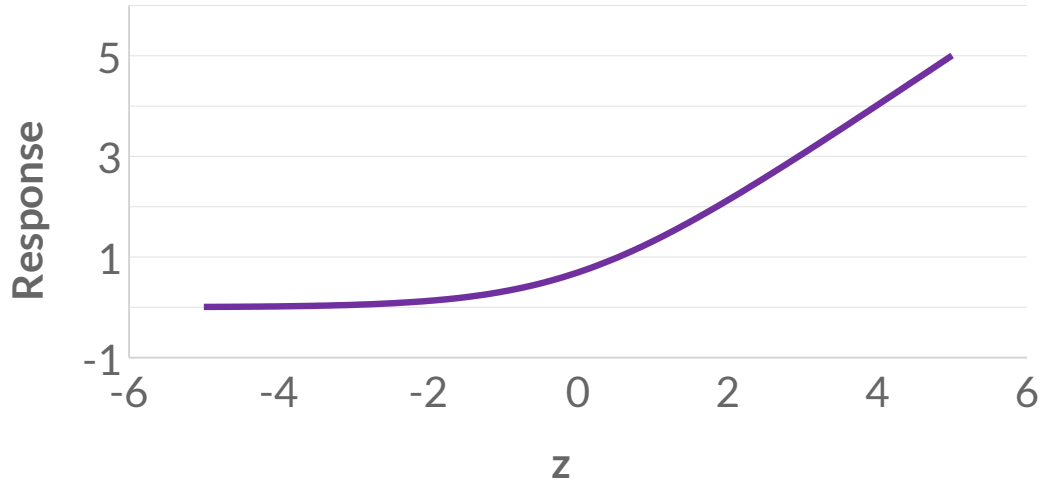


$$\max(\alpha z, z), \quad 0 < \alpha < 1$$

- Nonzero slope for negative inputs
- PReLU learns this slope

Softplus and ELU

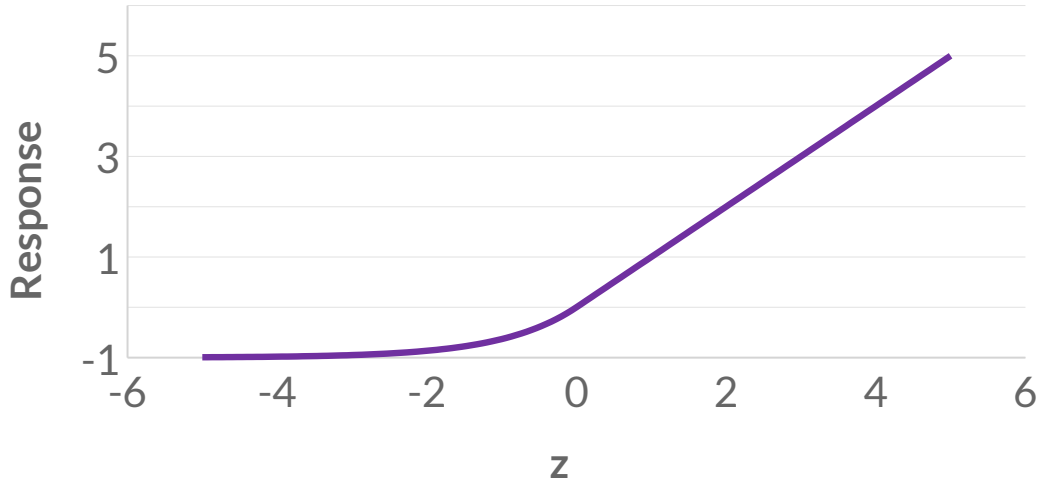
Softplus



$$\log(1 + e^z)$$

- Smooth and strictly positive
- Useful for positive scales or rates

ELU, $\alpha = 1$

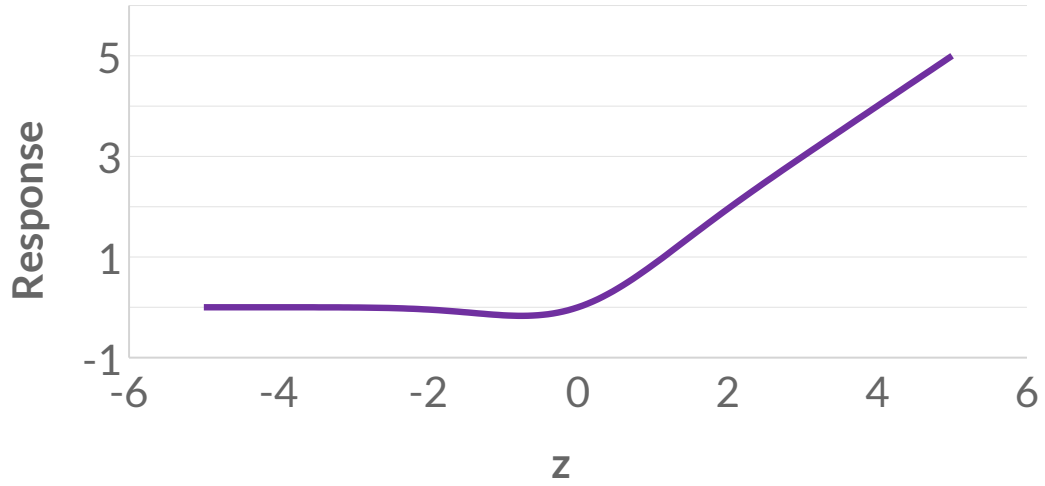


$$\begin{cases} z, & z > 0 \\ e^z - 1, & z \leq 0 \end{cases}$$

- Negative values; positive linear branch
- Alternative hidden activation

GELU and SiLU

GELU

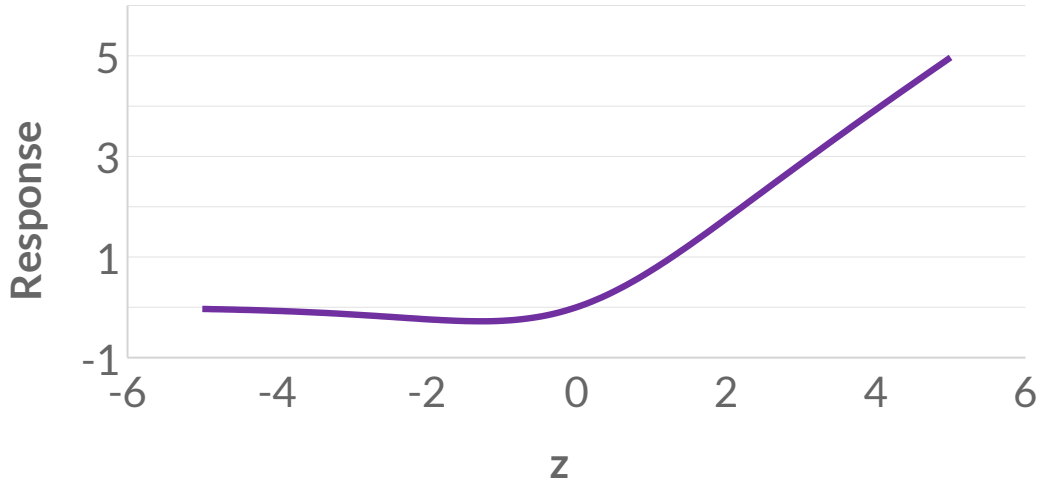


$$\mathrm{GELU}(z) = z\Phi(z)$$

Φ: standard normal CDF

- Smooth weighting by a normal CDF
- Used in BERT, a language model

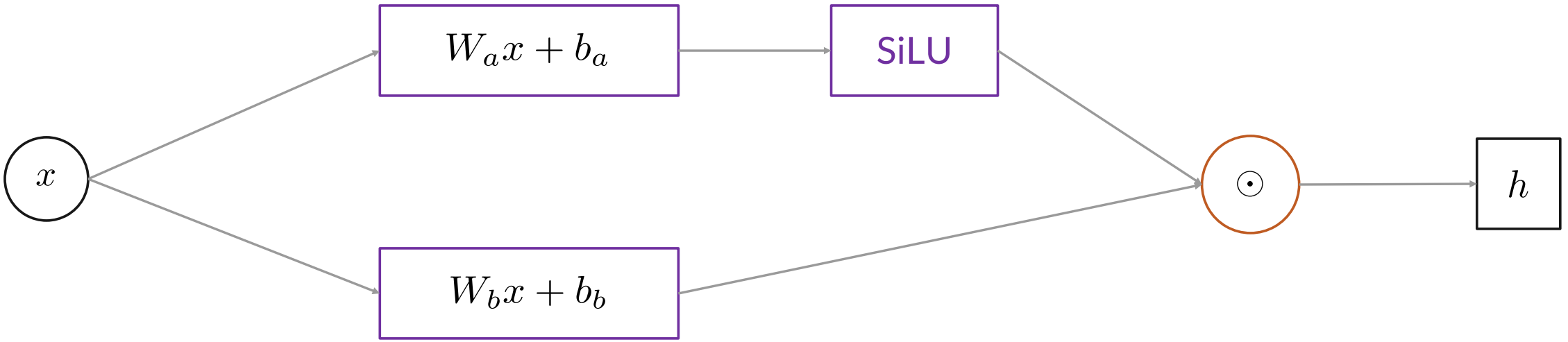
SiLU / Swish with $\beta = 1$



$$\mathrm{SiLU}(z) = z\sigma(z)$$

- Smooth weighting by a sigmoid
- Used within SwiGLU layers

SwiGLU



Two learned projections

$$h = \mathrm{SiLU}(W_a x + b_a) \odot (W_b x + b_b)$$

$$y = W_o h + b_o$$

Lecture agenda

01 Neurons, layers and networks

02 Capacity and activations

03 Predictions and losses

04 Learning and backpropagation

05 Training in practice

Match the output to the task

Regression

$$\widehat{y} \in \mathbb{R}$$

- Predict a numerical quantity
- Use a numerical error

Classification

$$p_k = P_\theta(y = k \mid x)$$

- Predict probabilities of categories
- Use the observed category

Recap: multiclass classification

$$s = Wx + b \in \mathbb{R}^K$$

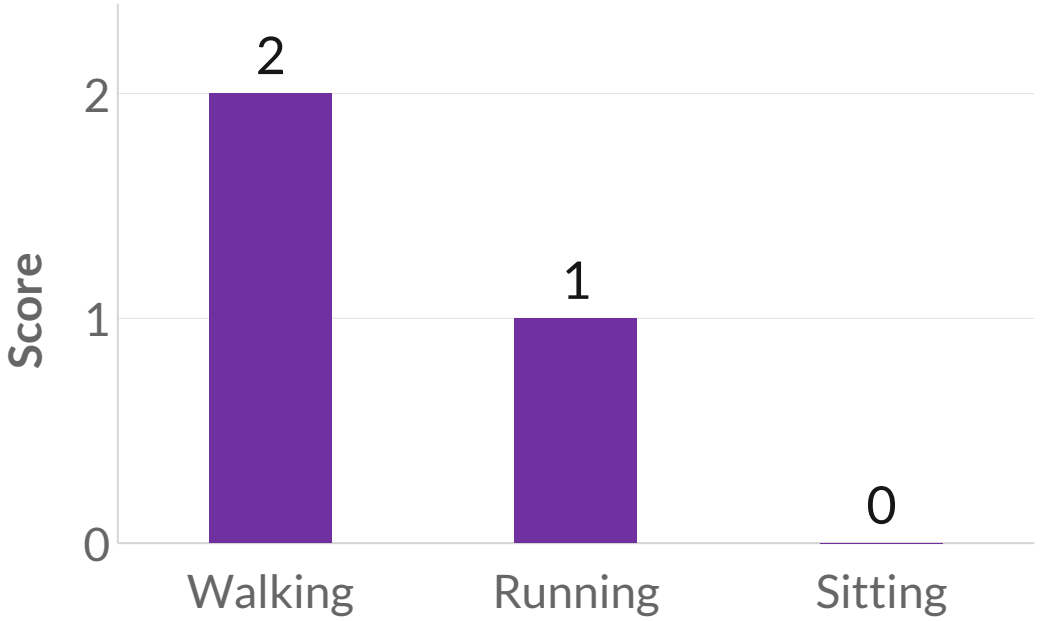
$$\widehat{y} = \arg\max_k s_k$$

$$\text{Neural classifier: } s = W_{\text{out}} h_\theta(x) + b_{\text{out}}$$

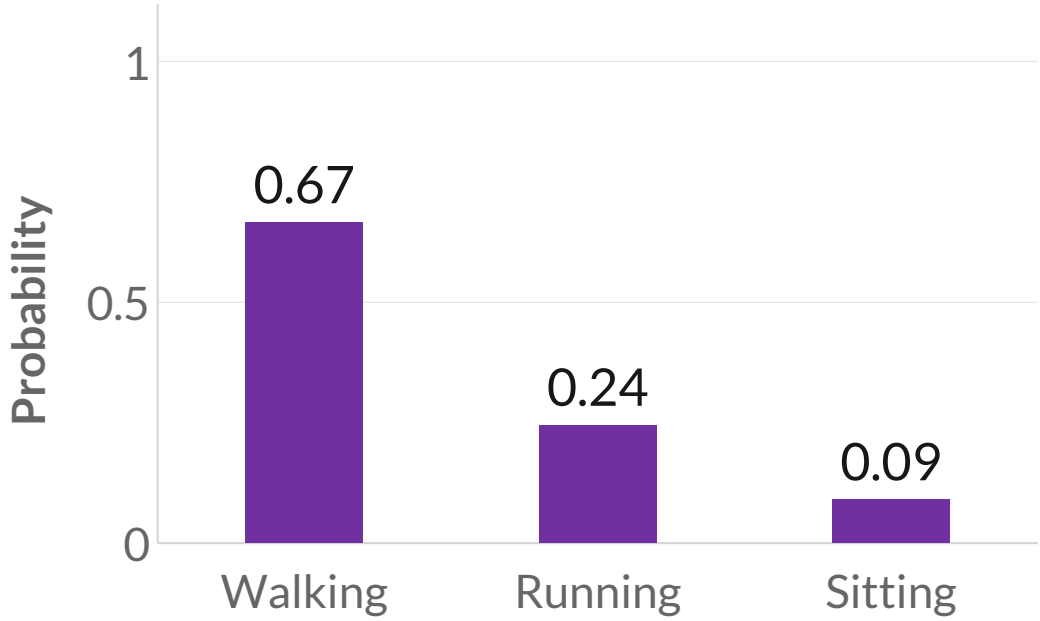
- One score for each mutually exclusive class.

Softmax converts logits into probabilities

Logits



Probabilities



$$p_k = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

Binary, multiclass and multilabel targets

Binary

One yes/no target

$$p = \sigma(s)$$

Multiclass

Exactly one of K categories

$$p = \mathrm{softmax}(s)$$

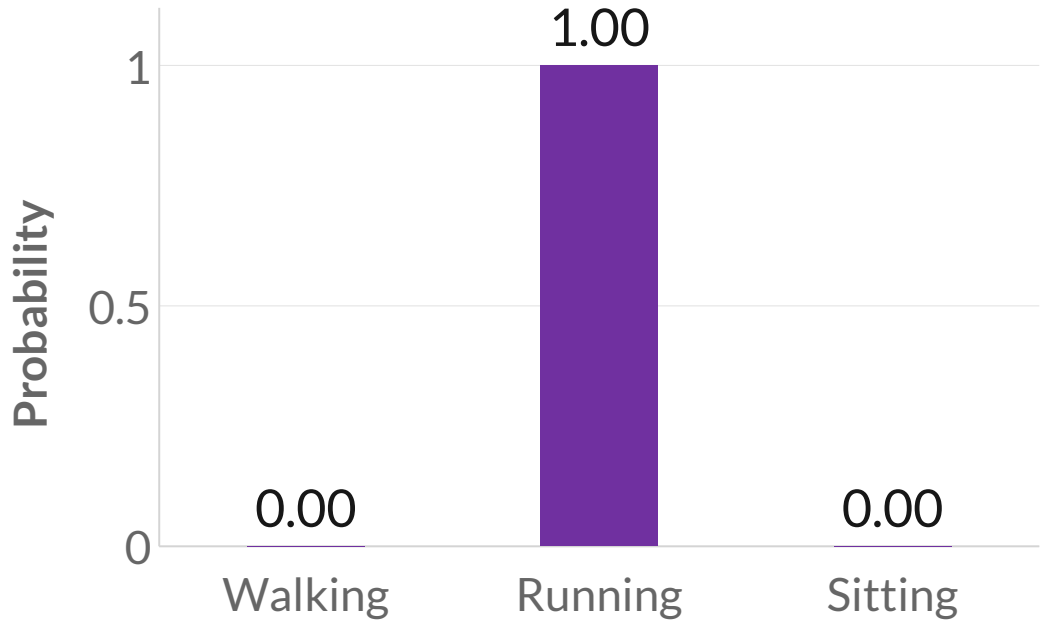
Multilabel

Several labels may apply

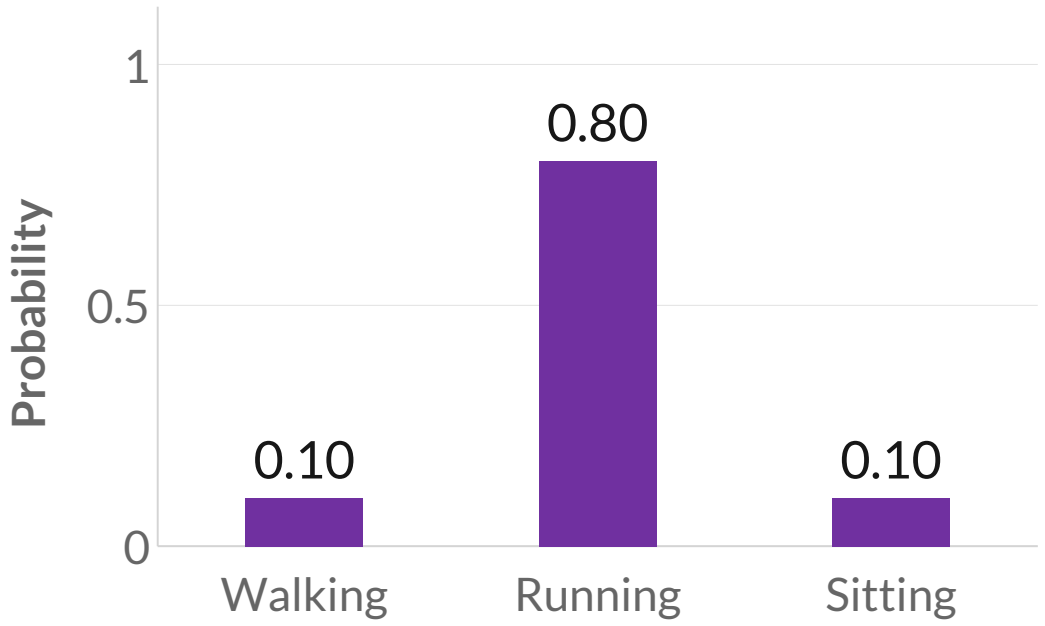
$$p_k = \sigma(s_k)$$

Labels as target distributions

One-hot target



Soft target



$$q_k = \mathbf{1}\{k = y\}, \qquad \sum_k q_k = 1$$

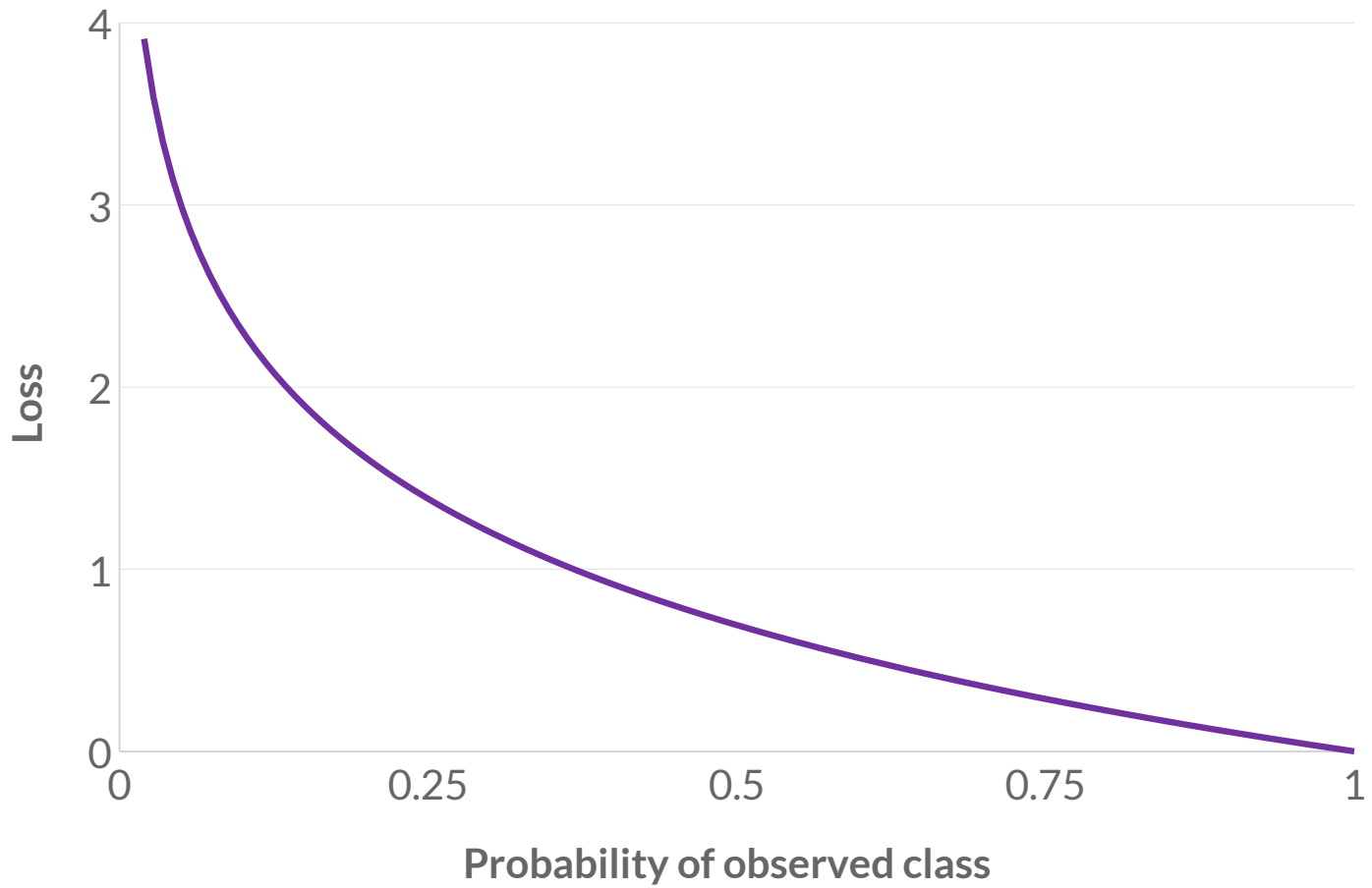
Maximum likelihood and negative log likelihood

$$\max_{\theta} \prod_{i=1}^{N} p_\theta(y_i \mid x_i)$$

$$\iff \quad \min_{\theta} -\sum_{i=1}^{N} \log p_\theta(y_i \mid x_i)$$

- Assign high probability to the observed targets.
- Logarithms turn a product into a sum.

Cross-entropy for classification



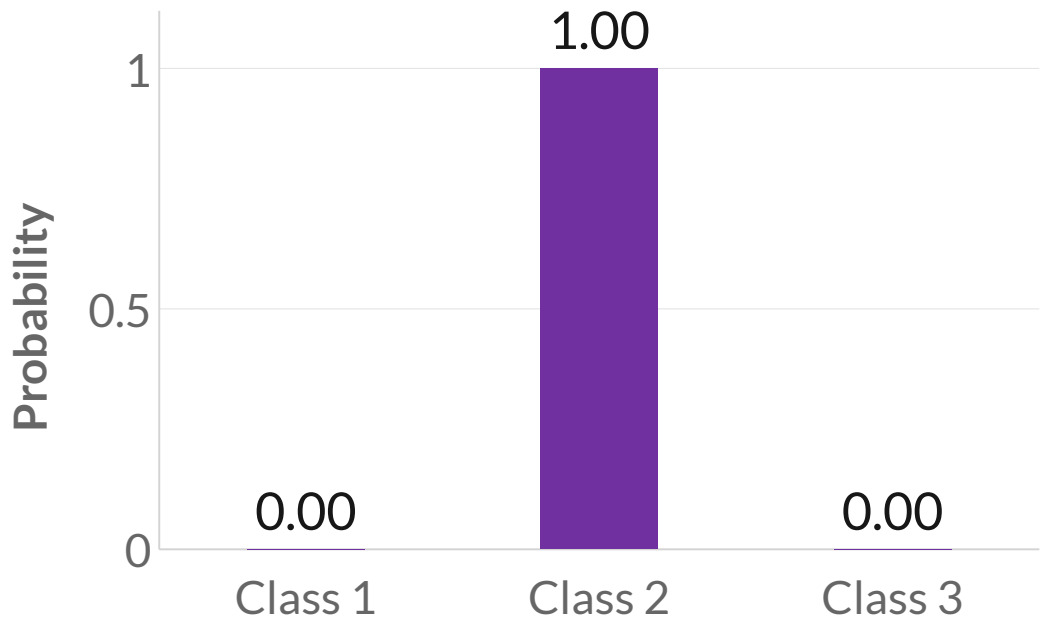
$$\ell(q, p) = -\sum_k q_k \log p_k$$

$$\ell(y, p) = -\log p_y$$

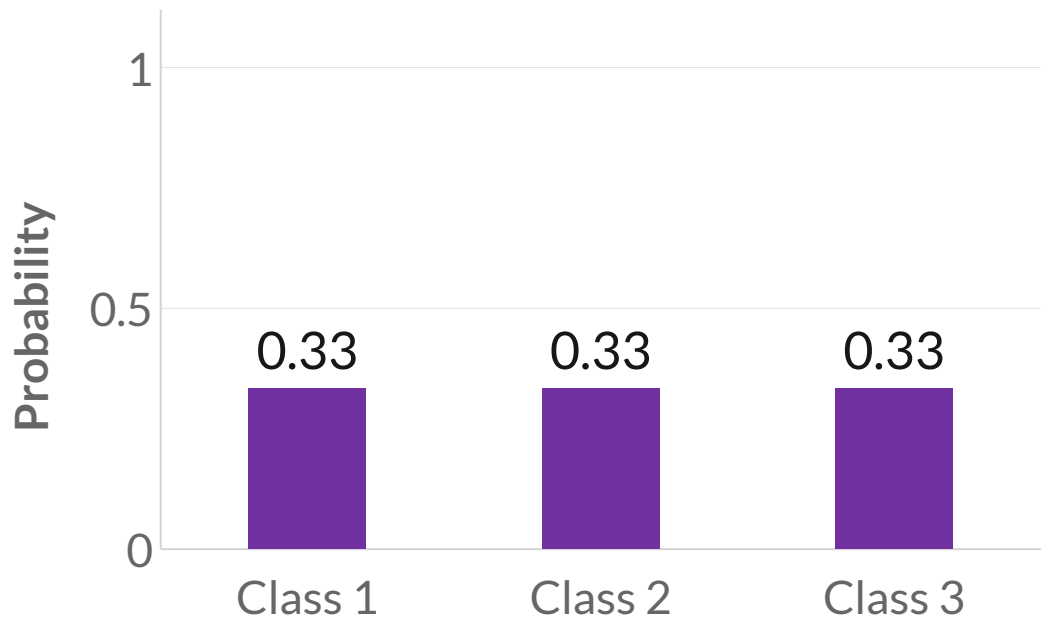
- Observed target: running
- Target probability: 0.245
- Loss ≈ 1.408

Entropy describes a distribution

Concentrated target



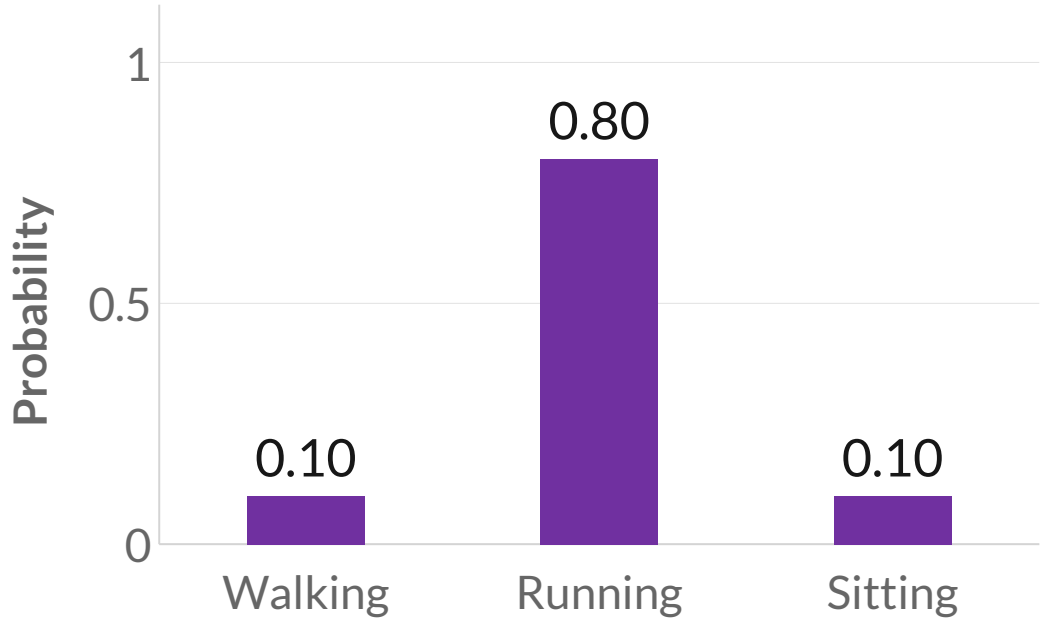
Uniform target



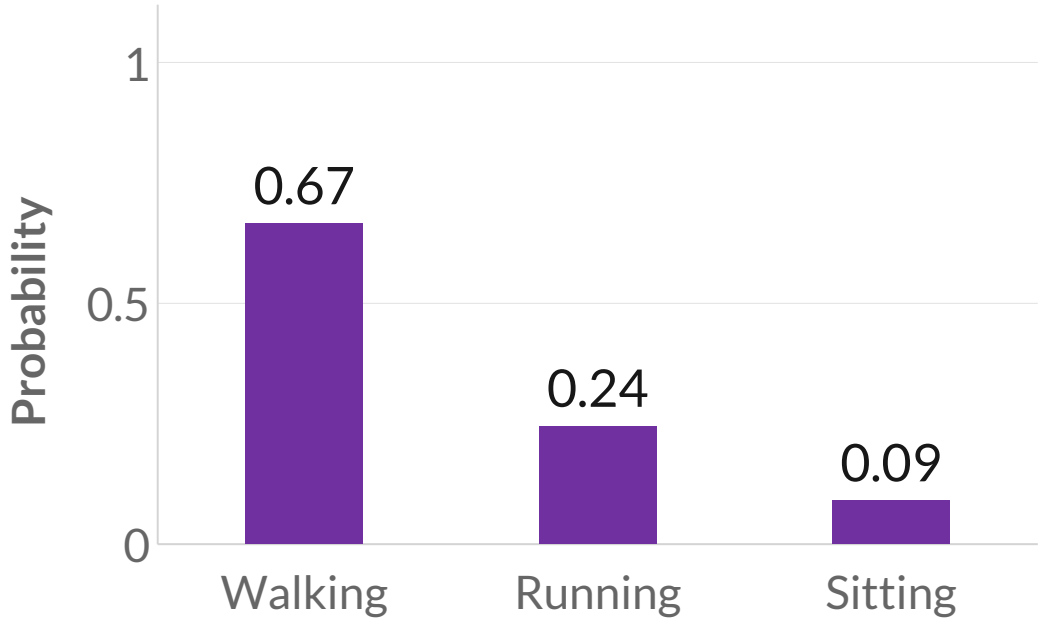
$$H(q) = -\sum_k q_k \log q_k$$

KL divergence compares distributions

Target q



Model p



$$D_{\mathrm{KL}}(q\|p) = \sum_k q_k \log \frac{q_k}{p_k}$$

Cross-entropy, entropy and KL

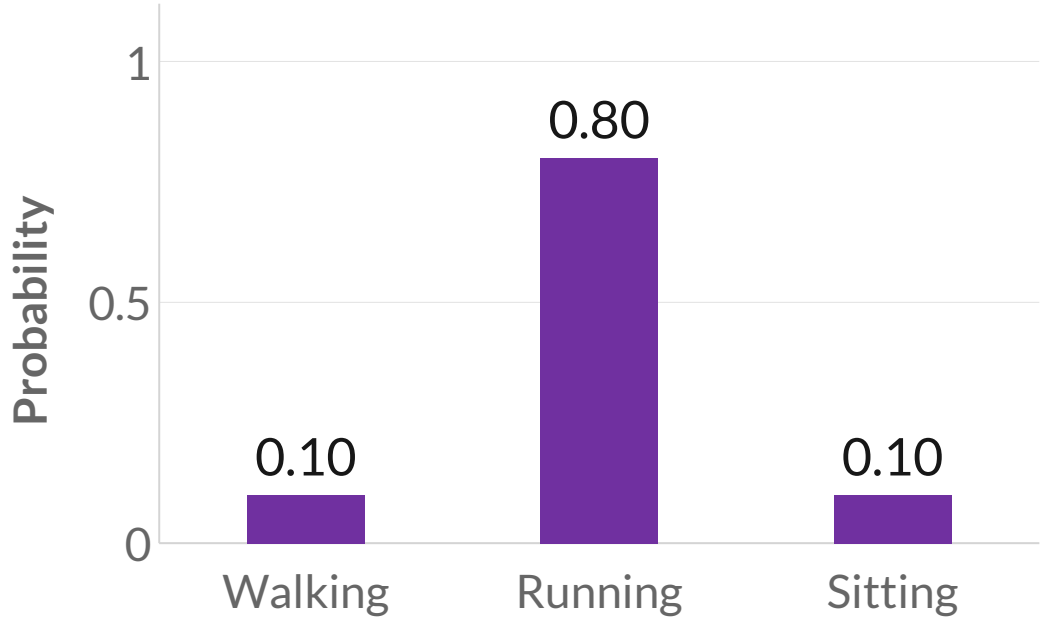
$$H(q, p) = H(q) + D_{\mathrm{KL}}(q \| p)$$

$$\min_{\theta} H(q, p_\theta) \quad \Longleftrightarrow \quad \min_{\theta} D_{\mathrm{KL}}(q \| p_\theta)$$

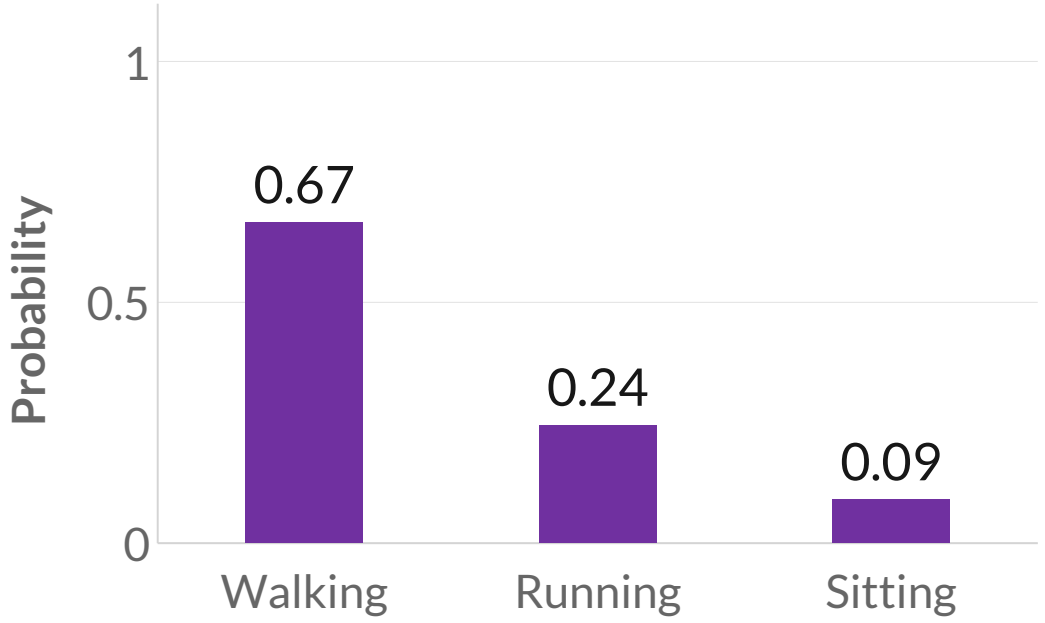
- The target entropy is constant with respect to the model.
- For one-hot targets, cross-entropy equals this KL.

KL direction

Distribution q



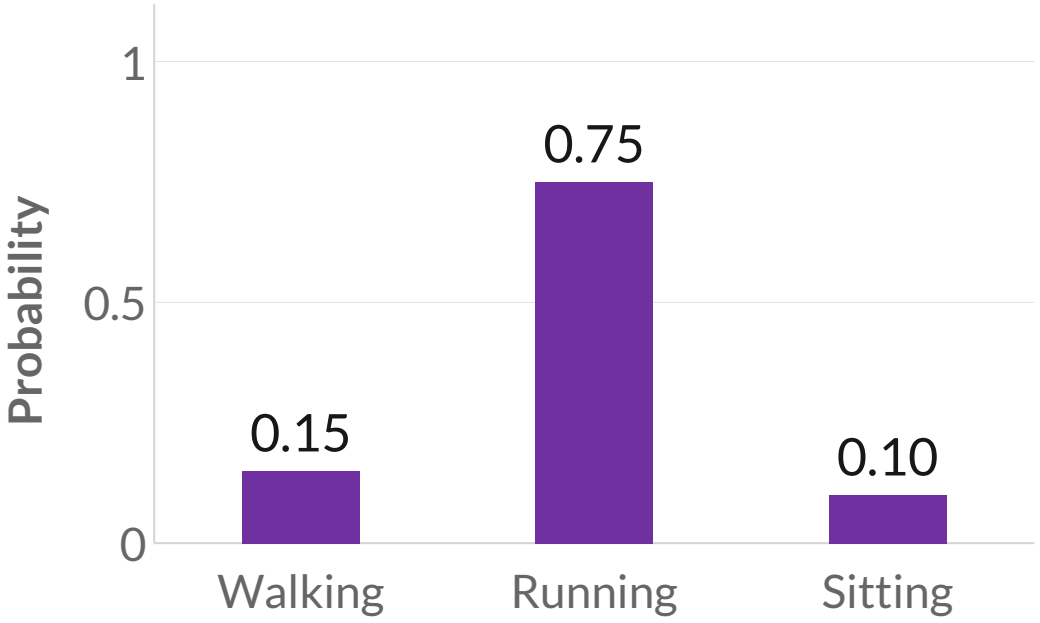
Distribution p



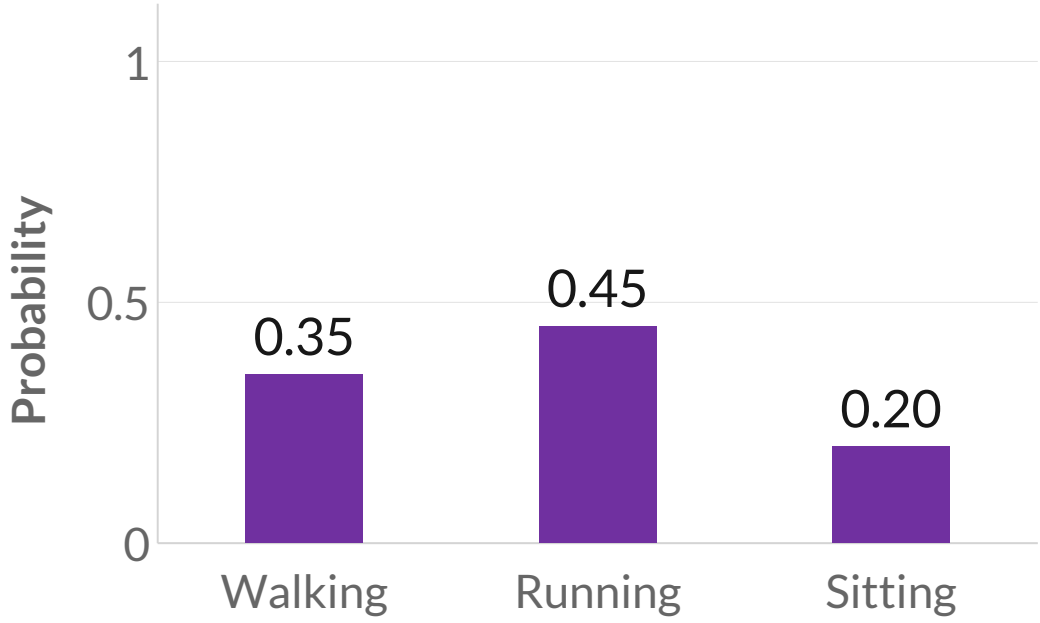
$$D_{\mathrm{KL}}(q\|p) \approx 0.769, \qquad D_{\mathrm{KL}}(p\|q) \approx 0.961$$

Soft targets and knowledge distillation

Teacher target q

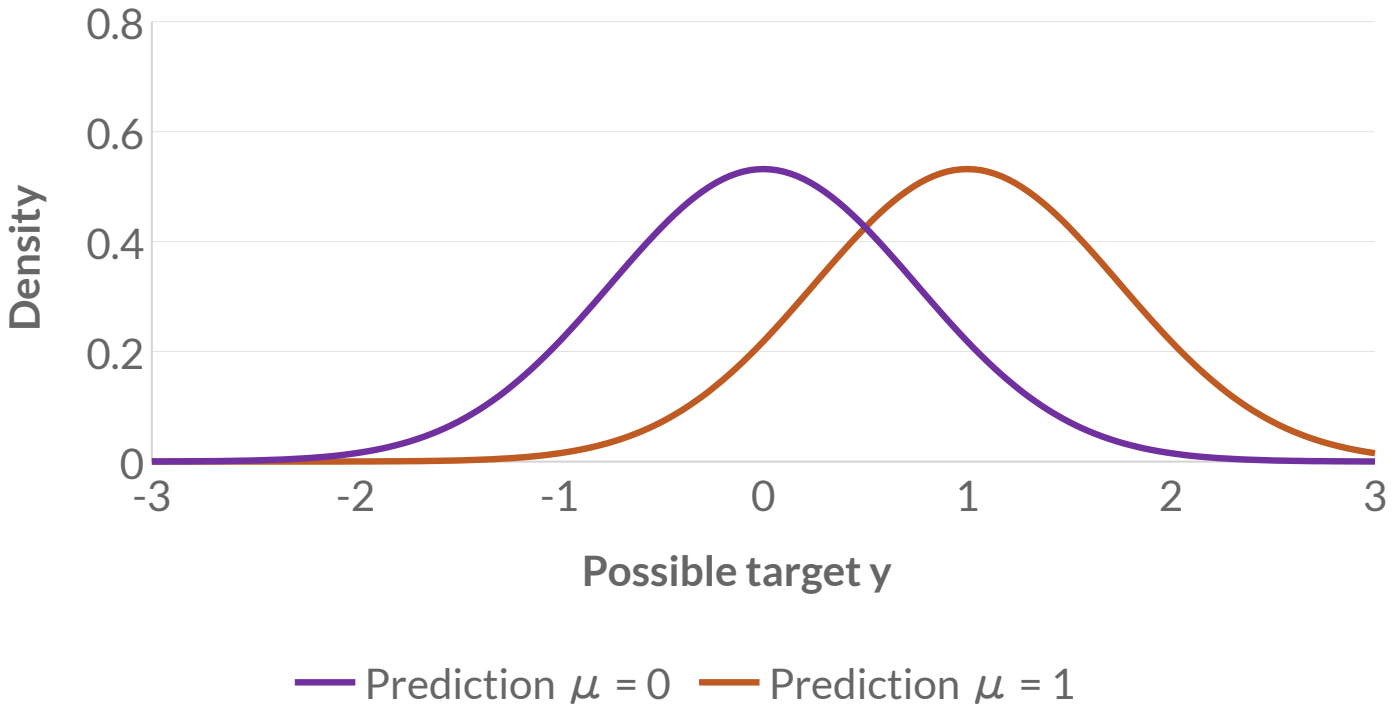


Student prediction p



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

Squared error and a Gaussian model



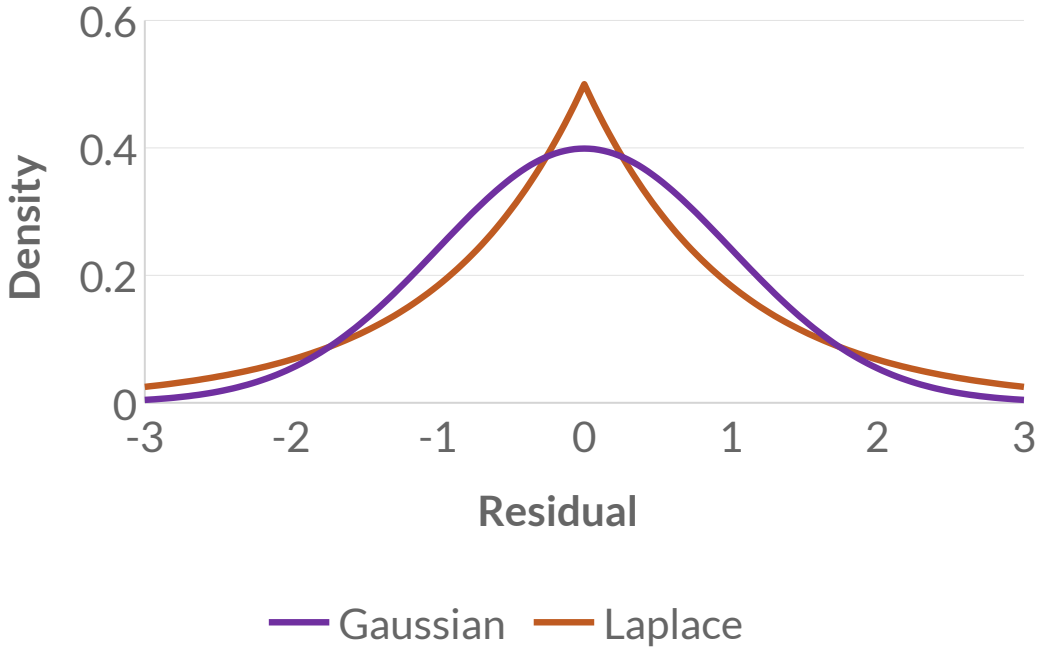
$$y \mid x \sim \mathcal{N}(\mu_\theta(x), \sigma^2)$$

- Assume a fixed variance.
- The prediction is the conditional mean.

$$-\log p_\theta(y \mid x) = \frac{(y - \mu_\theta(x))^2}{2\sigma^2} + C$$

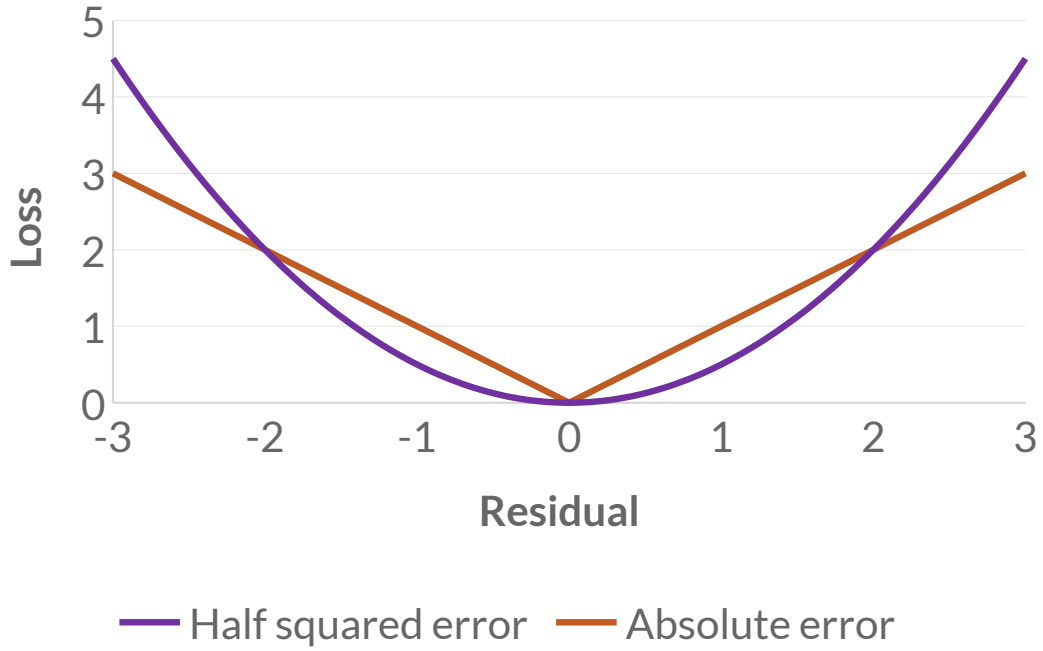
Absolute error and a Laplace model

Error distributions



$$p(y \mid \mu) = \frac{1}{2b} e^{-|y-\mu|/b}$$

Loss as the residual grows



$$-\log p(y \mid \mu) = \frac{|y-\mu|}{b} + C$$

Squared error as a distribution comparison

$$q = \mathcal{N}(y, \sigma^2), \qquad p_\theta = \mathcal{N}(\hat{y}, \sigma^2)$$

$$D_{\mathrm{KL}}(q \| p_\theta) = \frac{(y - \hat{y})^2}{2\sigma^2}$$

- Compare equal-variance Gaussian distributions.
- Many common losses have a statistical interpretation.

Compute classification losses stably

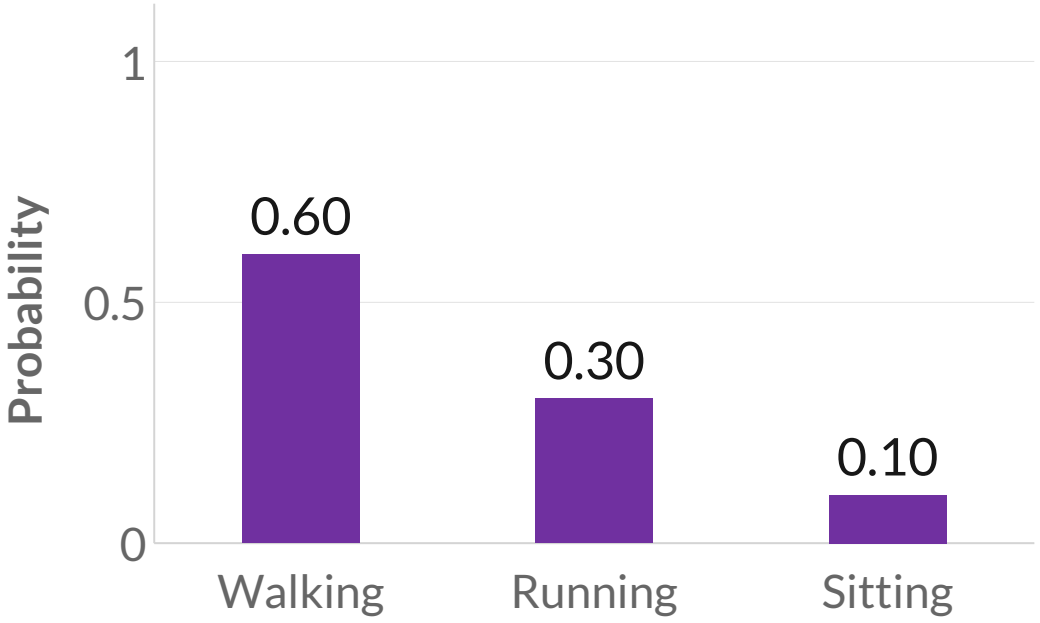
$$\mathrm{softmax}(s) = \mathrm{softmax}(s - c\mathbf{1})$$

$$\ell = -s_y + c + \log \sum_j e^{s_j - c}, \quad c = \max_j s_j$$

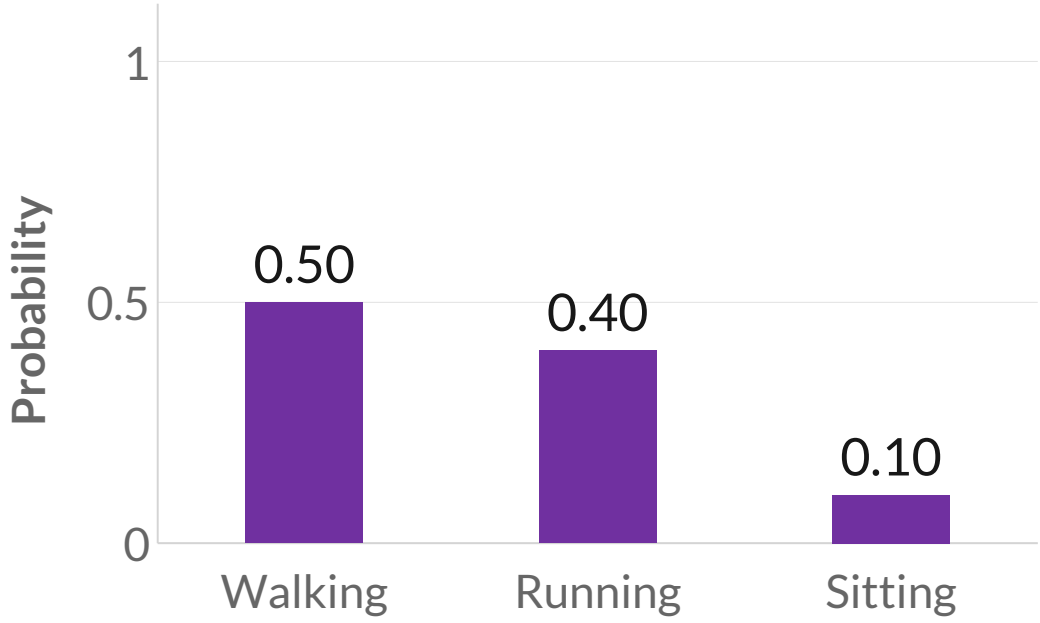
- Scores (1002, 1001, 1000) have the same probabilities as (2, 1, 0).
- Subtracting the maximum avoids huge exponentials.

Loss and accuracy measure different things

Prediction A



Prediction B



$$y = \text{running}: \quad \ell_A = 1.204, \quad \ell_B = 0.916$$

Common prediction objectives

Regression

Numerical target

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

Binary classification

Bernoulli target

$$\ell = -y \log p - (1 - y) \log(1 - p)$$

Multiclass classification

Categorical target

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

Lecture agenda

01 Neurons, layers and networks

02 Capacity and activations

03 Predictions and losses

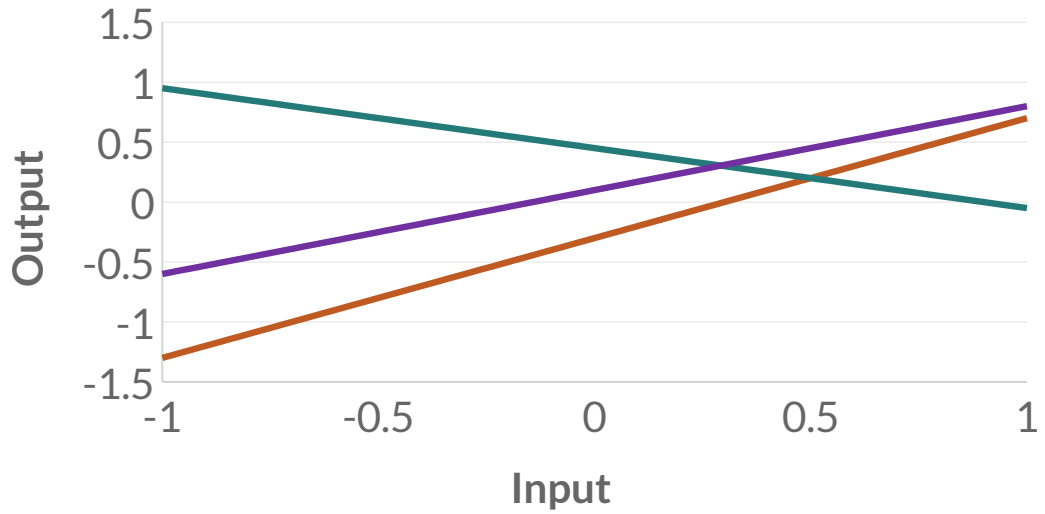
04 Learning and backpropagation

05 Training in practice

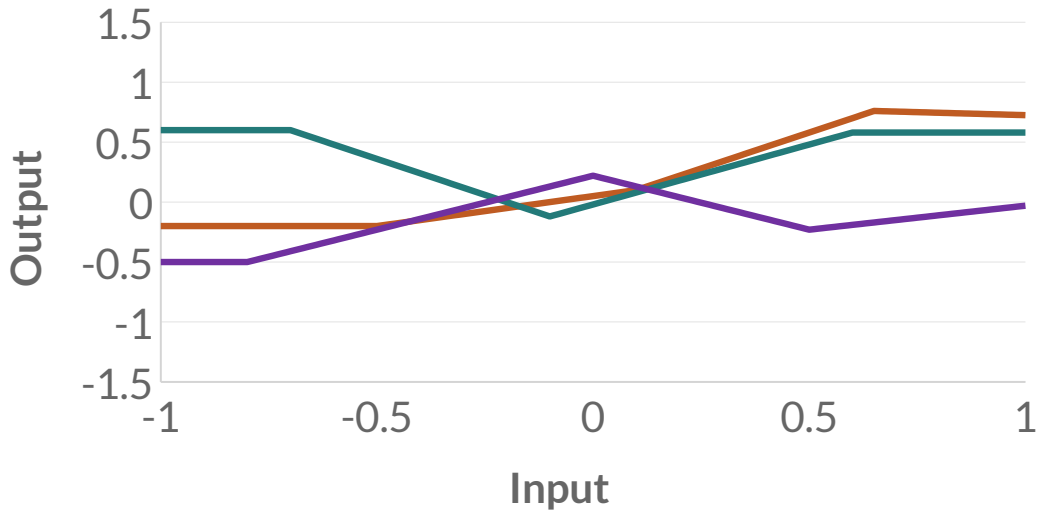
Hypothesis class

The family of prediction functions the model can represent

Linear model



MLP with 3 ReLU units



$$\mathcal{H} = \{f_\theta : \theta \in \mathbb{R}^P\}$$

Empirical loss

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

$$L_{\text{train}}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \ell(f_\theta(x_i), y_i)$$

- Empirical risk: average loss on the training data.
- Empirical: measured on the observed samples.

Empirical risk minimization

Choose a function from the hypothesis class

$$\min_{f \in \mathcal{H}} \; L_{\text{train}}(f)$$

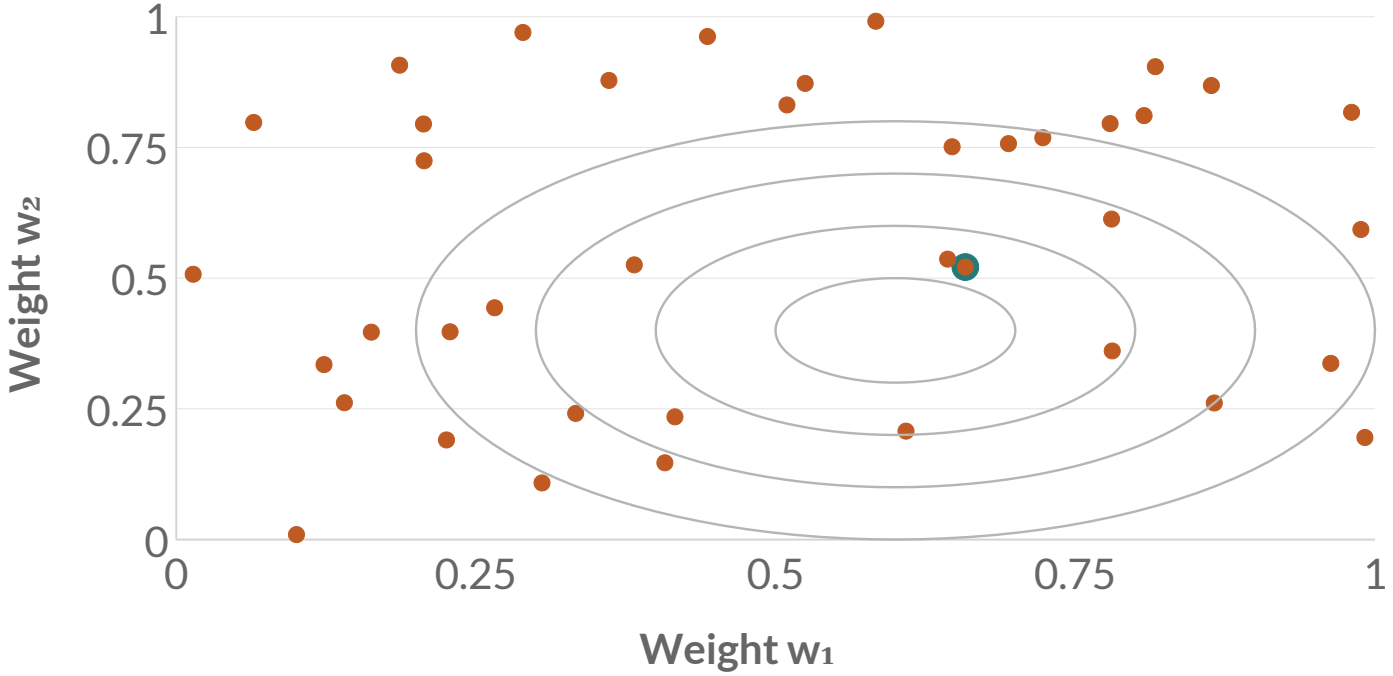
For a neural network, choose its weights and biases

$$\min_{\theta} \; L_{\text{train}}(\theta)$$

ERM defines the objective. An optimizer searches for a solution.

Random parameter guesses

Two weights sampled from [0, 1]



- Draw a complete set of weights.
- Evaluate its loss.
- Keep the best guess so far.

Orange: random guesses

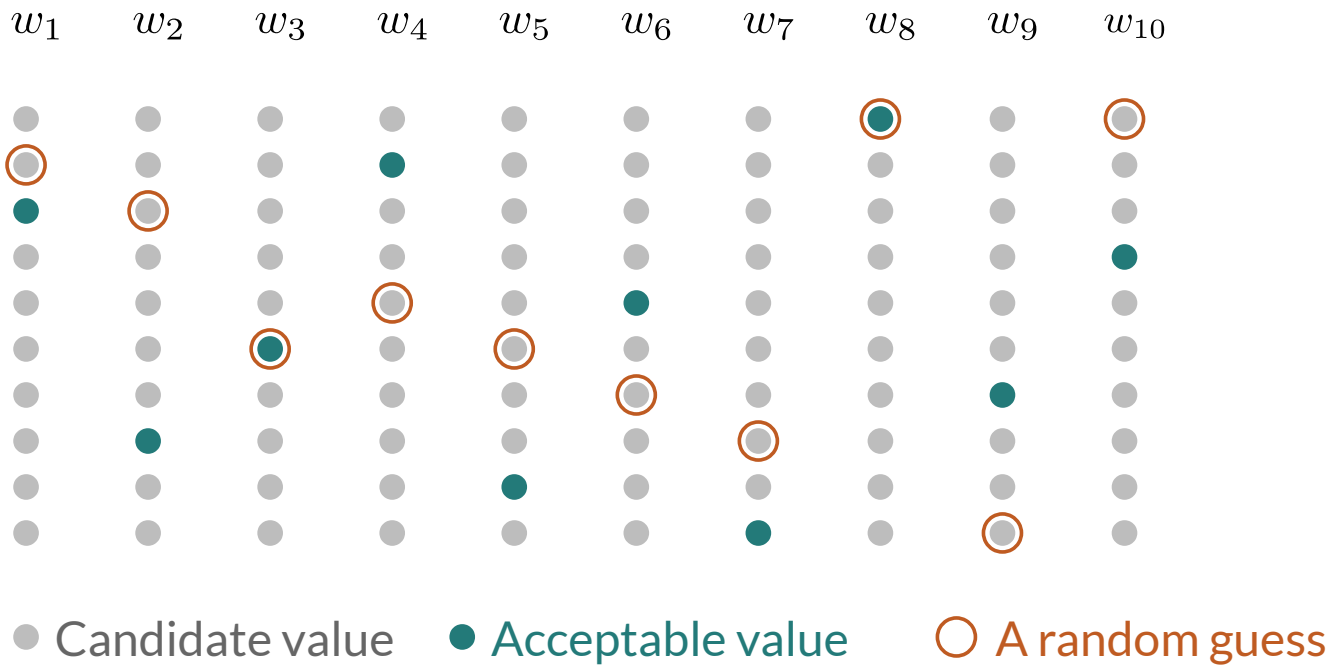
Teal: best sampled guess

$$L(w_1, w_2) = (w_1 - 0.6)^2 + (w_2 - 0.4)^2$$

Random guessing in high dimensions

Ten candidate values for each weight

Choose one value from every column.



Possible combinations

1 weight

10

2 weights

100

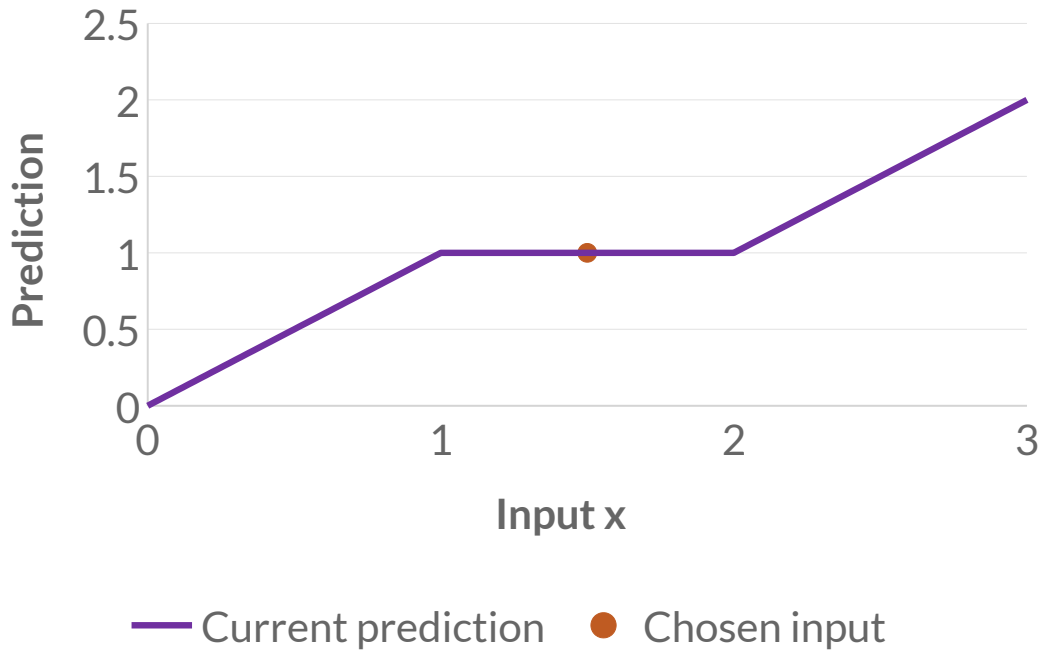
10 weights

10 billion

One acceptable combination out of 10 billion.

Parameter derivatives and input derivatives

Prediction as the input changes

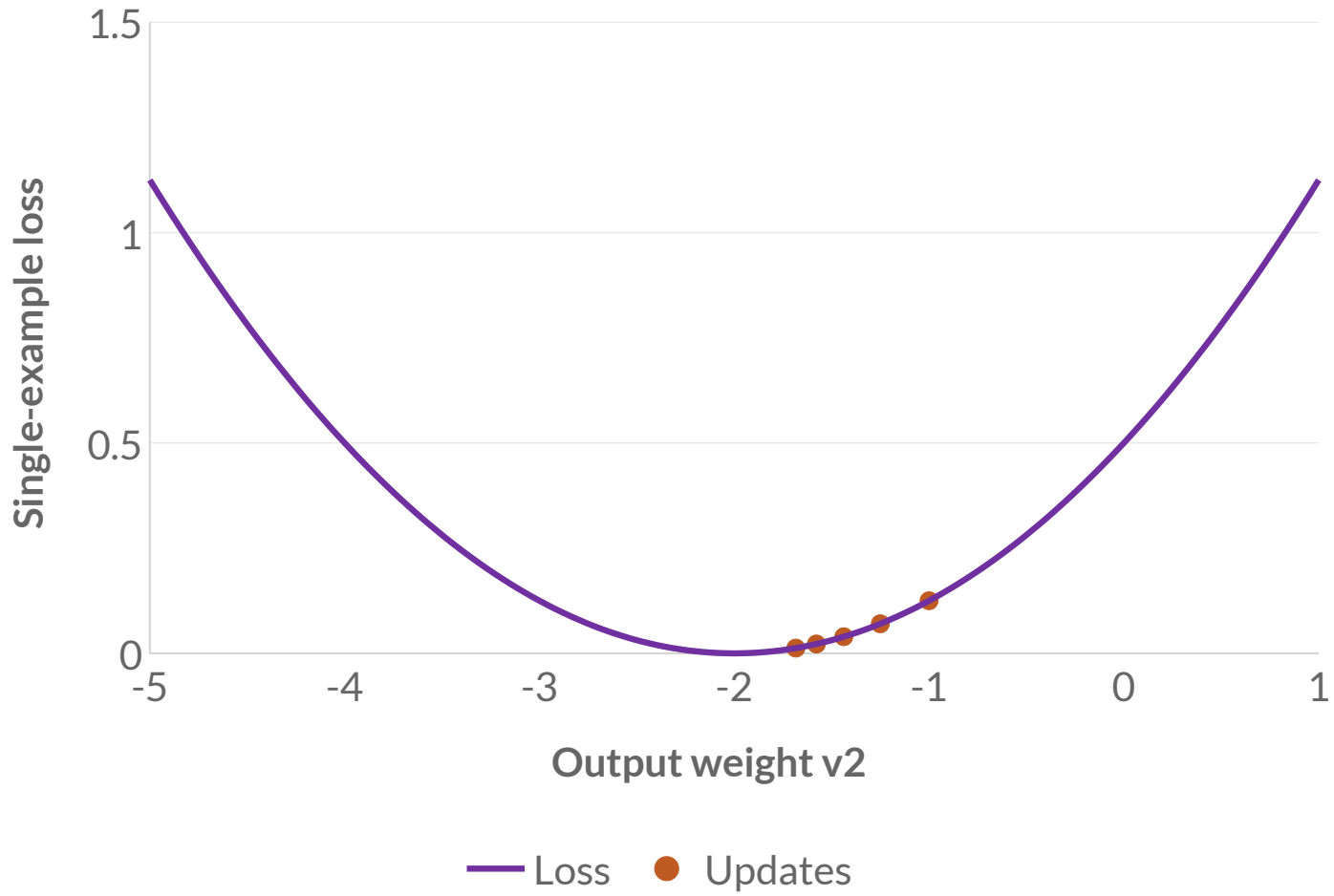


$$\frac{\partial f_\theta(x)}{\partial x}$$

Loss as one parameter changes

$$\frac{\partial \ell}{\partial v_2}$$

Gradient descent



$$\theta_{t+1} = \theta_t - \eta \nabla_\theta L(\theta_t)$$

- Use derivatives at the current parameters.
- Move opposite the local increase in loss.

Full-batch gradient descent

Algorithm 1 · Full-batch gradient descent

Require: $\text{Data } \mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}, \quad \eta > 0, \quad T \text{ updates}$

1 Initialize the model parameters θ_0

2 **for** $t = 0, \ldots, T-1$ **do**

3 $g_t \leftarrow N^{-1} \sum_{i=1}^{N} \nabla_\theta \ell(f_{\theta_t}(x_i), y_i)$

4 $\theta_{t+1} \leftarrow \theta_t - \eta g_t$

5 **end for**

6 **return** θ_T

Training choices in gradient descent

Algorithm 1 · Full-batch gradient descent

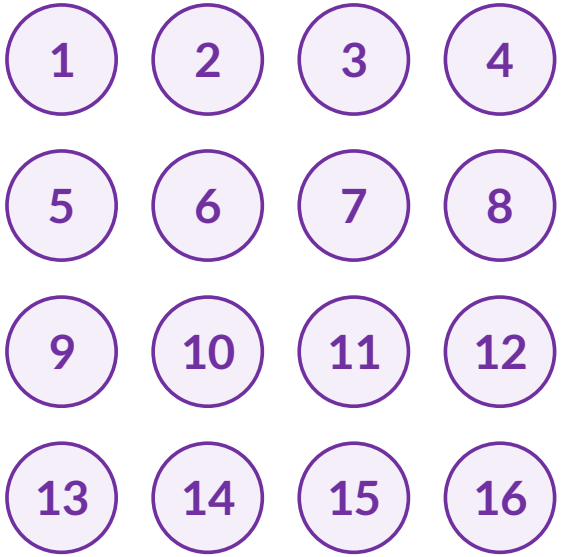
Require: Data $\mathcal{D}$, N examples, $\eta > 0$, T updates

1 Initialize the parameters θ_0
2 **for** $t = 0, \ldots, T-1$ **do**
3 $\quad \mathcal{I}_t \leftarrow \{1, \ldots, N\}$
4 $\quad g_t \leftarrow |\mathcal{I}_t|^{-1} \sum_{i \in \mathcal{I}_t} \nabla_\theta \ell(f_{\theta_t}(x_i), y_i)$
5 $\quad \eta_t \leftarrow \eta$
6 $\quad \theta_{t+1} \leftarrow \theta_t - \eta_t g_t$
7 **end for**
8 **return** θ_T

— Initialization
— Number of updates
— **Data per update**
— **Gradient calculation**
— Learning rate
— Update rule

Cost of full-batch updates

Full batch: every training example



- Every update waits for all N examples.
- Similar examples may repeat gradient information.

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

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)

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

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

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

Minibatch SGD algorithm

Algorithm 2 · Minibatch stochastic gradient descent

Require: $\text{Data } \mathcal{D}, \quad B \leq N, \quad \eta > 0, \quad T \text{ updates}$

1 Initialize the parameters θ_0 — Initialization

2 **for** $t = 0, \ldots, T-1$ **do** — Number of updates

3 $\quad \mathcal{I}_t \leftarrow \text{sample}_B\{1, \ldots, N\}$ — **Uniform minibatch**

4 $\quad g_t \leftarrow |\mathcal{I}_t|^{-1} \sum_{i \in \mathcal{I}_t} \nabla_\theta \ell(f_{\theta_t}(x_i), y_i)$ — **Mean batch gradient**

5 $\quad \eta_t \leftarrow \eta$ — Learning rate

6 $\quad \theta_{t+1} \leftarrow \theta_t - \eta_t g_t$ — Update rule

7 **end for**

8 **return** θ_T

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

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

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

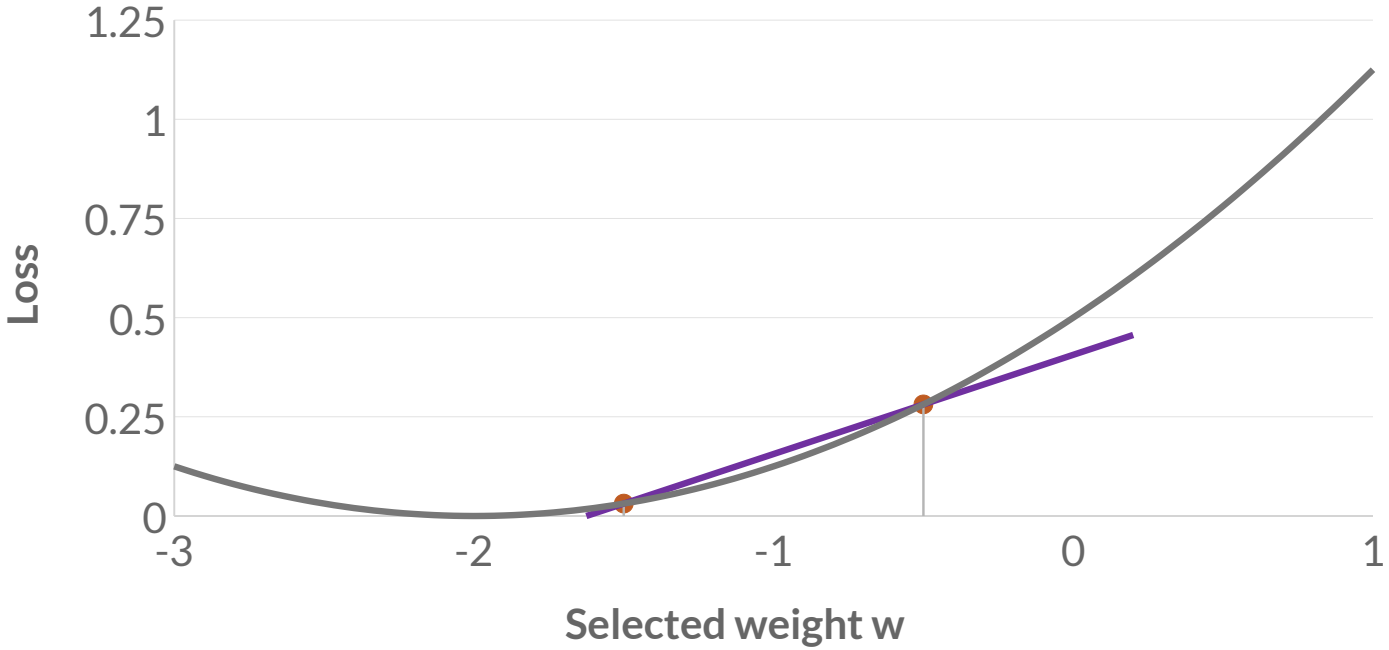
$$\ell = \tfrac{1}{2}(v^\mathsf{T} h_2 + c - y)^2$$

- Hand derivations grow with the model.

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

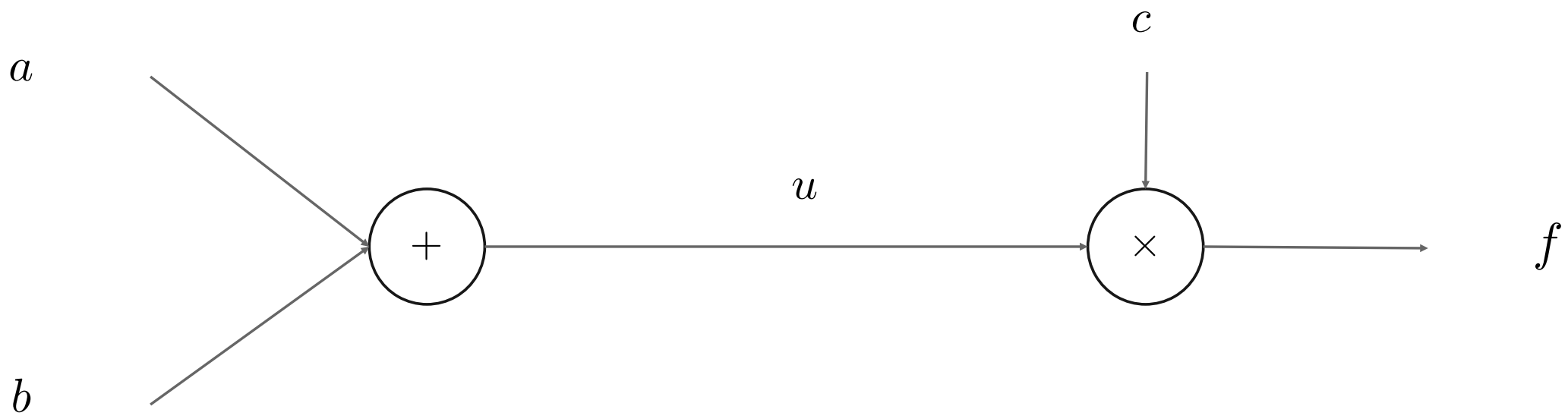
Two forward evaluations for each parameter

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

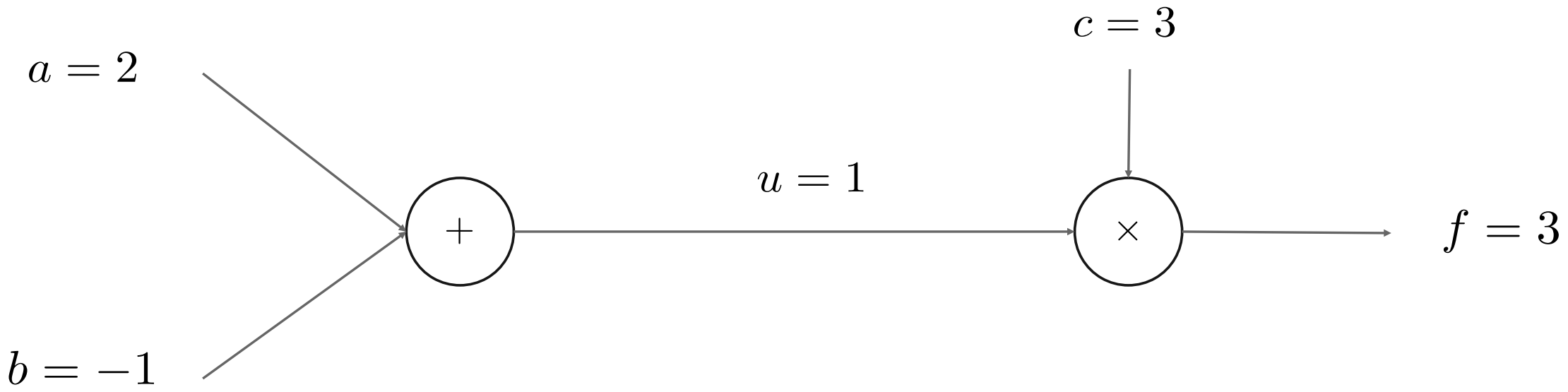
Circles are operations; arrows carry values.



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

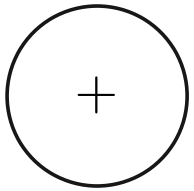
Forward evaluation

Circles are operations; arrows carry values.

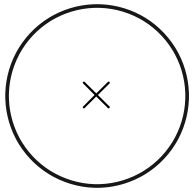


$$a = 2, \quad b = -1, \quad c = 3, \qquad u = 1, \quad f = 3$$

Local derivatives and the chain rule



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

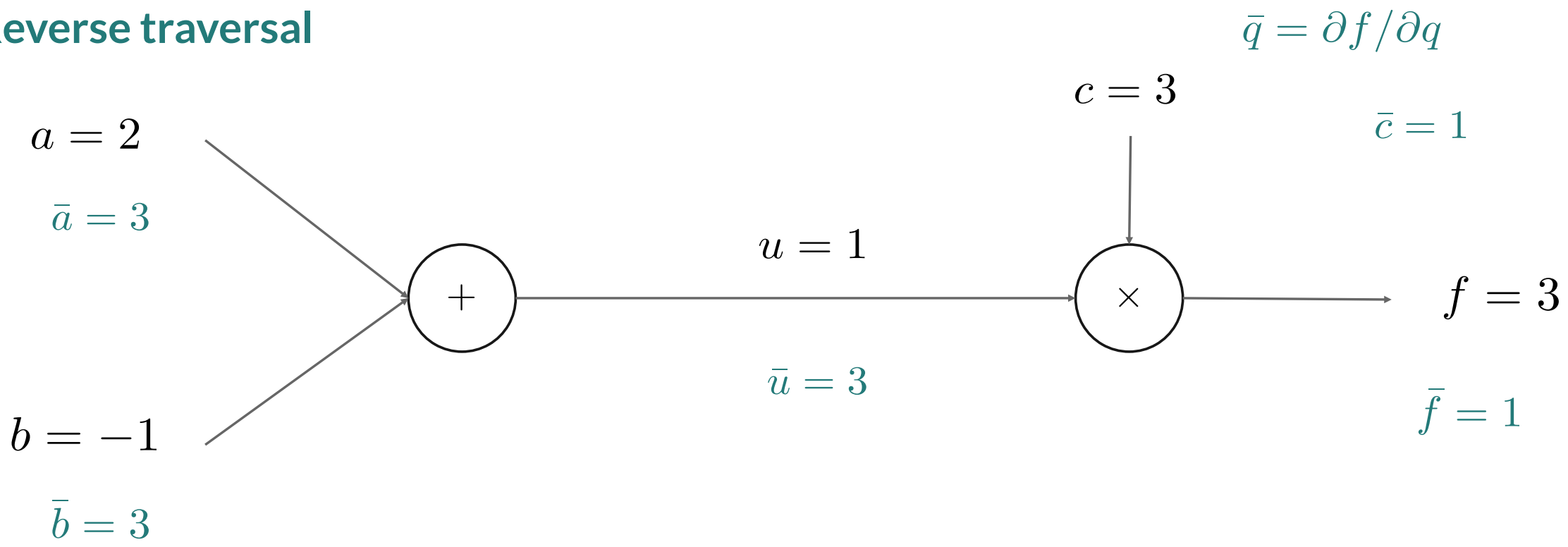


$$f = uc: \quad \frac{\partial f}{\partial u} = c, \quad \frac{\partial f}{\partial c} = u$$

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

Reverse-mode differentiation

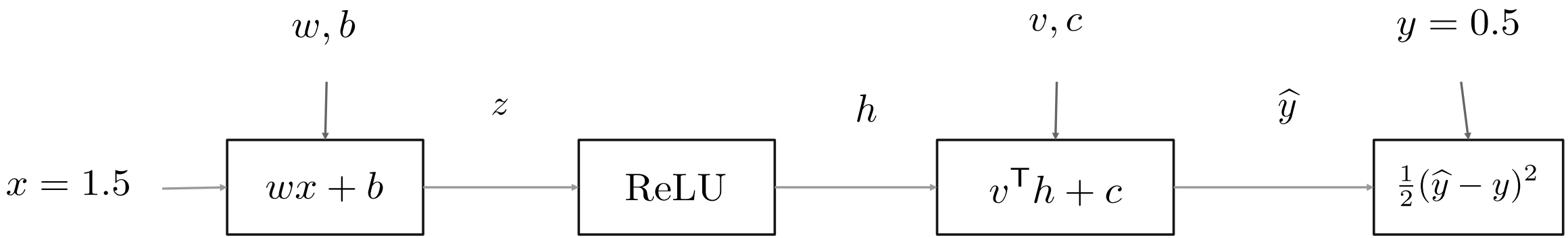
Reverse traversal



$$\left(\frac{\partial f}{\partial a}, \frac{\partial f}{\partial b}, \frac{\partial f}{\partial c}\right) = (3, 3, 1)$$

The MLP as a computation graph

A node can operate on a whole vector.



$$z = \begin{bmatrix} 1.5 \\ 0.5 \\ -0.5 \end{bmatrix} \qquad h = \begin{bmatrix} 1.5 \\ 0.5 \\ 0 \end{bmatrix} \qquad \widehat{y} = 1 \qquad \ell = 0.125$$

Forward values

Backward pass: start at the loss

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

$$\frac{\partial \ell}{\partial \widehat{y}} = \widehat{y} - y = 0.5$$

- The prediction is above the target.
- Reducing this prediction would reduce the local loss.

Backward pass: output weights

$$\widehat{y} = v^\mathsf{T} h + c, \qquad h = (1.5, 0.5, 0)^\mathsf{T}$$

$$\nabla_v \ell = (\widehat{y} - y) h = (0.75, 0.25, 0)^\mathsf{T}$$

$$\frac{\partial \ell}{\partial c} = 0.5, \qquad \frac{\partial \ell}{\partial h} = (0.5, -0.5, 0.5)^\mathsf{T}$$

Backward pass: through ReLU

$$z = (1.5, 0.5, -0.5)^\mathsf{T}$$

$$\frac{\partial \ell}{\partial h} = (0.5, -0.5, 0.5)^\mathsf{T}$$

$$\frac{\partial \ell}{\partial z} = \frac{\partial \ell}{\partial h} \odot \mathbf{1}\{z > 0\} = (0.5, -0.5, 0)^\mathsf{T}$$

- The third path is inactive for this input.

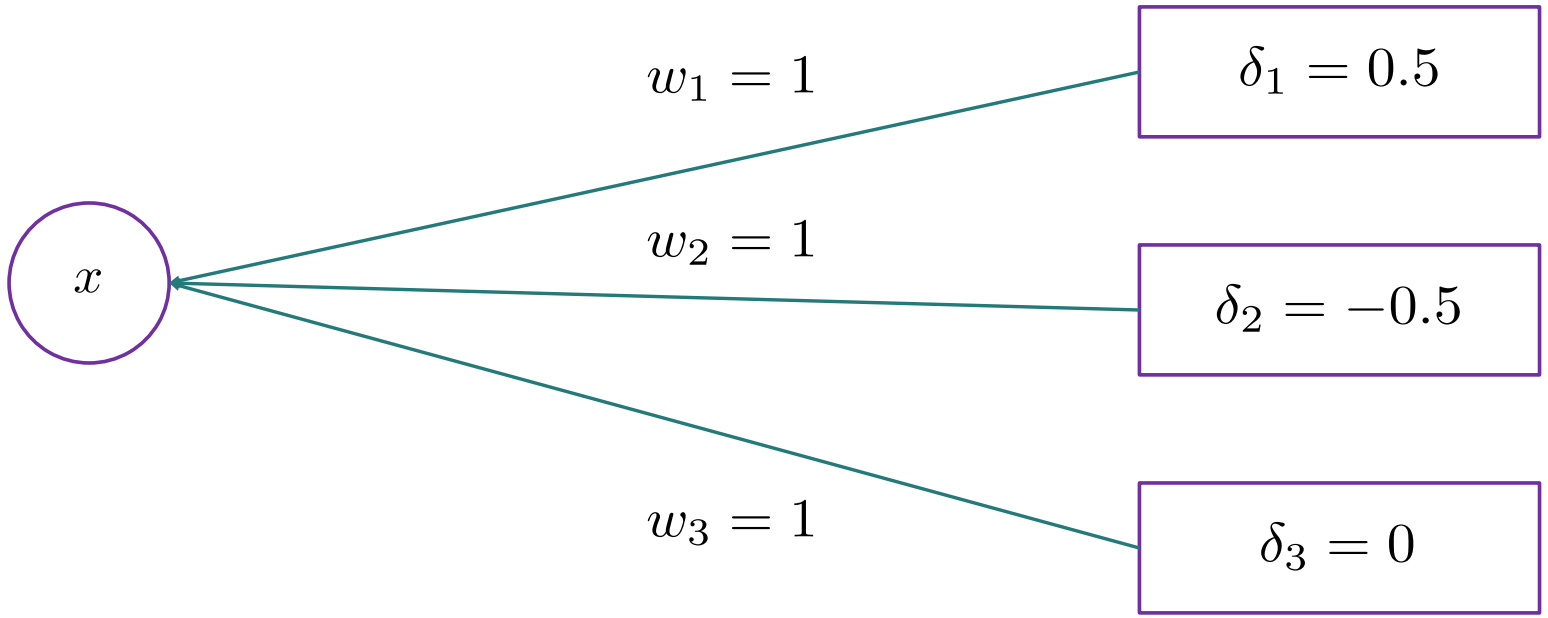
Backward pass: first-layer parameters

$$z_j = w_j x + b_j, \qquad x = 1.5$$

$$\nabla_w \ell = x \frac{\partial \ell}{\partial z} = (0.75, -0.75, 0)^\mathsf{T}$$

$$\nabla_b \ell = \frac{\partial \ell}{\partial z} = (0.5, -0.5, 0)^\mathsf{T}$$

Several paths: add their contributions



$$\delta_j = \frac{\partial \ell}{\partial z_j}$$

$$\frac{\partial \ell}{\partial x} = \sum_j \frac{\partial \ell}{\partial z_j} w_j = 0.5 - 0.5 + 0 = 0$$

Layerwise gradients in matrix form

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

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

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

- Weight gradient = output sensitivity × input activation.

The same rules for a batch

$$A \in \mathbb{R}^{d \times B}, \quad Z = WA + b\mathbf{1}_B^\mathsf{T}$$

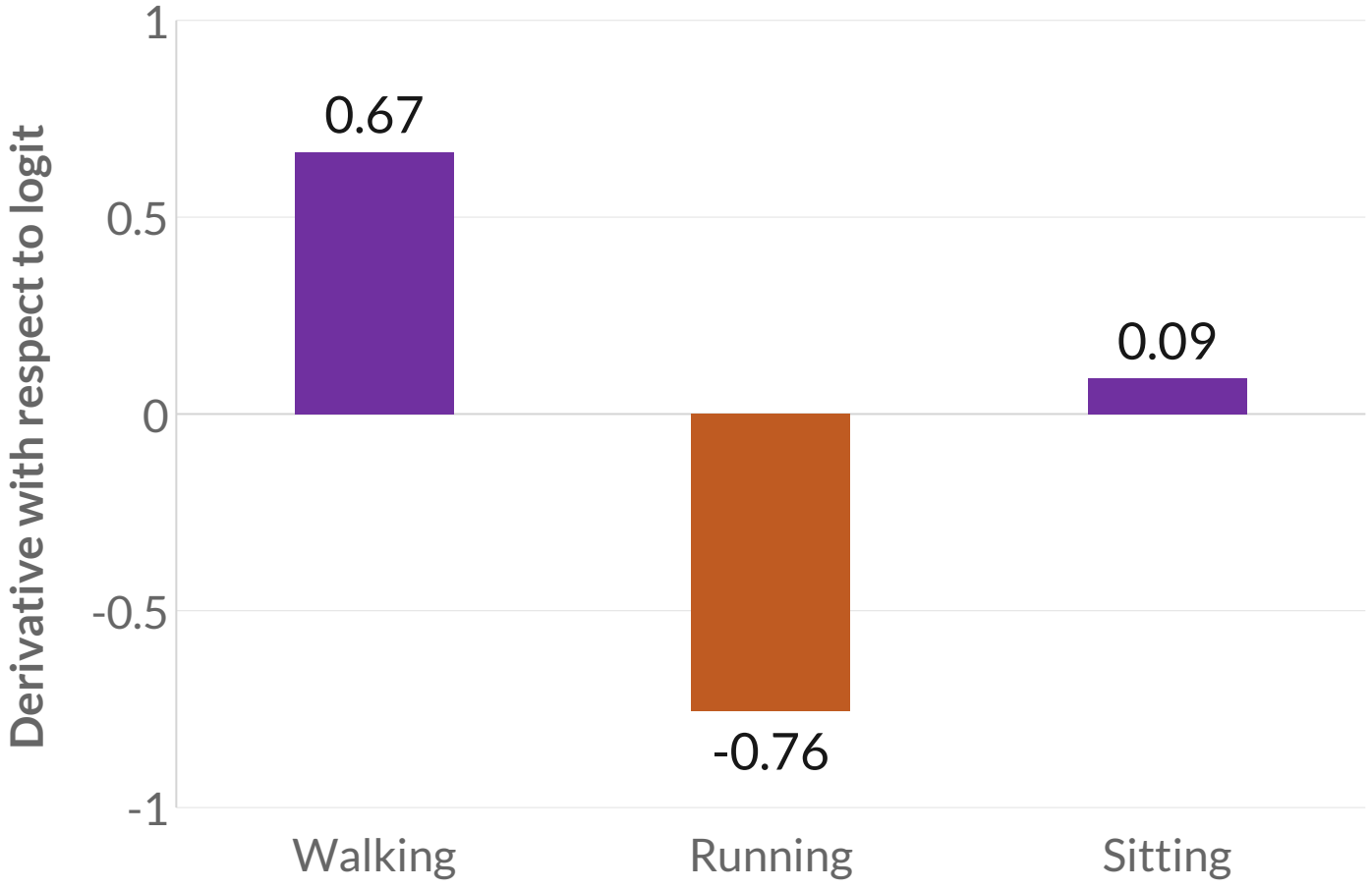
$$\Delta = \frac{\partial L_\mathcal{B}}{\partial Z} \in \mathbb{R}^{m \times B}$$

$$\nabla_W L_\mathcal{B} = \Delta A^\mathsf{T}, \quad \nabla_b L_\mathcal{B} = \Delta \mathbf{1}_B$$

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

- Δ already includes the mean-loss factor: do not divide by B again.

Classification output derivative



$$\frac{\partial \ell}{\partial s_k} = p_k - q_k$$

- Target: running
- Raising the running score locally reduces the loss.

Forward-pass algorithm

Procedure · Forward pass

Require: $X, Y, \quad \{W^{[l]}, b^{[l]}\}_{l=1}^{L}$

1 $A^{[0]} \leftarrow X$

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

3 $Z^{[l]} \leftarrow W^{[l]} A^{[l-1]} + b^{[l]} \mathbf{1}_B^{\mathsf{T}}$

4 $A^{[l]} \leftarrow \phi_l(Z^{[l]})$

5 **end for**

6 $Z^{[L]} \leftarrow W^{[L]} A^{[L-1]} + b^{[L]} \mathbf{1}_B^{\mathsf{T}}$

7 Apply the task output and evaluate the mean batch loss.

8 Retain the values needed by the backward rules.

Backpropagation algorithm

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: $\Delta^{[L]} = (P - Q)/B$

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

  $G_W^{[l]} \leftarrow \Delta^{[l]} (A^{[l-1]})^\mathsf{T}, \qquad G_b^{[l]} \leftarrow \Delta^{[l]} \mathbf{1}_B$

  if $l > 1$ **then**

    $\Delta^{[l-1]} \leftarrow ((W^{[l]})^\mathsf{T} \Delta^{[l]}) \odot \phi'_{l-1}(Z^{[l-1]})$

  end if

end for

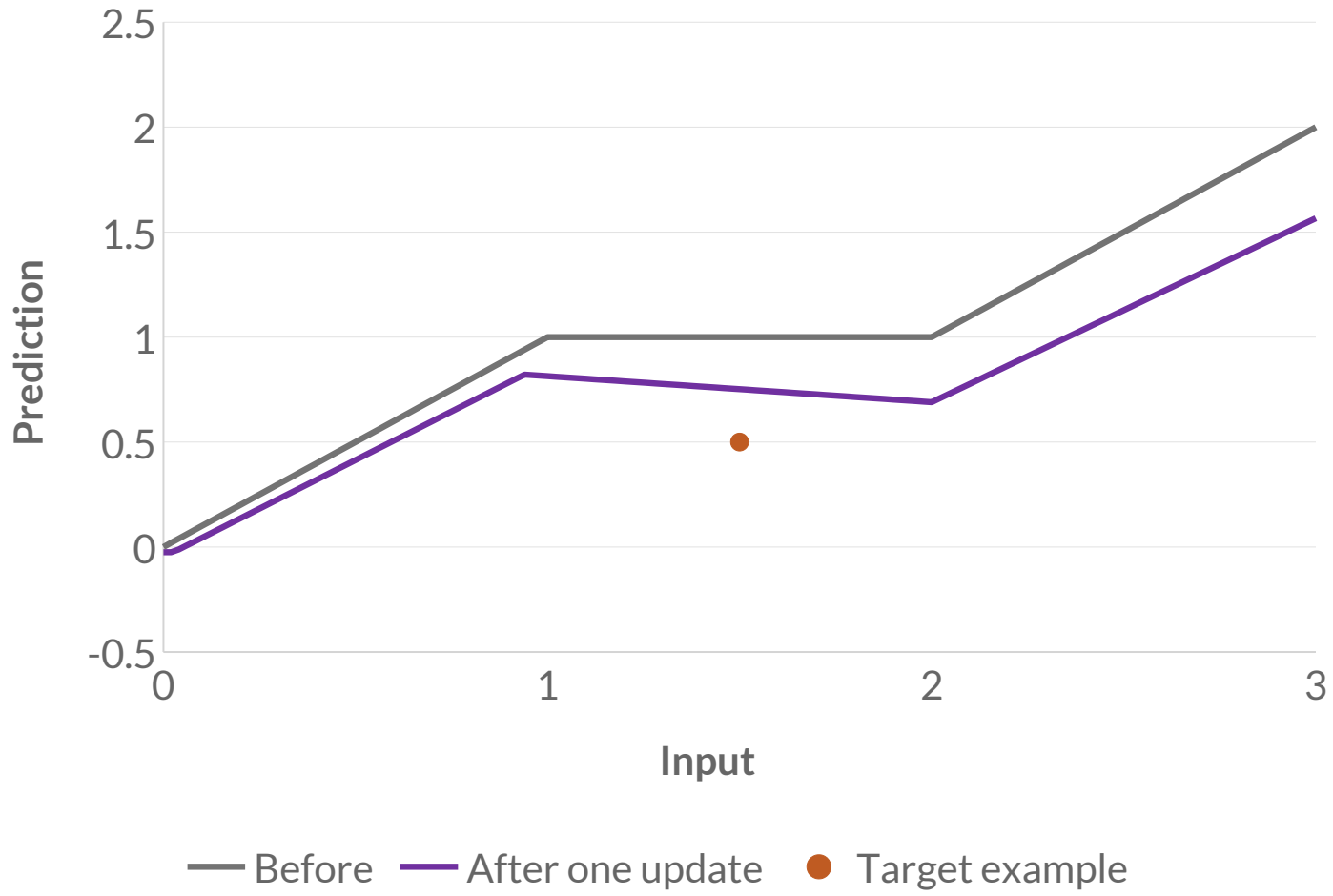
MLP training algorithm

Algorithm 3 · MLP training with minibatch SGD

Require: Data $\mathcal{D}$, batch size B, learning rate η

1 Initialize all weights and biases.
2 **repeat**
3 **for each minibatch X, Y do**
4 Forward pass: compute hidden activations and task outputs.
5 $L_{\mathcal{B}} \leftarrow (\ell_1 + \cdots + \ell_B)/B$
6 Backward pass: compute every weight and bias gradient.
7 $W^{[l]} \leftarrow W^{[l]} - \eta G_W^{[l]}, \quad b^{[l]} \leftarrow b^{[l]} - \eta G_b^{[l]}$ for all l
8 **end for**
9 **until the stopping rule is met**

One SGD step



$$\theta\prime = \theta - 0.05\nabla_\theta \ell$$

$$\widehat{y} : 1 \longrightarrow 0.7520$$

$$\ell : 0.125 \longrightarrow 0.03176$$

- All ten parameters use gradients from the same forward pass.

Automatic differentiation

$$u = f(v), \qquad J_{ij} = \frac{\partial u_i}{\partial v_j}$$

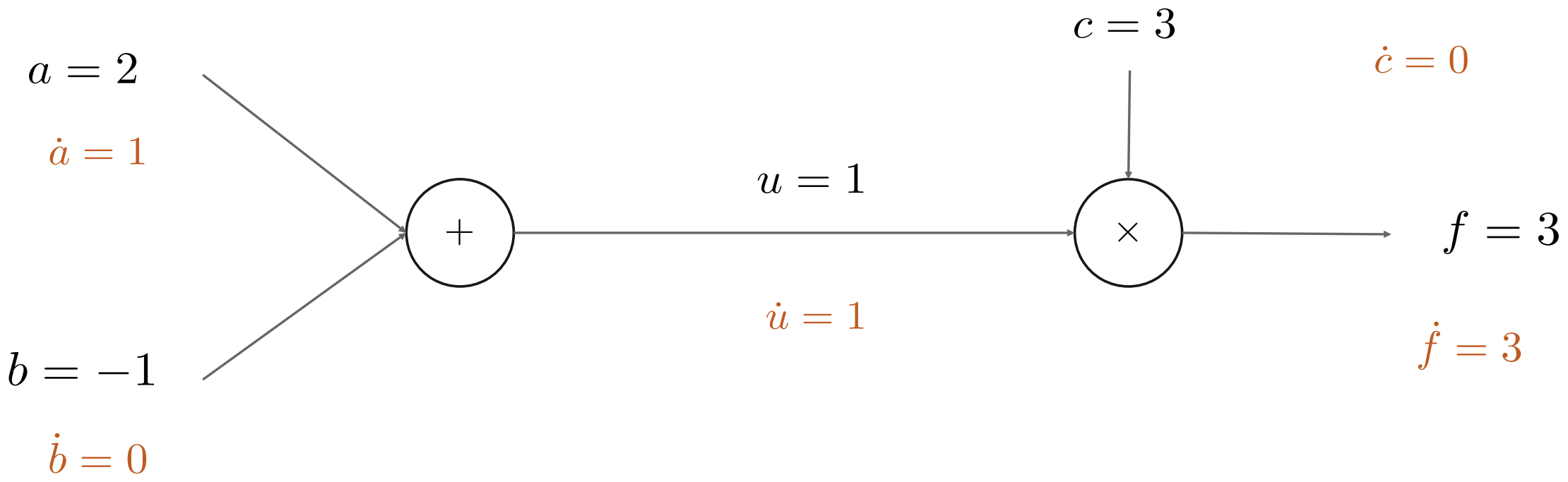
$$\nabla_v \ell = J^\mathsf{T} \nabla_u \ell$$

- Reverse mode applies local derivative rules.
- The full Jacobian need not be constructed.

Forward-mode differentiation

Differentiate with respect to a

$\dot{q} = \partial q / \partial a$



$$\dot{a} = 1, \quad \dot{b} = 0, \quad \dot{c} = 0$$

$$\dot{u} = \dot{a} + \dot{b} = 1, \qquad \dot{f} = c\dot{u} + u\dot{c} = 3$$

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}			

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

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					

$$v = e_2 : \quad Jv = \text{column } 2$$

One chosen input direction

$$u = e_2 : \quad J^\mathsf{T} u = \text{row } 2^\mathsf{T}$$

One chosen output seed

$$L : \mathbb{R}^P \to \mathbb{R} \quad \Rightarrow \quad \nabla_\theta L \text{ from one reverse seed}$$

Tensors and gradients in PyTorch

```
import torch

x = torch.tensor([[1.5]])
w = torch.tensor([[2.0]],
                 requires_grad=True)

y = x @ w
y.sum().backward()
print(w.grad)  # tensor([[1.5000]])
```

- A tensor stores numerical data.
- requires_grad enables gradients.
- Compute gradients

Autodiff and the worked example

```
w = torch.ones(3, requires_grad=True)
b = torch.tensor([0., -1., -2.], requires_grad=True)
v = torch.tensor([1., -1., 1.], requires_grad=True)
c = torch.tensor(0., requires_grad=True)

h = torch.relu(w * 1.5 + b)
loss = 0.5 * (v @ h + c - 0.5) ** 2
loss.backward()
```

$$\nabla_w \ell = (0.75, -0.75, 0)^\mathsf{T}$$

$$\nabla_v \ell = (0.75, 0.25, 0)^\mathsf{T}$$

Checking a derivative numerically

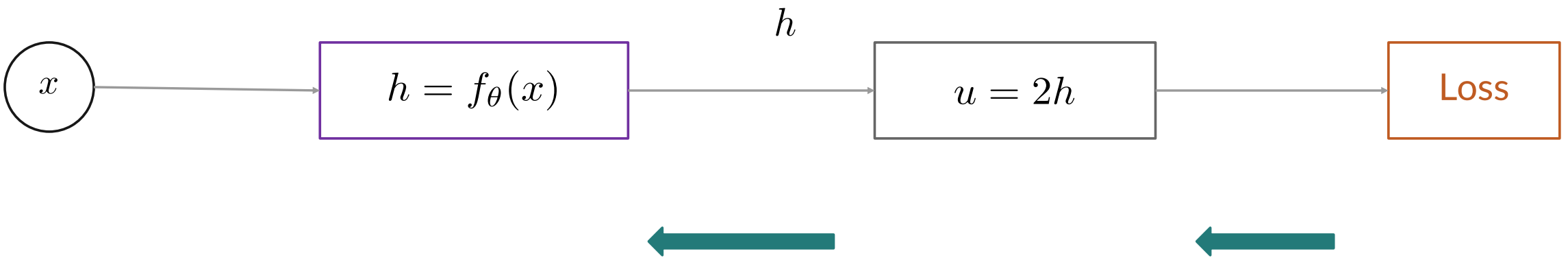
$$F(v_2) = \tfrac{1}{2}(1 + 0.5v_2)^2, \quad v_2 = -1, \quad \varepsilon = 0.01$$

$$F(-0.99) = 0.1275125 \qquad F(-1.01) = 0.1225125$$

$$\frac{0.1275125 - 0.1225125}{0.02} = 0.25$$

From backpropagation: $\dfrac{\partial \ell}{\partial v_2} = 0.25$

Gradients through fixed functions



The fixed function still has an input derivative

$$\frac{\partial \ell}{\partial \theta} = \frac{\partial \ell}{\partial u} 2 \frac{\partial h}{\partial \theta}$$

- The coefficient 2 is fixed; the input h can change.

Saved forward values

Affine layer

$$z = Wa + b$$

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

- Weight gradients require the input.

ReLU

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

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

- Retain the active/inactive mask.

Lecture agenda

01 Neurons, layers and networks

02 Capacity and activations

03 Predictions and losses

04 Learning and backpropagation

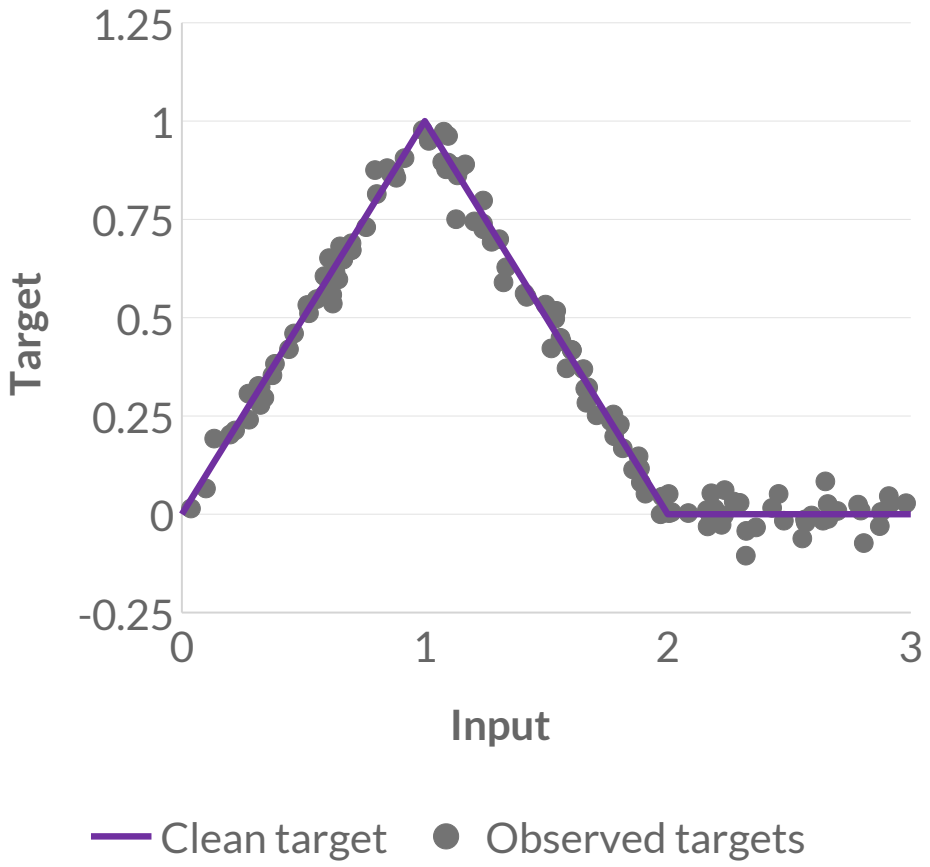
05 Training in practice

Regression example

```
def target(x):
    return (torch.relu(x)
            - 2 * torch.relu(x - 1)
            + torch.relu(x - 2))

x_train = 3 * torch.rand(128, 1)
y_train = target(x_train)
noise = torch.randn_like(x_train)
y_train += 0.04 * noise
```

- 128 inputs on [0, 3]
- Small additive noise



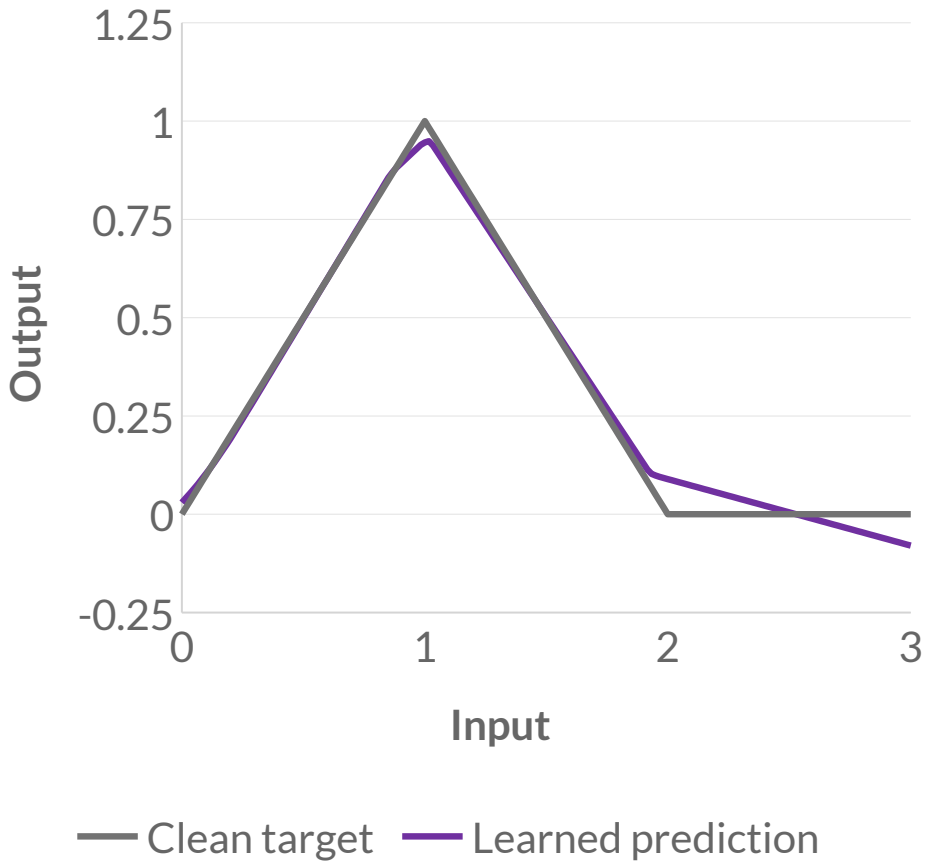
Training loop

```
from torch.utils.data import DataLoader, TensorDataset
train_data = TensorDataset(x_train, y_train)
loader = DataLoader(train_data, batch_size=32, shuffle=True)
criterion = nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.05)

for epoch in range(400):
    model.train()
    for xb, yb in loader:
        optimizer.zero_grad(set_to_none=True)
        prediction = model(xb)
        loss = criterion(prediction, yb)
        loss.backward()
        optimizer.step()
```

Model evaluation

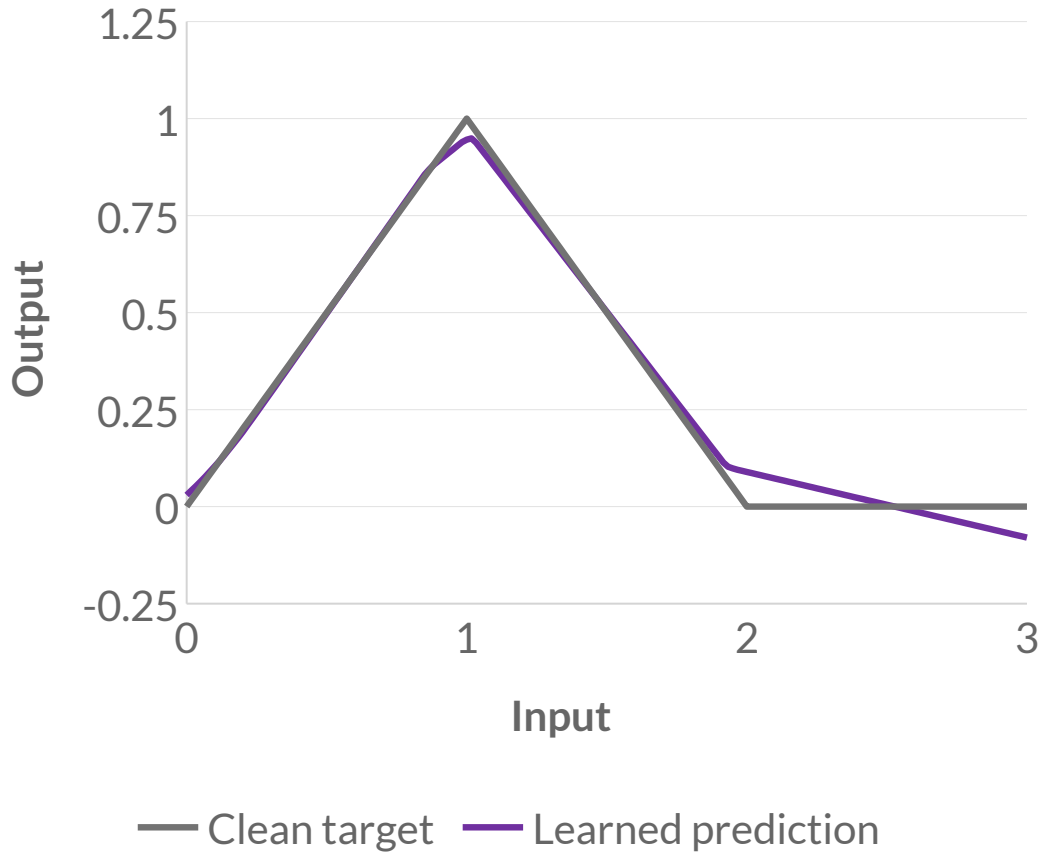
```
model.eval()
with torch.no_grad():
    prediction = model(x_val)
    val_loss = criterion(
        prediction, y_val
    )
```



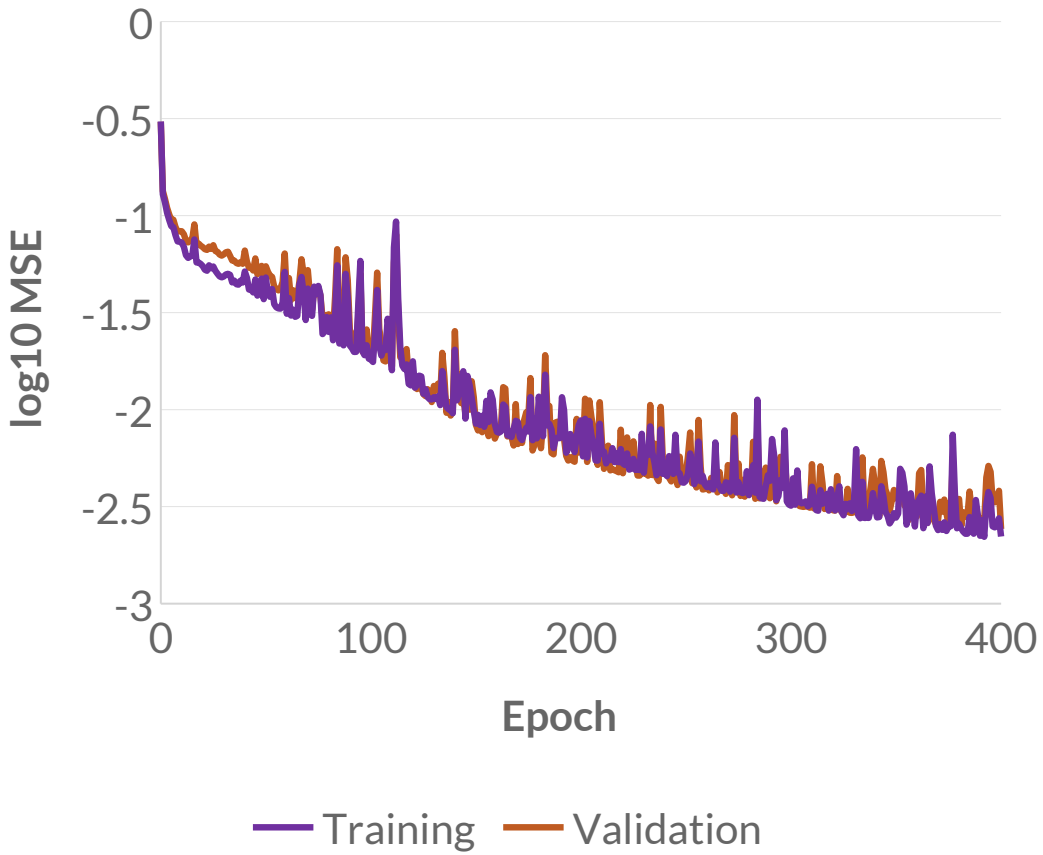
- Validation data is held out from updates.
- no_grad skips derivative recording.

Training the MLP

Learned prediction



Training and validation loss



Next: optimization

- Learning rates and stochastic gradients
- Momentum and adaptive optimizers
- Adam and AdamW
- Schedules, effective batches and training memory