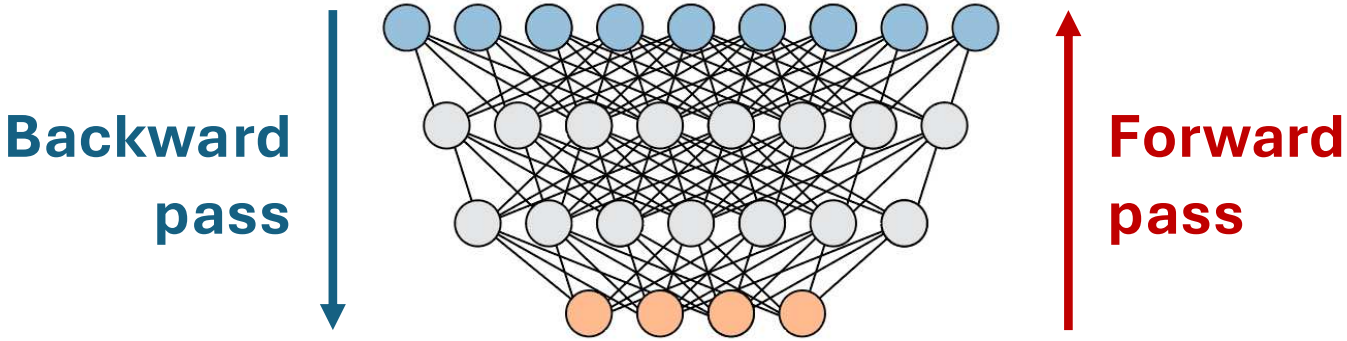


COM SCI M148, Lecture 14:
Training NNs with backprop

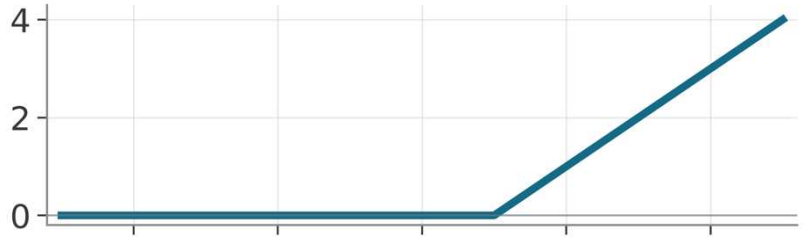
Prof. Nathan Szymanski
nszymanski@seas.ucla.edu
Department of Materials Science

UCLA

Samueli
School of Engineering

From last time: One neuron

$z_i^{(l)}$ is a weighted sum of activations from the previous layer



$w_{i1}^{(l)}$

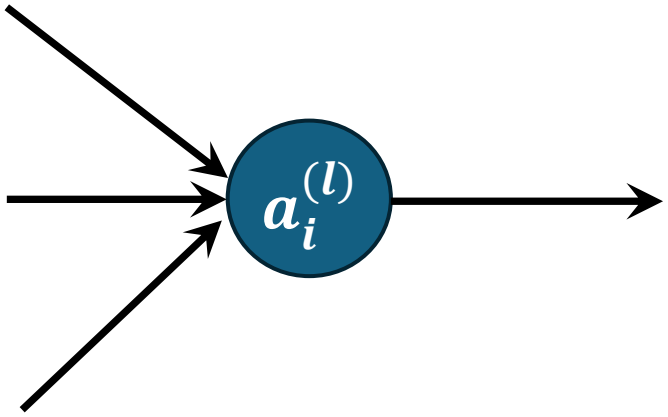
$w_{i2}^{(l)}$

$w_{i3}^{(l)}$

$$z_i^{(l)} = w_{i1}^{(l)} a_1^{(l-1)} + w_{i2}^{(l)} a_2^{(l-1)} + w_{i3}^{(l)} a_3^{(l-1)}$$

$$a_i^{(l)} = g^{(l)}(z_i^{(l)})$$

To next layer



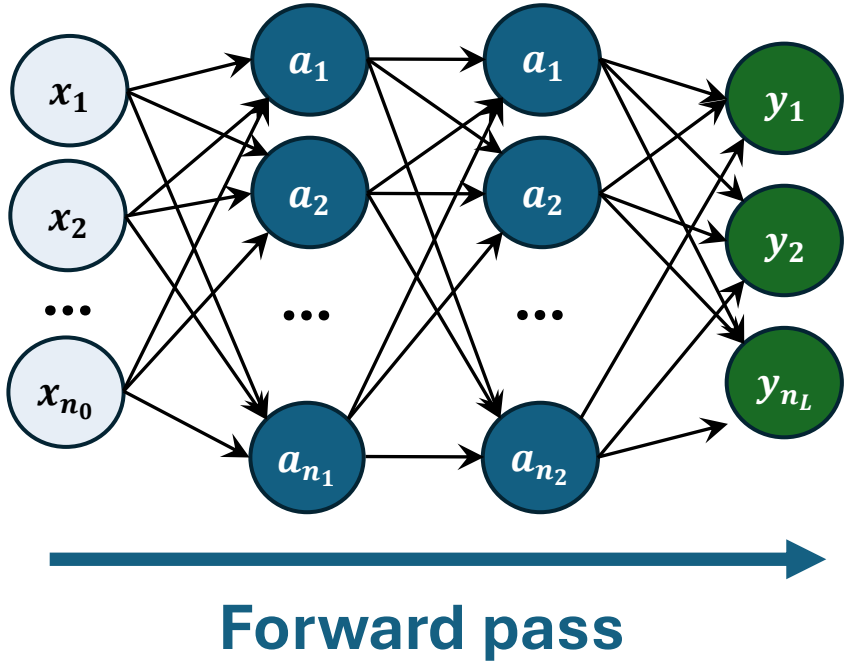
In the neural network, we draw this as one neuron

From last time: Neural networks

$$z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)}$$

$$a^{(l)} = g^{(l)}\left(z^{(l)}\right)$$

These equations are all we need to make a **forward pass** through the network: **input → output**



We choose activation when designing the model

We provide the input

Layer 0→1: $a^{(1)} = g^{(1)}\left(W^{(1)} x + b^{(1)}\right)$

Layer 1→2: $a^{(2)} = g^{(2)}\left(W^{(2)} a^{(1)} + b^{(2)}\right)$

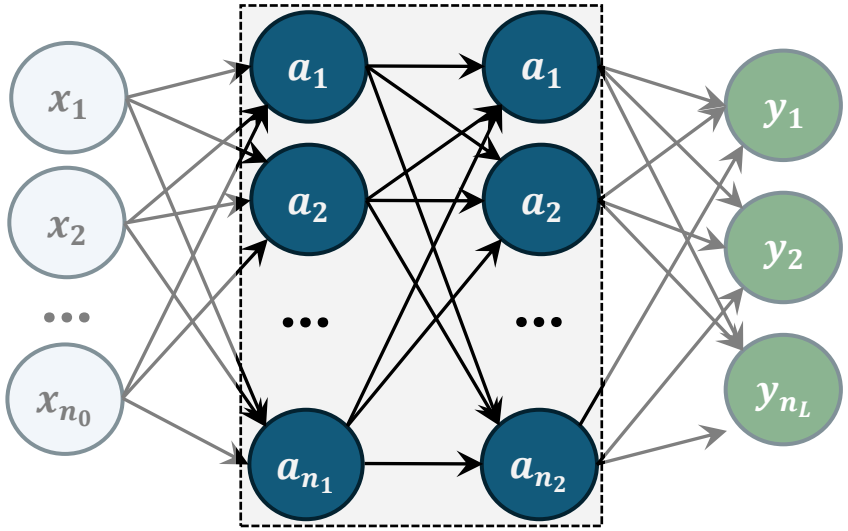
Layer 2→3: $\hat{y} = g^{(3)}\left(W^{(3)} a^{(2)} + b^{(3)}\right)$

Model gives us output

Weights and biases are *learned* during training

From last time: Weights and biases

How many trainable parameters does our network have?



$$N_{\text{params}}^{\text{total}} = \sum_{l=1}^{L} n_l(n_{l-1}+1)$$

The weights feeding into layer l are:

$$W^{(l)} = \begin{pmatrix} w_{11}^{(l)} & w_{12}^{(l)} & \cdots & w_{1n_{l-1}}^{(l)} \\ w_{21}^{(l)} & w_{22}^{(l)} & \cdots & w_{2n_{l-1}}^{(l)} \\ \cdots & \cdots & \cdots & \cdots \\ w_{n_l1}^{(l)} & w_{n_l2}^{(l)} & \cdots & w_{n_1n_{l-1}}^{(l)} \end{pmatrix}$$

This is an $n_l \times n_{l-1}$ matrix

Hence: $(n_l)(n_{l-1})$ params

And the bias vector for layer l is:

$$b^{(l)} = \begin{pmatrix} b_1^{(l)} \\ b_2^{(l)} \\ \cdots \\ b_{n_l}^{(l)} \end{pmatrix}$$

We also have n_l params for the bias

Together, layer l has $(n_l)(n_{l-1}) + (n_l)$ **trainable params**

As usual: Learn parameters that *minimize a loss*

Trainable model parameters:

$$\theta = \{W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}, \ldots, W^{(L)}, b^{(L)}\}$$

For one observation, the model produces:

$$\hat{y} = f(\boldsymbol{x}; \theta)$$

A **loss function** measures the prediction error:

$$\mathcal{L}(\hat{y}, y)$$

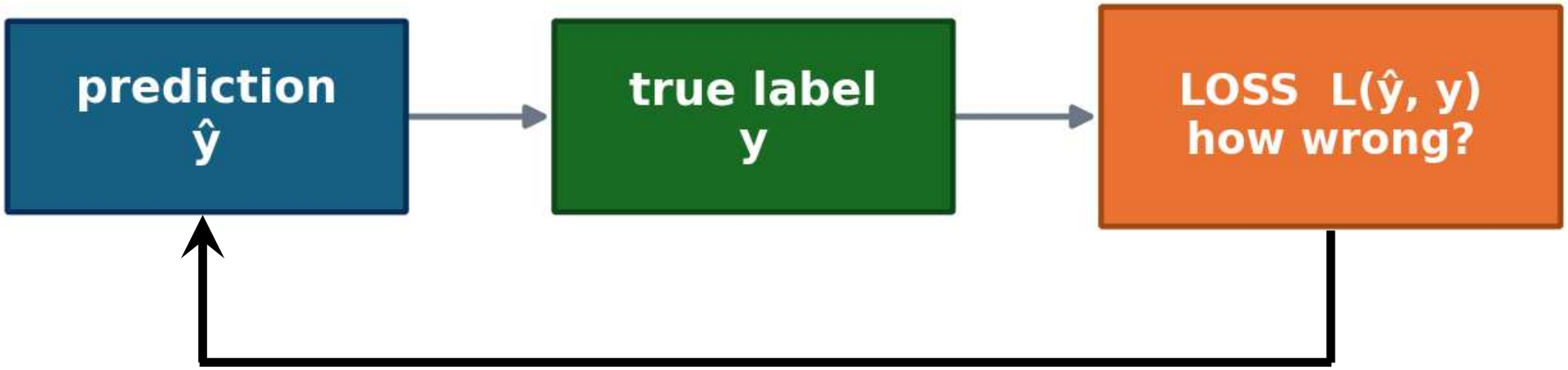
Minimize over the training set:

$$J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(\hat{y}_i, y_i)$$

Task	Output activation	Output shape	Loss
Regression	Linear (identity)	1 real value	Squared error (MSE)
Binary classification	Sigmoid	1 probability	Binary cross-entropy
Multiclass (K)	Softmax	K probs, sum = 1	Cross-entropy

This will be an iterative process

Training = finding the weights that make our model as “least wrong” as possible based on the observed data we have at hand

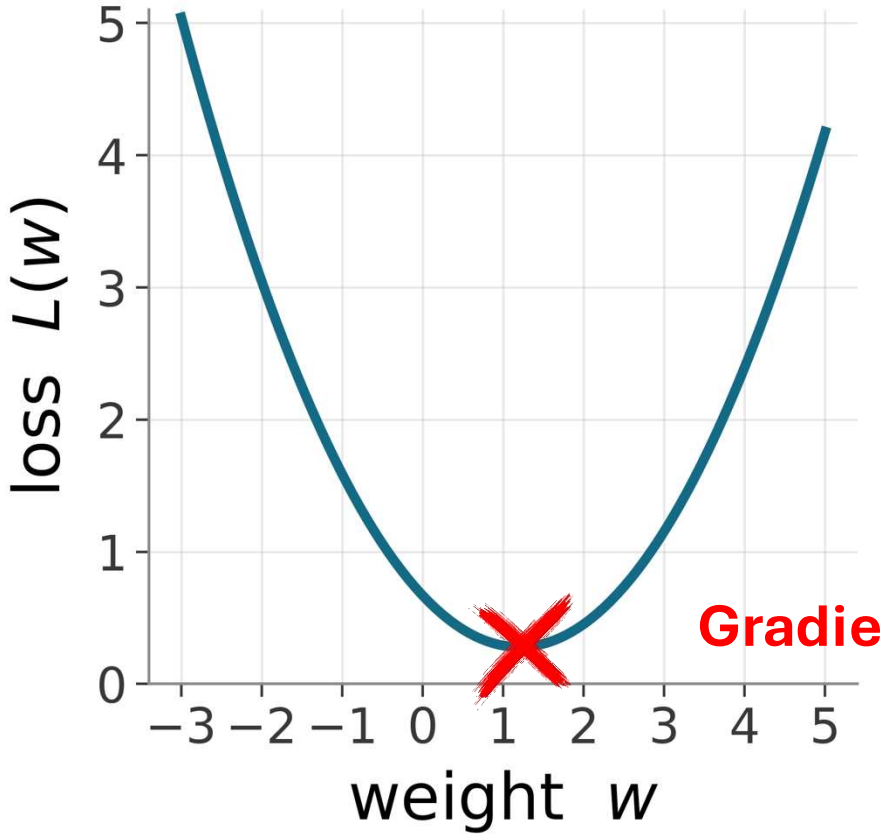


Update our model’s weights to make its prediction less wrong

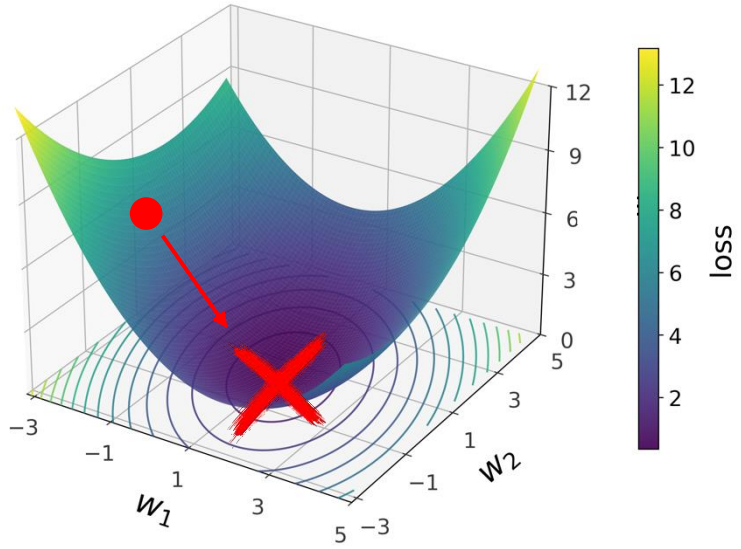
Why is this iteration necessary? Can’t we solve it directly?

Challenges in training neural networks

1. **No analytical solution:** Neural networks generally require iterative optimization, such as **gradient descent**, to minimize the loss.

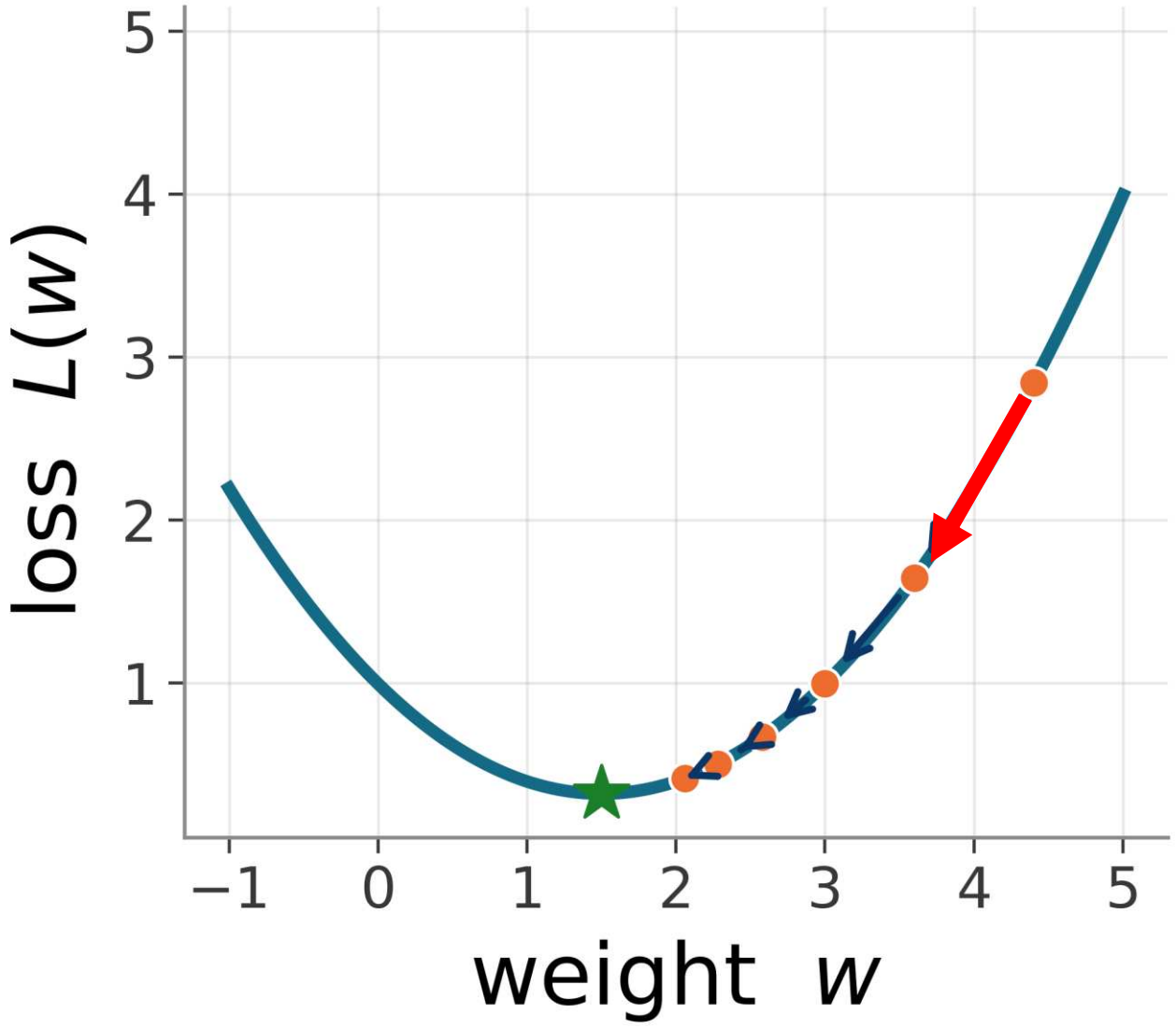


Can we set $\frac{\partial \mathcal{L}}{\partial w_i} = 0$ and solve? Sadly, because of the **non-linear activation functions**, we cannot…



But we *can* travel downhill to find the minimum iteratively

Recall: Gradient descent



For each weight (and bias)

In one dimension (1D):

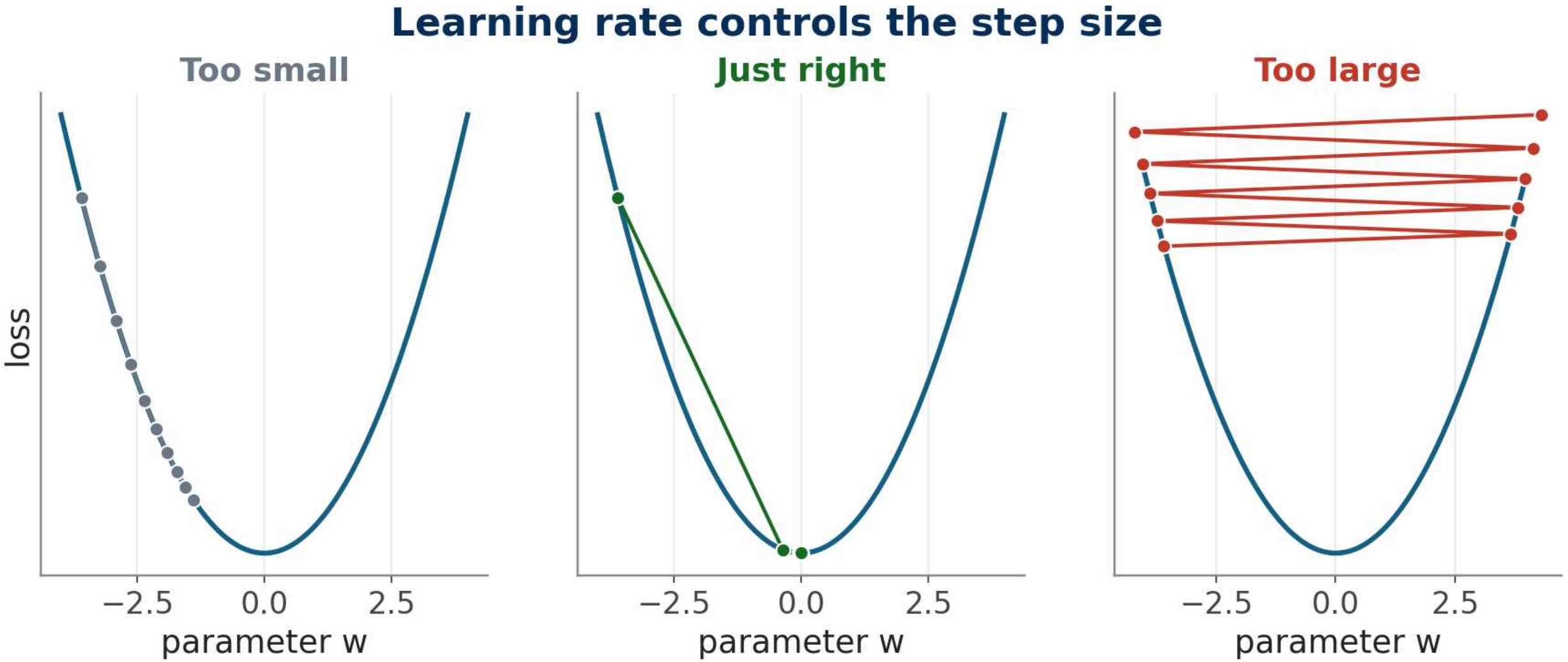
- Give a starting point ($\boldsymbol{w_i^0}$)
- Compute the derivative: $\boldsymbol{d\mathcal{L}/dw_i}$
- Update your parameters using:

$$\boldsymbol{w_i' = w_i^0 - \eta \left(\frac{d\mathcal{L}}{dw_i} \right) \Big|_{w_i = w_i^0}}$$

Learning rate

Choosing an appropriate learning rate is important!

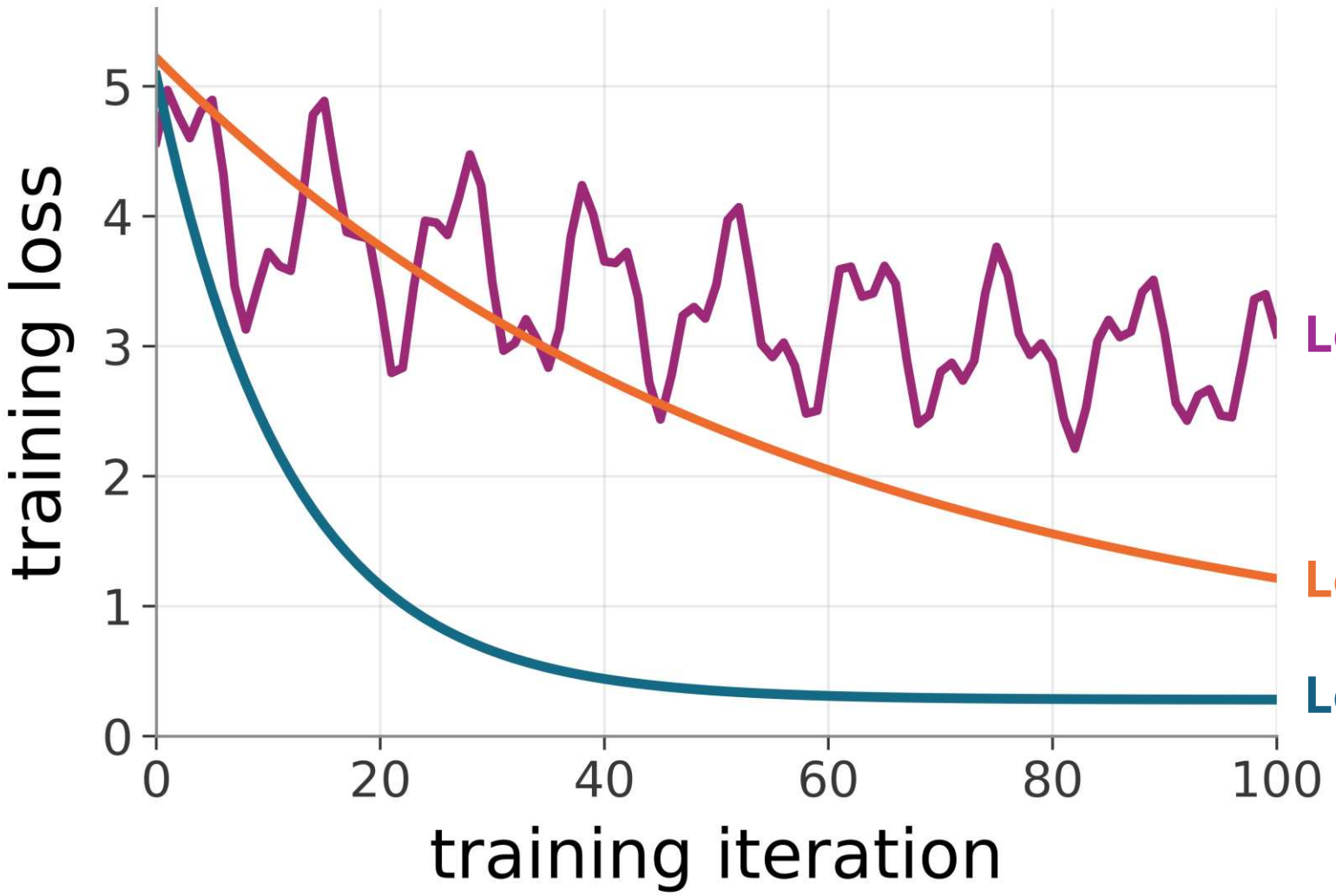
Effect of the learning rate



Usually, we start with a small default ($\eta \sim 10^{-3}$) and vary it from there, if needed

Diagnosing η from the loss curve



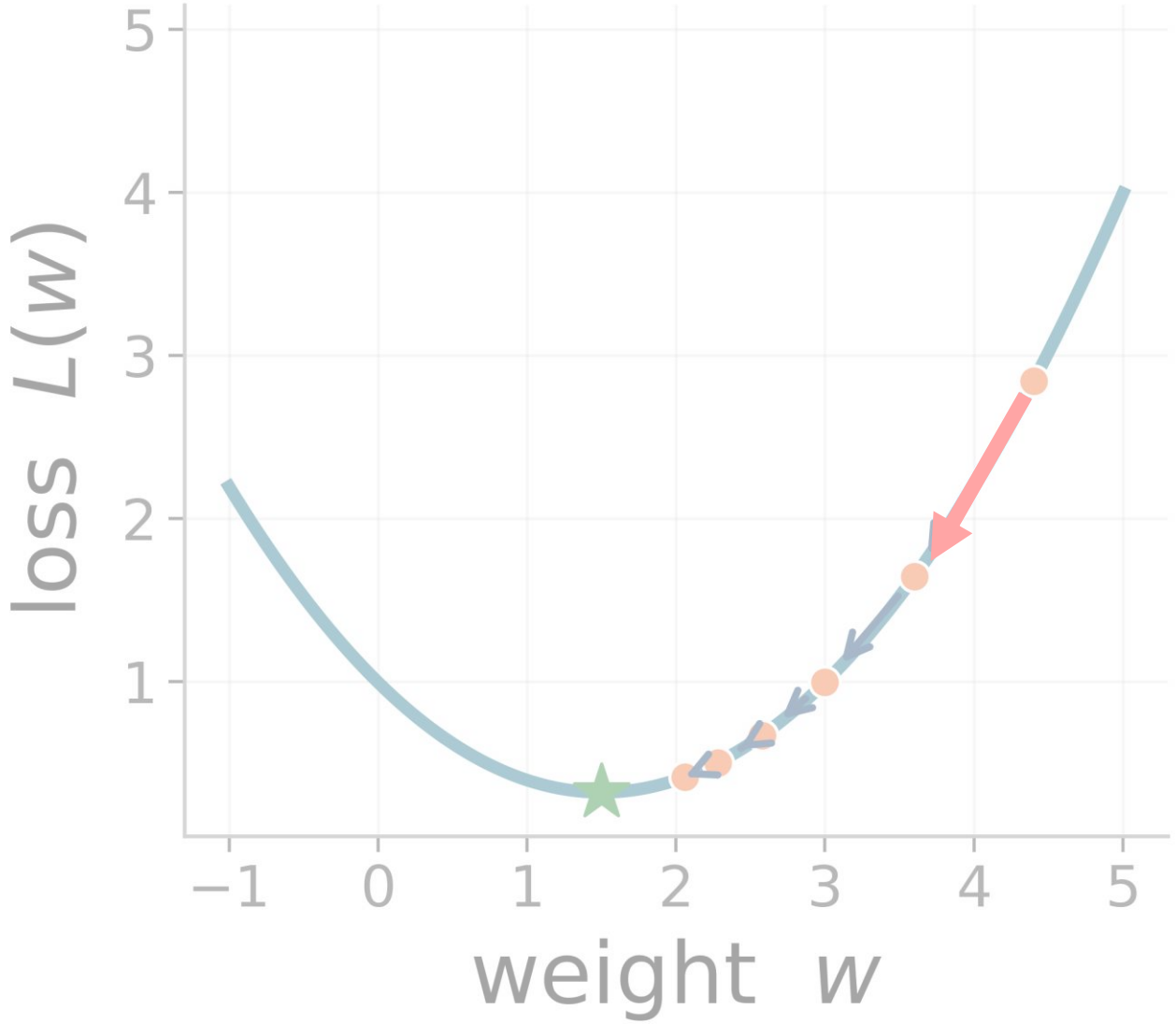


Learning rate too large

Learning rate too small

Learning rate just right

Recall: Gradient descent



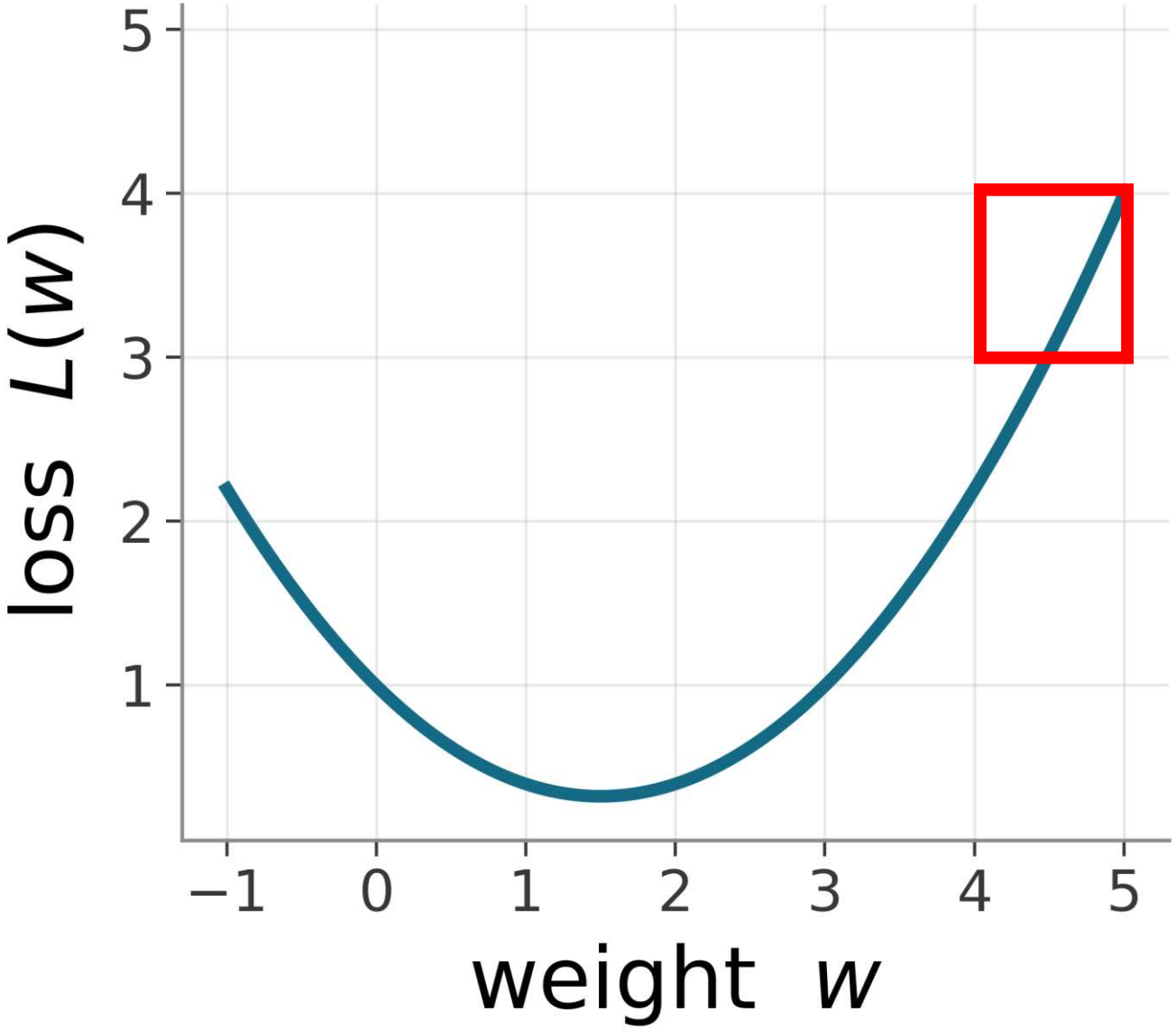
In one dimension (1D):

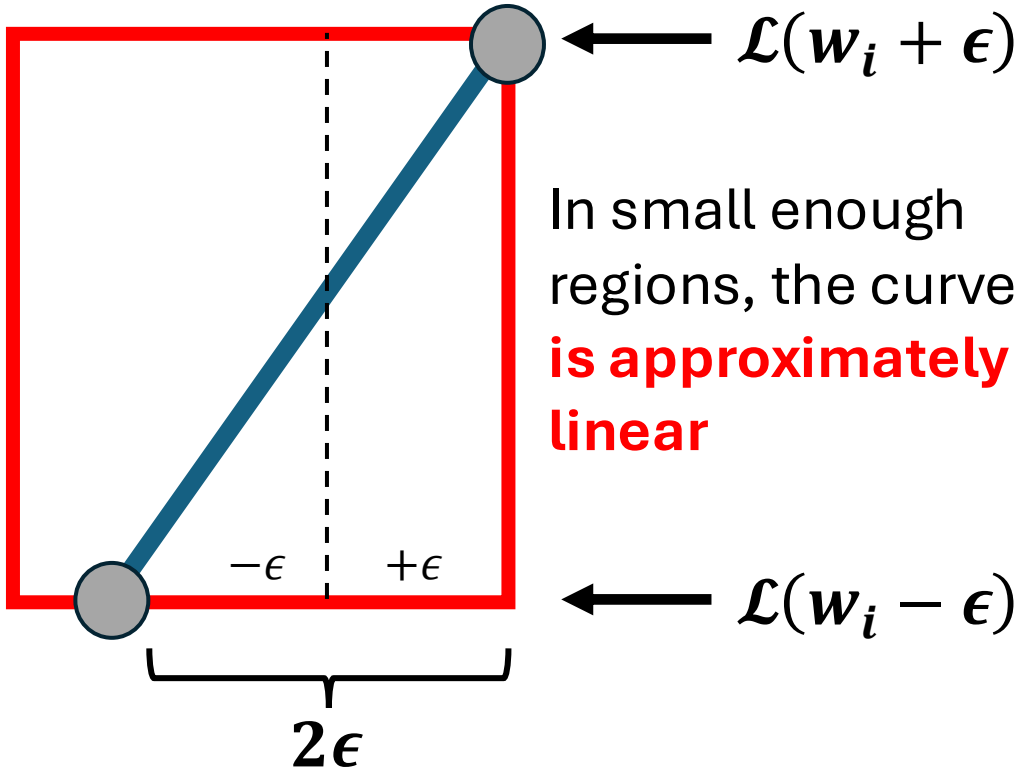
- Give a starting point (w_i^0)
- Compute the derivative: $d\mathcal{L}/dw_i$
- Update your parameters using:

$$w_i' = w_i^0 - \eta \left(\frac{d\mathcal{L}}{dw_i} \right) \Bigg|_{w_i = w_i^0}$$

How do we compute this derivative?

Option #1: Finite differences

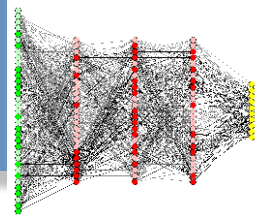




In small enough regions, the curve **is approximately linear**

$$\frac{\partial \mathcal{L}}{\partial w_i} \approx \frac{\mathcal{L}(w_i + \epsilon) - \mathcal{L}(w_i - \epsilon)}{2\epsilon} \quad \frac{\textbf{(Rise)}}{\textbf{(Run)}}$$

Challenges in training neural networks



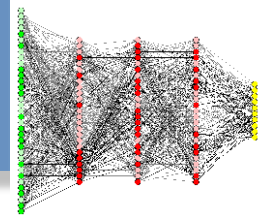
2. **Neural networks often have many trainable parameters**, so computing the derivative *via* finite differences becomes inefficient

- To compute a derivative with respect to w_i, we **compute the loss twice**
- To compute the **gradient**, we need to do this for ***all* weights and biases**

$$\nabla \mathcal{L} = \left\{ \frac{\partial \mathcal{L}}{\partial w_1}, \frac{\partial \mathcal{L}}{\partial w_2}, \frac{\partial \mathcal{L}}{\partial w_3}, \cdots, \frac{\partial \mathcal{L}}{\partial w_n} \right\}$$

$$\frac{\partial \mathcal{L}}{\partial w_i} \approx \frac{\mathcal{L}(w_i + \epsilon) - \mathcal{L}(w_i - \epsilon)}{2\epsilon} \quad \frac{(\text{Rise})}{(\text{Run})}$$

Challenges in training neural networks



2. **Neural networks often have many trainable parameters**, so computing the derivative *via* finite differences becomes inefficient

- To compute a derivative with respect to w_i, we **compute the loss twice**
- To compute the **gradient**, we need to do this for ***all* weights and biases**

$$\nabla \mathcal{L} = \left\{ \frac{\partial \mathcal{L}}{\partial w_1}, \frac{\partial \mathcal{L}}{\partial w_2}, \frac{\partial \mathcal{L}}{\partial w_3}, \cdots, \frac{\partial \mathcal{L}}{\partial w_n} \right\}$$

- This means that for a network with N parameters, $\mathbf{2N_{params}^{total}}$ **forward passes are needed for just one iteration of gradient descent!**

$$\boldsymbol{N_{\text{params}}^{\text{total}} = \sum_{l=1}^{L} n_l (n_{l-1} + 1)}$$

$$\boldsymbol{\frac{\partial \mathcal{L}}{\partial w_i} \approx \frac{\mathcal{L}(w_i + \epsilon) - \mathcal{L}(w_i - \epsilon)}{2\epsilon} \quad \frac{(\text{Rise})}{(\text{Run})}}$$

Remember…Modern neural networks are huge!

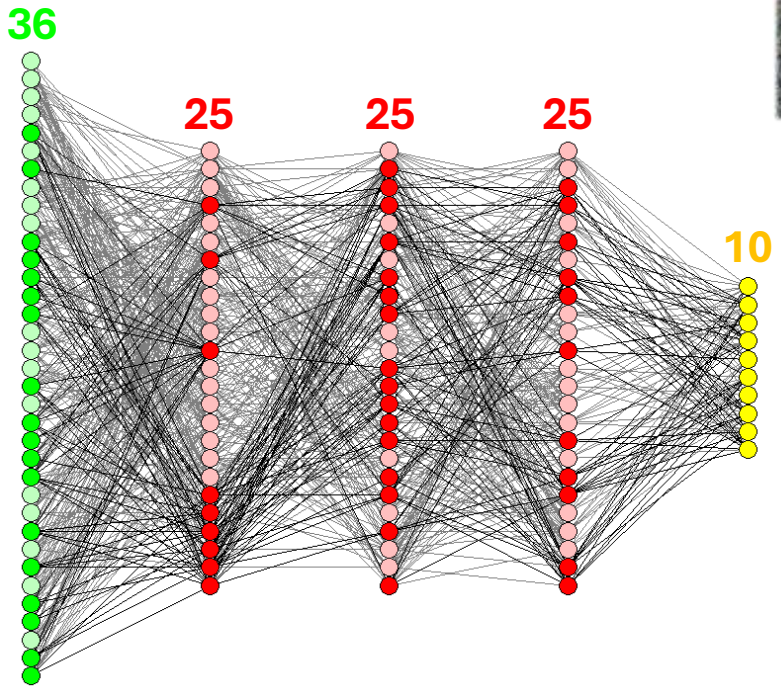
Model	Parameters
GPT-2	**1.5 billion**
GPT-3	**175 billion**
GPT-4	**1.8 trillion**
GPT-5	**50 trillion**

Training these models… Expensive!



$$N_{\text{params}} = 25(36+1) + 25(25+1) + 25(25+1) + 10(25+1) = \mathbf{2,485}$$

This is a relatively simple NN!



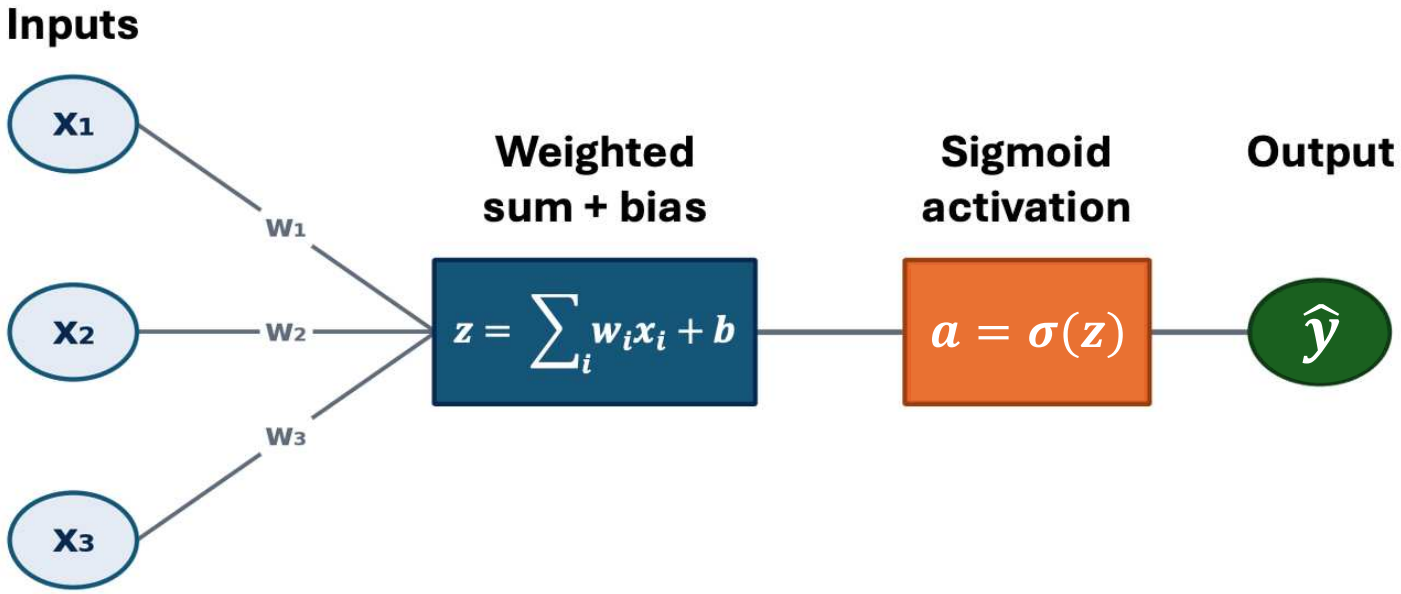


Option #2: Compute derivatives using chain rule

$$\frac{d}{dx}\left[f\left(g(x)\right)\right] = f'\left(g(x)\right)g'(x)$$

$$\frac{df}{dx} = \frac{df}{dg}\frac{dg}{dx}$$

Option #2: Compute derivatives using chain rule



First consider just one neuron:

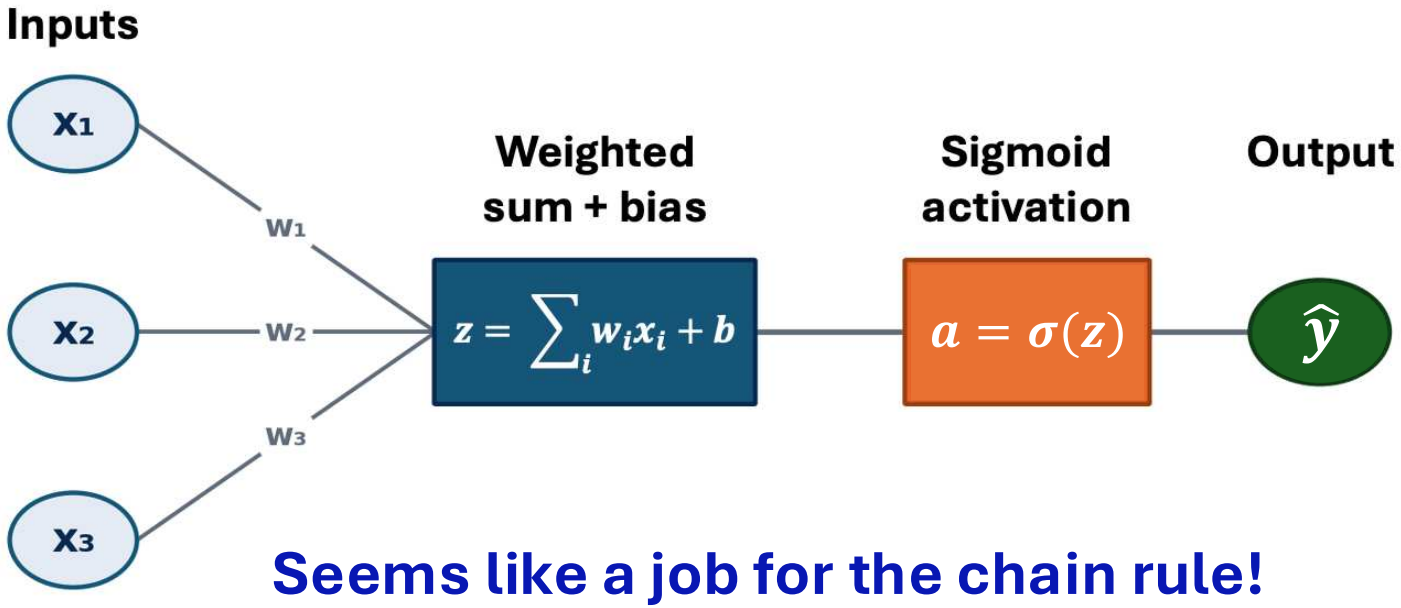
$$\hat{y} = a(z(w, b, x))$$

A composition of functions

The **loss** is itself is a function of the model's output:

$$\mathcal{L}(\hat{y}, y) = \mathcal{L}(a(z(w, b, x)), y)$$

Option #2: Compute derivatives using chain rule



Seems like a job for the chain rule!

$$\frac{\partial \mathcal{L}}{\partial w_i} = \frac{\partial \mathcal{L}}{\partial a}\frac{\partial a}{\partial w_i}$$

But a is also some nested function of w

First consider just one neuron:

$$\hat{y} = a(z(w, b, x))$$

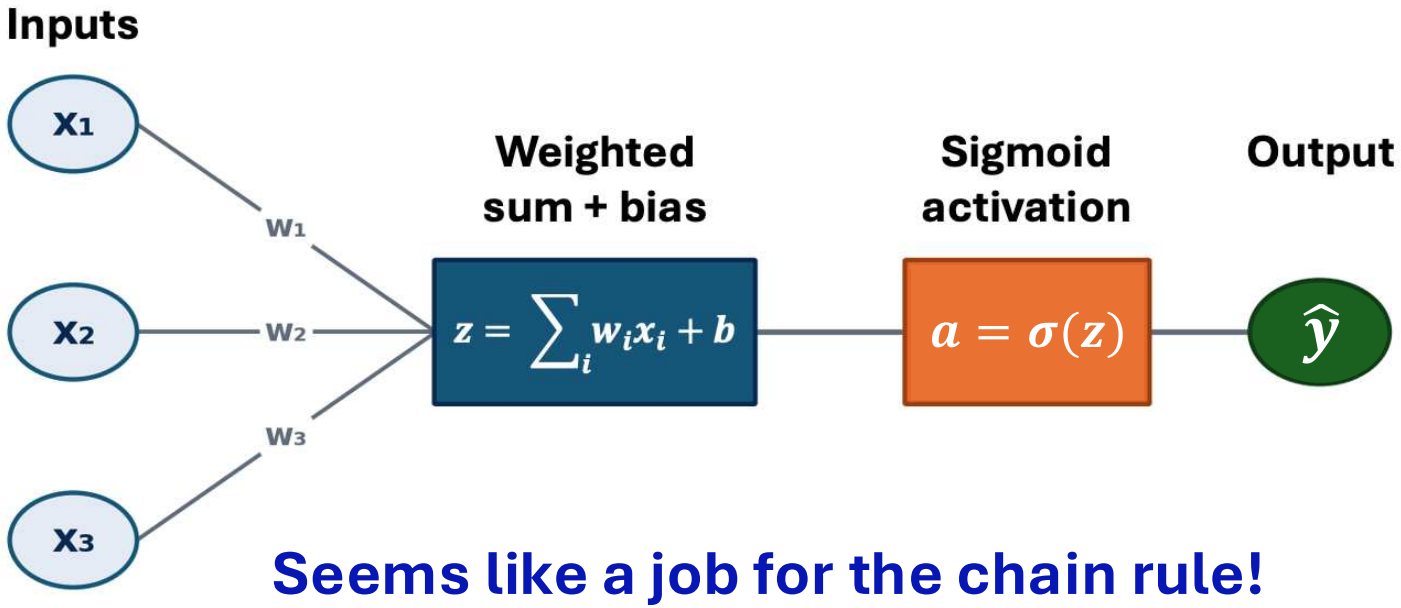
A composition of functions

The **loss** is itself is a function of the model's output:

$$\mathcal{L}(\hat{y}, y) = \mathcal{L}(a(z(w, b, x)), y)$$

What we want is the **derivative of $\mathcal{L}$** with respect to each **weight w_i**

Option #2: Compute derivatives using chain rule



Seems like a job for the chain rule!

$$\frac{\partial \mathcal{L}}{\partial w_i} = \frac{\partial \mathcal{L}}{\partial a}\frac{\partial a}{\partial w_i} = \frac{\partial \mathcal{L}}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial w_i}$$

First consider just one neuron:

$$\hat{y} = a(z(w, b, x))$$

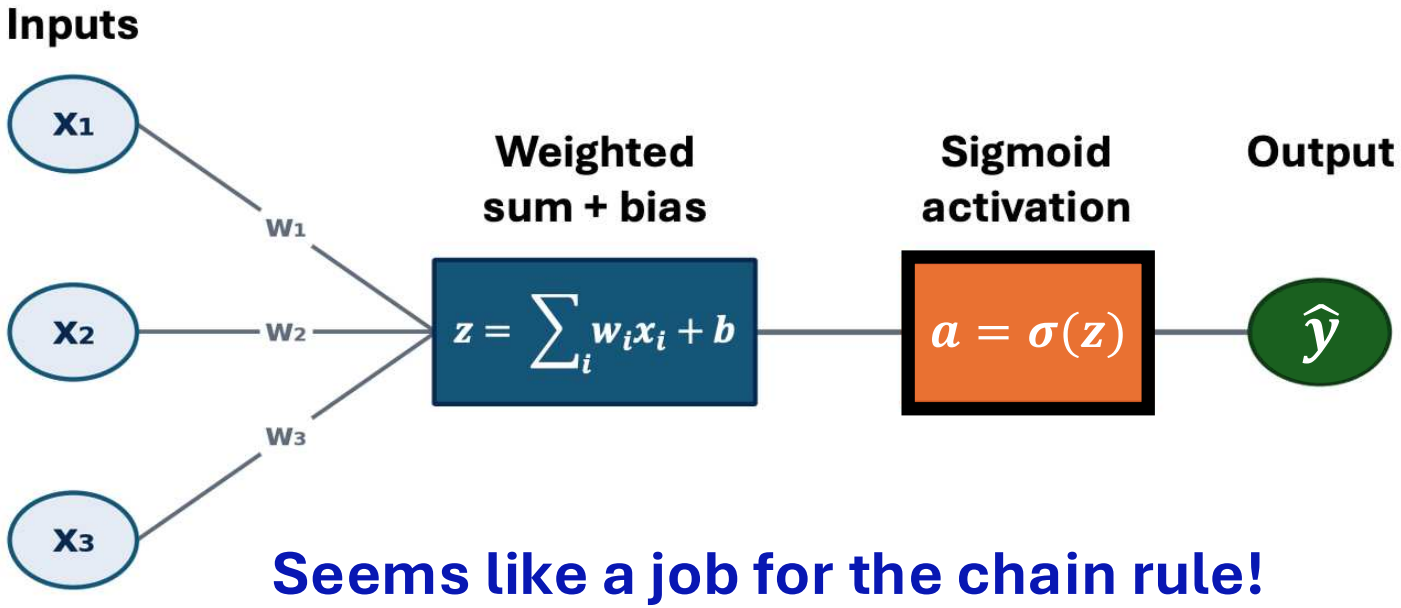
A composition of functions

The **loss** is itself is a function of the model's output:

$$\mathcal{L}(\hat{y}, y) = \mathcal{L}(a(z(w, b, x)), y)$$

What we want is the **derivative of $\mathcal{L}$** with respect to each **weight w_i**

Option #2: Compute derivatives using chain rule



First consider just one neuron:

$$\hat{y} = a(z(w, b, x))$$

A composition of functions

The **loss** is itself is a function of the model's output:

$$\mathcal{L}(\hat{y}, y) = \mathcal{L}(a(z(w, b, x)), y)$$

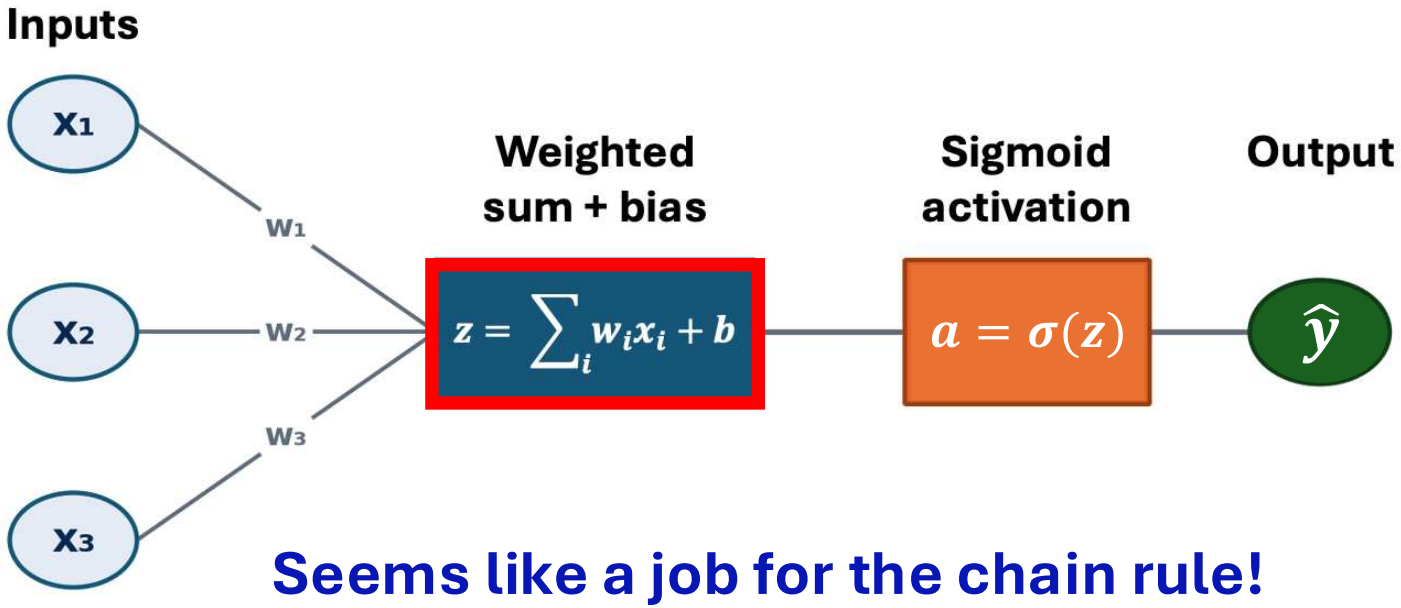
Seems like a job for the chain rule!

$$\frac{\partial \mathcal{L}}{\partial w_i} = \frac{\partial \mathcal{L}}{\partial a}\frac{\partial a}{\partial w_i} = \frac{\partial \mathcal{L}}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial w_i}$$

This is just the derivative of sigmoid!

$$\frac{\partial a}{\partial z} = \sigma'(z)$$

Option #2: Compute derivatives using chain rule



First consider just one neuron:

$$\hat{y} = a(z(w, b, x))$$

A composition of functions

The **loss** is itself is a function of the model's output:

$$\mathcal{L}(\hat{y}, y) = \mathcal{L}(a(z(w, b, x)), y)$$

Seems like a job for the chain rule!

$$\frac{\partial \mathcal{L}}{\partial w_i} = \frac{\partial \mathcal{L}}{\partial a}\frac{\partial a}{\partial w_i} = \frac{\partial \mathcal{L}}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial w_i}$$

$$\frac{\partial}{\partial w_i}(w_1 x_1 + \cdots w_i x_i + \cdots) = x_i$$

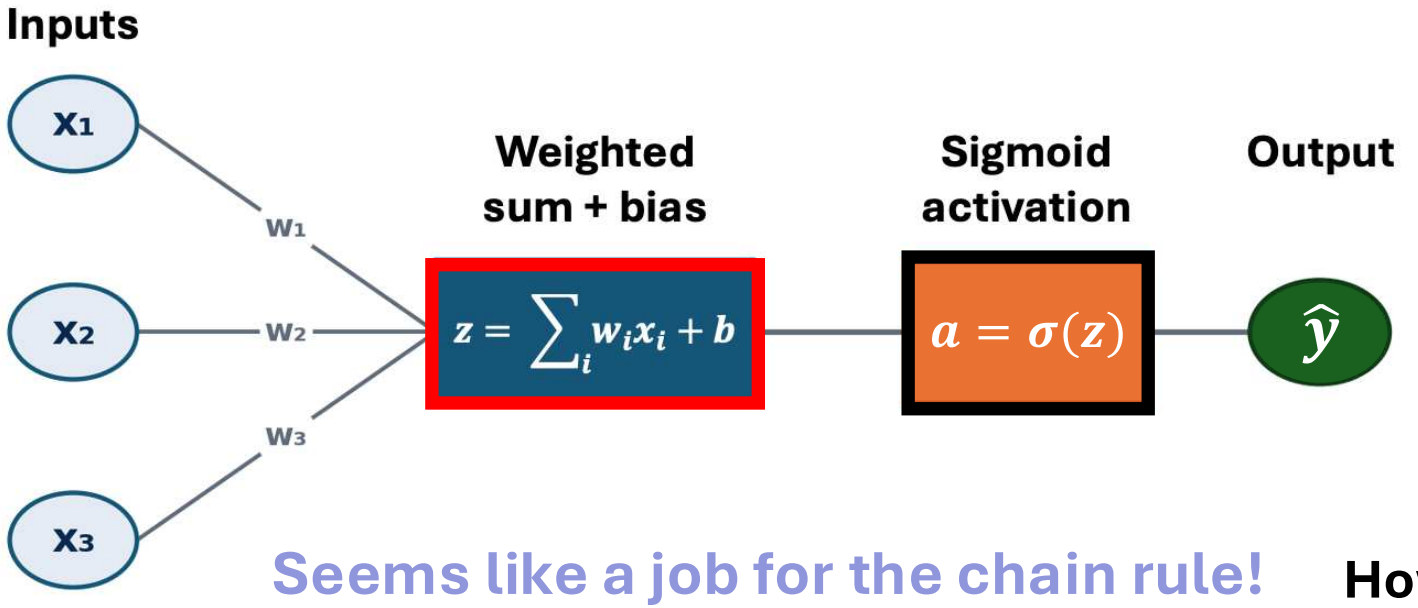
This is just the derivative of sigmoid!

$$\frac{\partial a}{\partial z} = \sigma'(z)$$

$$\frac{\partial z}{\partial w_i} = x_i$$

This is just one of our inputs!

Option #2: Compute derivatives using chain rule



Seems like a job for the chain rule!

$$\frac{\partial \mathcal{L}}{\partial w_i} = \frac{\partial \mathcal{L}}{\partial a}\frac{\partial a}{\partial w_i} = \frac{\partial \mathcal{L}}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial w_i}$$

This is just the derivative of sigmoid!

$$\frac{\partial a}{\partial z} = \sigma'(z)$$

$$\frac{\partial z}{\partial w_i} = x_i$$

This is just one of our inputs!

Putting it all together:

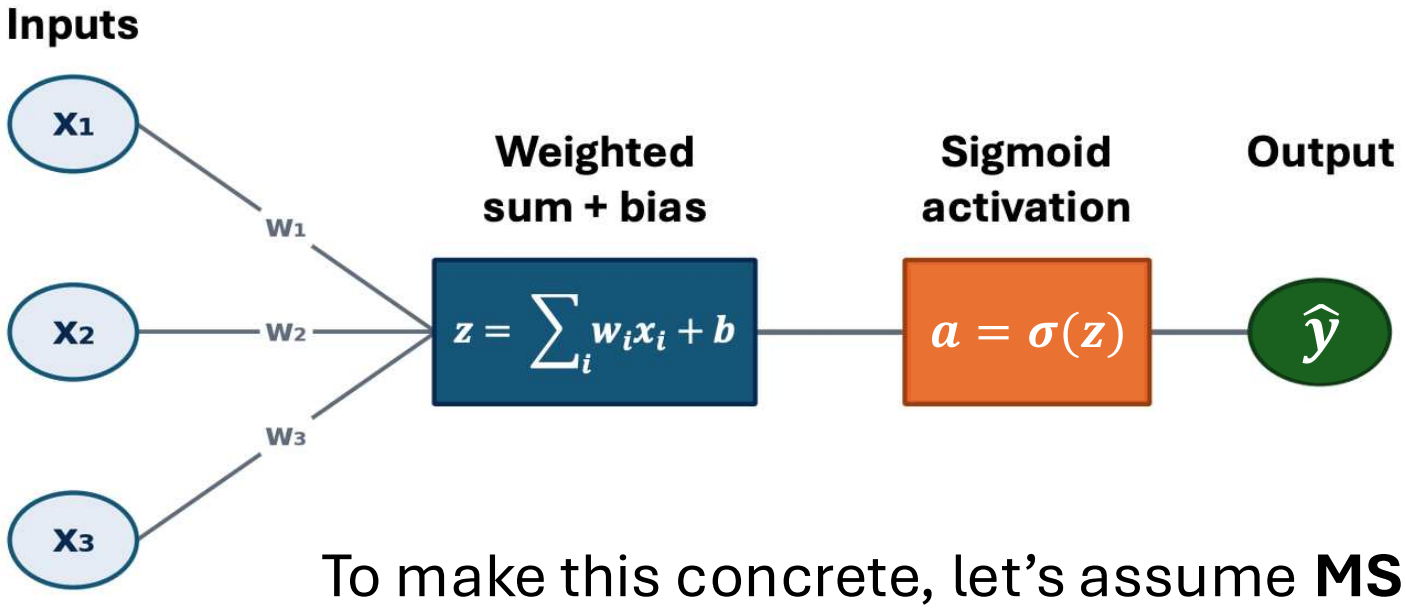
$$\frac{\partial \mathcal{L}}{\partial w_i} = \frac{\partial \mathcal{L}}{\partial a}\sigma'(z)x_i$$

How the loss changes with the output

Activation slope

Input feeding the weight

Option #2: Compute derivatives using chain rule



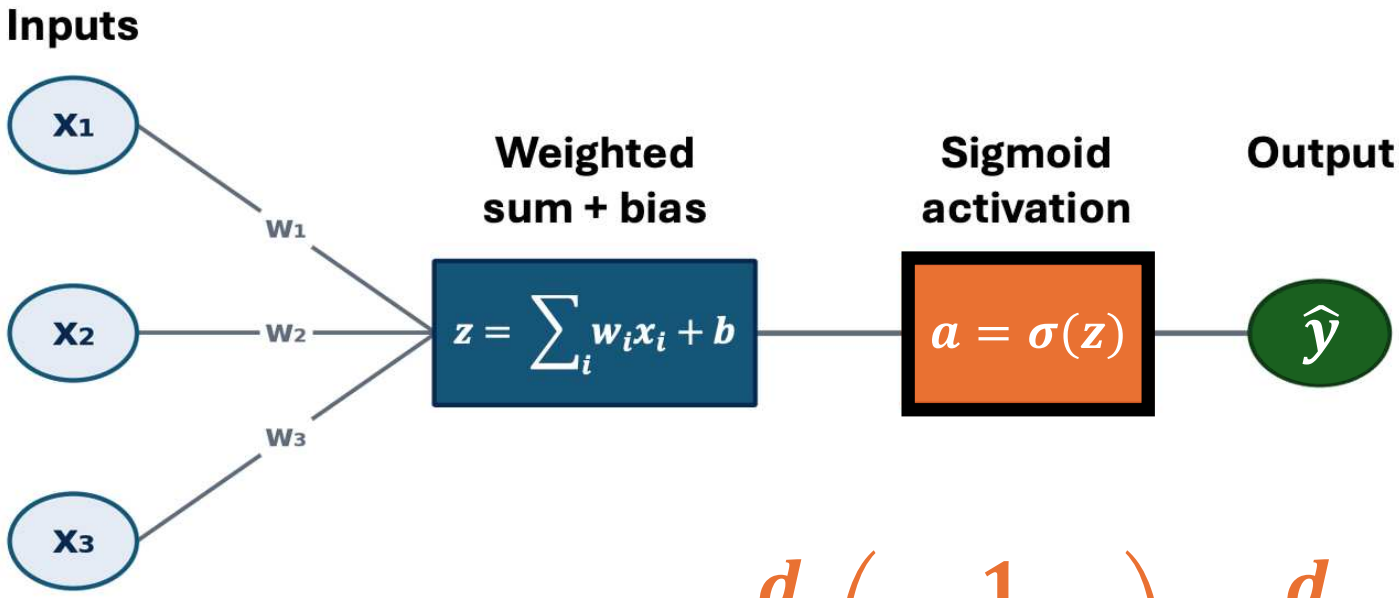
Putting it all together:

$$\frac{\partial \mathcal{L}}{\partial w_i} = \frac{\partial \mathcal{L}}{\partial a} \sigma'(z) x_i$$

To make this concrete, let's assume **MSE loss:** $\mathcal{L} = (\hat{y} - y)^2 = (a - y)^2$

The **derivative** is then: $\partial \mathcal{L} / \partial a = 2(a - y)$

Option #2: Compute derivatives using chain rule



Putting it all together:

$$\frac{\partial \mathcal{L}}{\partial w_i} = \frac{\partial \mathcal{L}}{\partial a}\sigma'(z)x_i$$

More chain rule!

$$\sigma'(z) = \frac{d}{dz}\left(\frac{1}{1+e^{-z}}\right) = \frac{d}{dz}(1+e^{-z})^{-1} = e^{-z}(1+e^{-z})^{-2}$$

We can rewrite this as:

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

$$= \sigma(z)\left(1-\frac{1}{1+e^{-z}}\right) = \sigma(z)\big(1-\sigma(z)\big) = a(1-a)$$

Option #2: Compute derivatives using chain rule

For sigmoid activation, MSE loss:

$$\frac{\partial \mathcal{L}}{\partial w_i} = 2(a - y)a(1 - a)x_i$$

Putting it all together:

$$\frac{\partial \mathcal{L}}{\partial w_i} = \frac{\partial \mathcal{L}}{\partial a}\sigma'(z)x_i$$

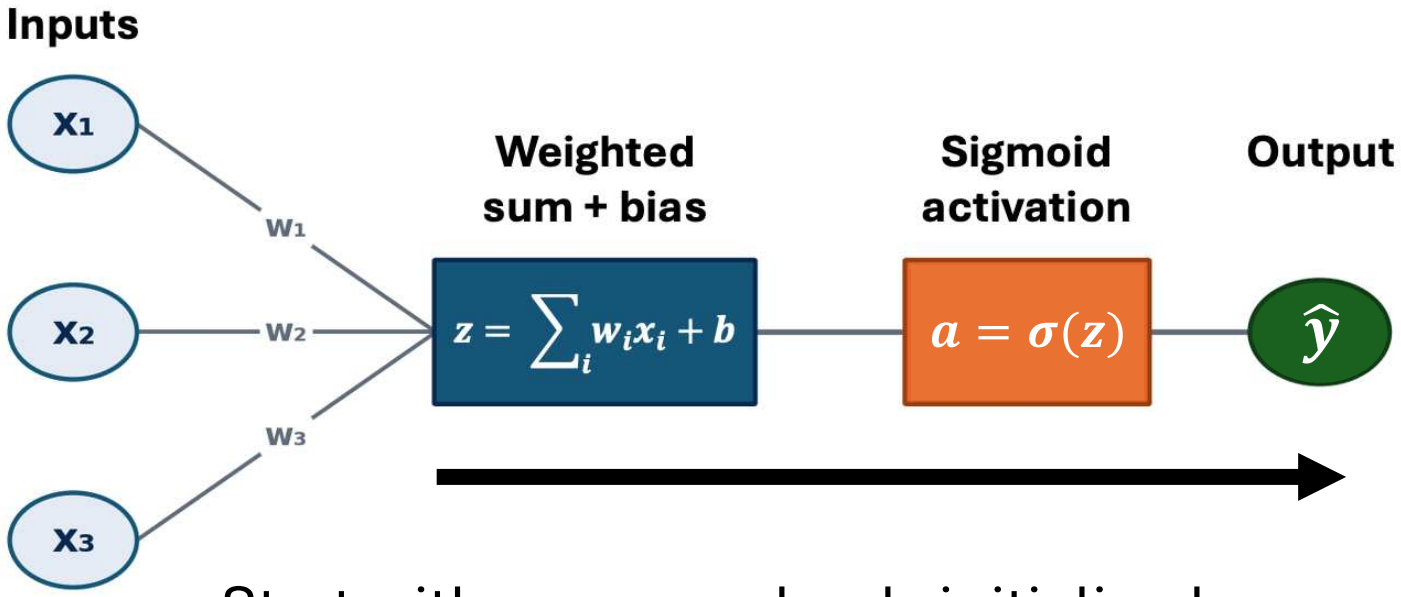
A nice, simple equation to compute the derivative of each weight! All we need is:

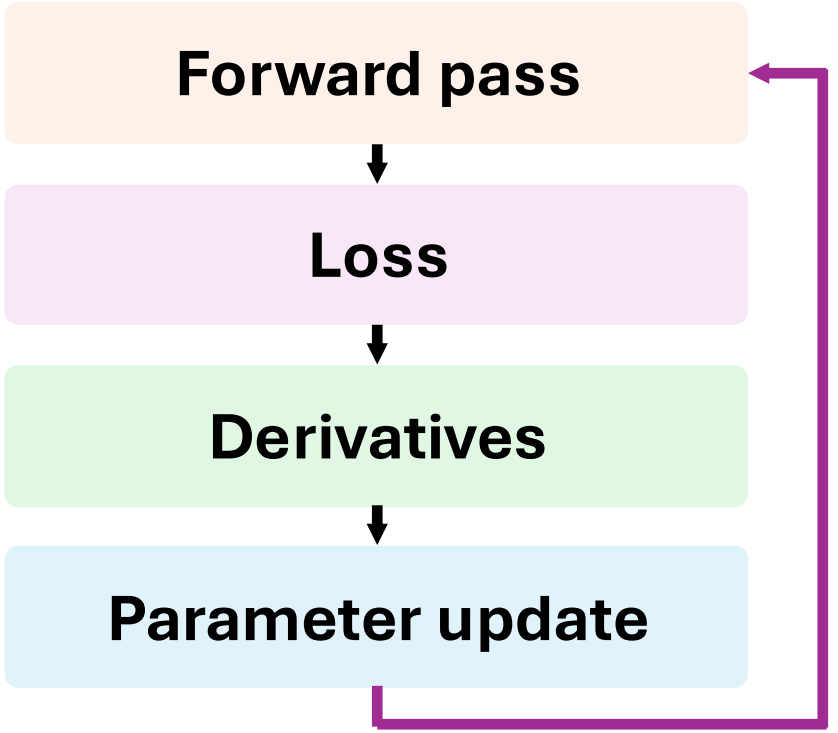
- **The input** (x_i) feeding that weight
- **The predicted output** (a) of the model
- **The ground truth** (y) to compare against

We get all these from one forward pass.
For all weights!

A similar derivation can be done for the bias.
Try it at home if you get bored ☺

Plug the derivatives into gradient descent





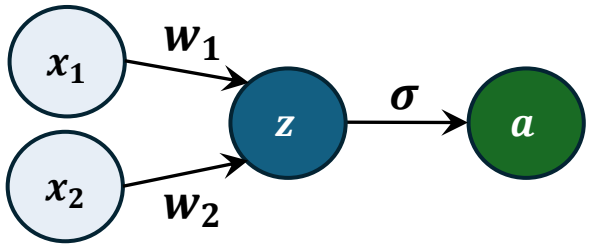
Start with some randomly initialized weights w^0 and then compute:

$$w_i' = w_i^0 - \eta \left(\frac{d\mathcal{L}}{dw_i} \right) \Bigg|_{w_i = w_i^0}$$

Do the same for the bias (b)

$$= w_i^0 - 2\eta a(a - y)(1 - a)x_i$$ ⟶ **This tells us what next weight to try!**

An example with real numbers



Suppose we have a sample with features: $\boldsymbol{x_1 = 1, x_2 = 2}$ and a known target of $\boldsymbol{y = 1}$ used for training

Randomly initialize small weights ($\boldsymbol{w_1 = 0.2, w_2 = -0.1}$) and zero bias ($\boldsymbol{b = 0}$) → this is a typical starting point

Do a **forward pass:** $z = w_1x_1 + w_2x_2 + b = (0.2)(1) + (-0.1)(2) = 0 \rightarrow a = \sigma(0) = \mathbf{0.5}$

Compute the **loss:** $\mathcal{L} = (0.5 - 1)^2 = \mathbf{0.25}$

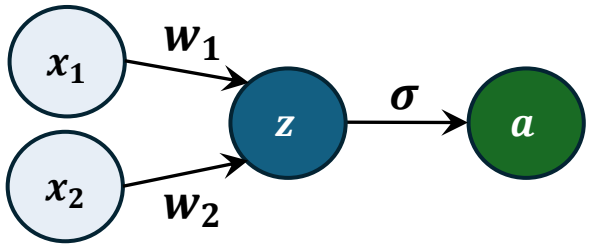
Compute **derivatives:** $\frac{\partial \mathcal{L}}{\partial w_i} = \boldsymbol{2a(a-y)(1-a)x_i}$ and $\frac{\partial \mathcal{L}}{\partial b} = \boldsymbol{2a(a-y)(1-a)}$ **Similar but no x_i**

$$\frac{\partial \mathcal{L}}{\partial w_1} = 2(0.5)(0.5-1)(1-0.5)(1) = -0.25$$

$$\frac{\partial \mathcal{L}}{\partial w_2} = 2(0.5)(0.5-1)(1-0.5)(2) = -0.5$$

$$\frac{\partial \mathcal{L}}{\partial b} = 2(0.5)(0.5-1)(1-0.5) = -0.25$$

An example with real numbers



$$w_i' = w_i^0 - \eta \left(\frac{d\mathcal{L}}{dw_i} \right) \Bigg|_{w_i = w_i^0}$$

Set the learning rate $\eta = 0.1$

$$w_0' = (0.2) - (0.1)(-0.25) = 0.225$$

$$w_1' = (-0.1) - (0.1)(-0.5) = -0.05$$

$$b' = (0) - (0.1)(-0.25) = 0.025$$

Forward pass

↓

Loss

↓

Derivatives

↓

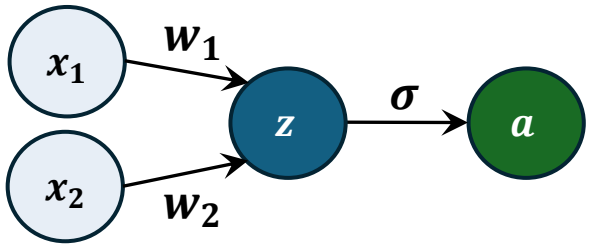
Parameter update

$$\frac{\partial \mathcal{L}}{\partial w_1} = 2(0.5)(0.5 - 1)(1 - 0.5)(1) = -0.25$$

$$\frac{\partial \mathcal{L}}{\partial w_2} = 2(0.5)(0.5 - 1)(1 - 0.5)(2) = -0.5$$

$$\frac{\partial \mathcal{L}}{\partial b} = 2(0.5)(0.5 - 1)(1 - 0.5) = -0.25$$

An example with real numbers



$$w_i' = w_i^0 - \eta \left(\frac{d\mathcal{L}}{dw_i} \right) \Bigg|_{w_i = w_i^0}$$

Set the learning rate $\eta = 0.1$

$$w_0' = (0.2) - (0.1)(-0.25) = 0.225$$

$$w_1' = (-0.1) - (0.1)(-0.5) = -0.05$$

$$b' = (0) - (0.1)(-0.25) = 0.025$$

Forward pass

↓

Loss

↓

Derivatives

↓

Parameter update

Do another forward pass using the updated weights and bias:

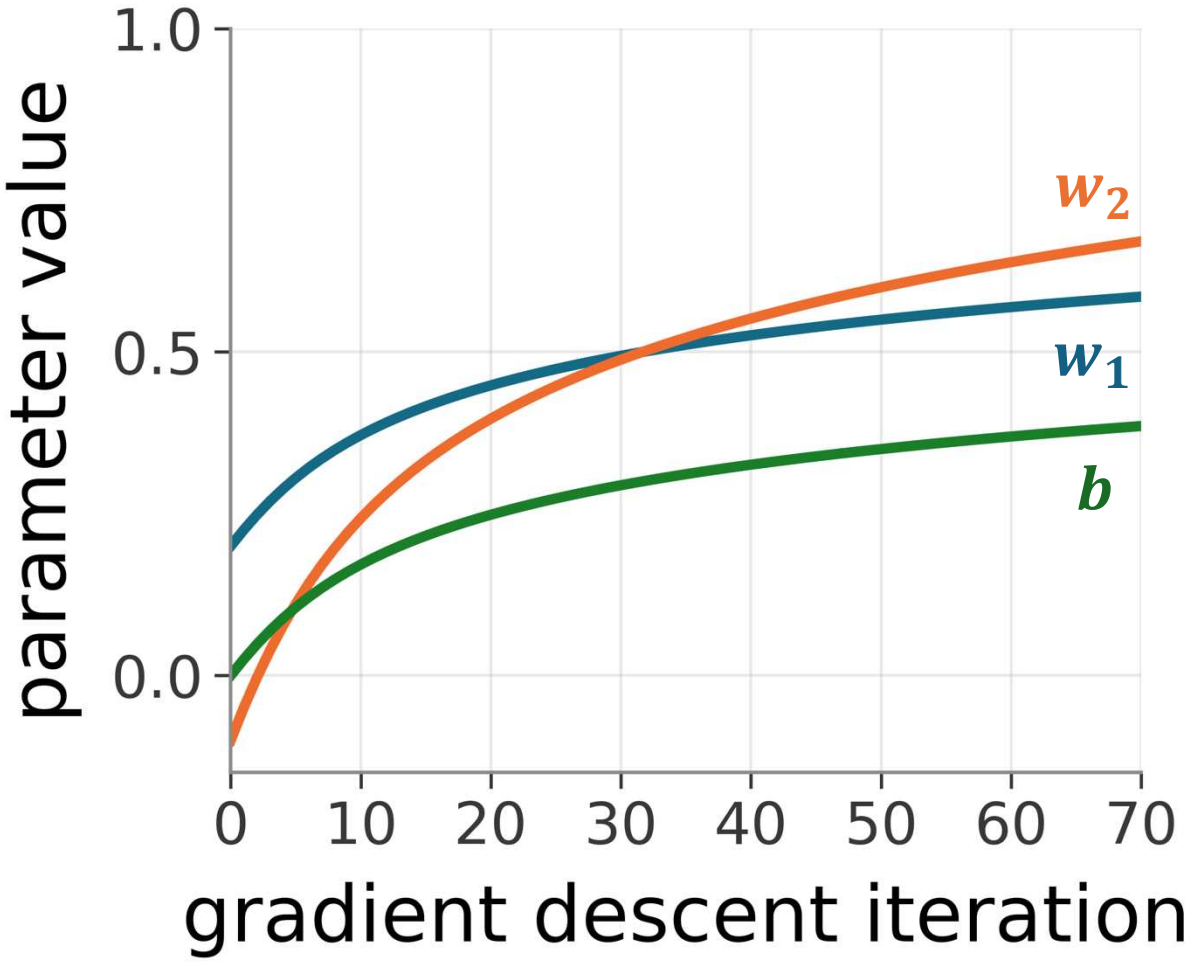
Ground truth: $y = 1$

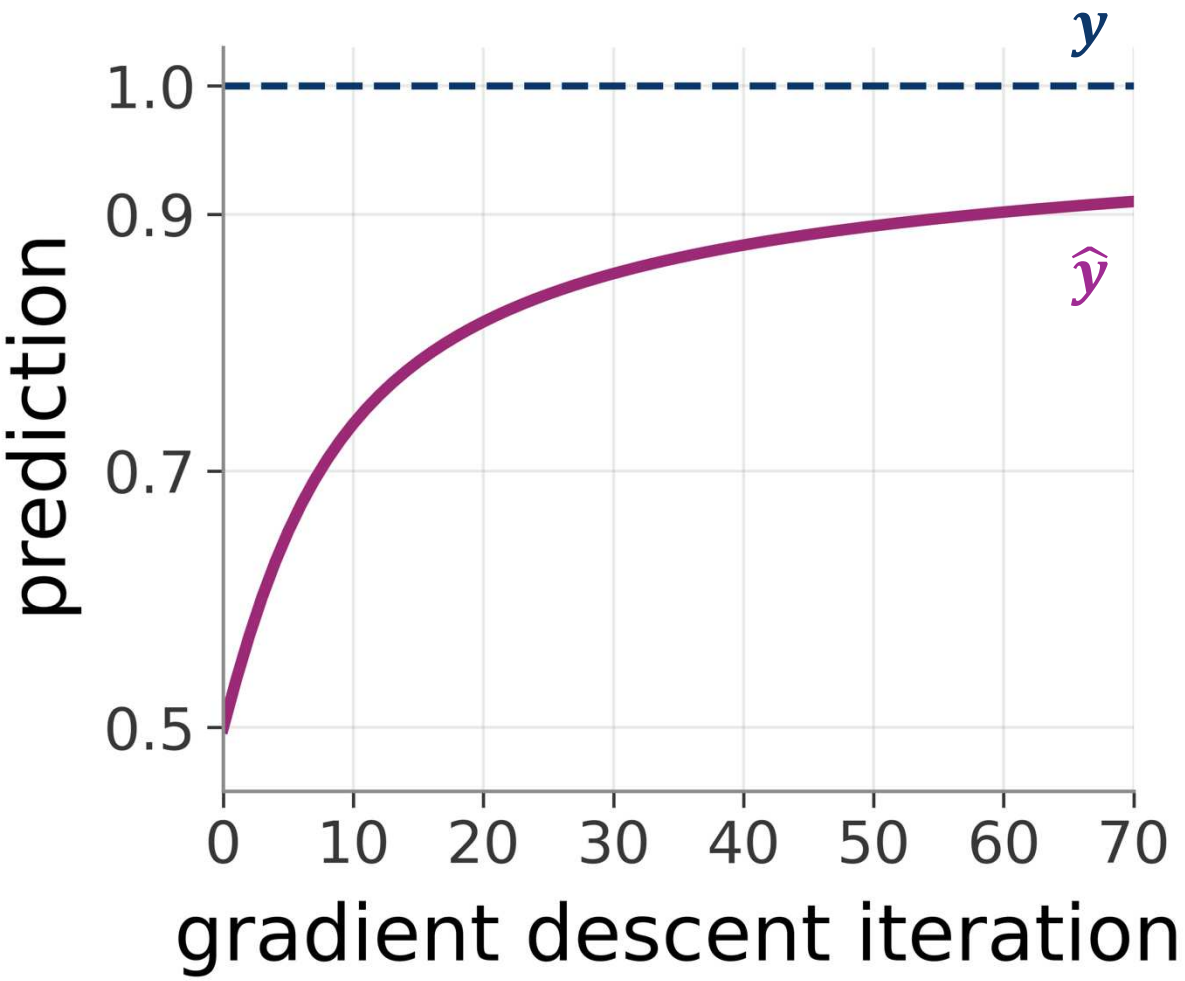
$$z = w_1 x_1 + w_2 x_2 + b = (0.225)(1) + (-0.05)(2) + 0.025 = 0.15$$

$$\rightarrow a = \sigma(0.15) = 0.53$$

We went from 0.5 to 0.53 after one update (our target is 1) – we're slowly getting closer!

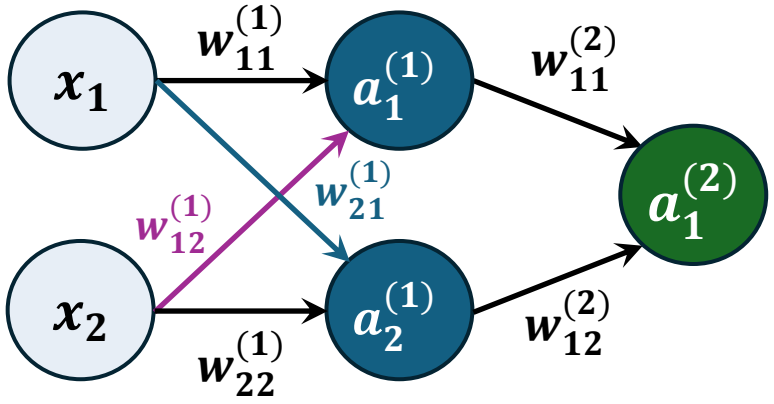
Here's what happens if we keep iterating





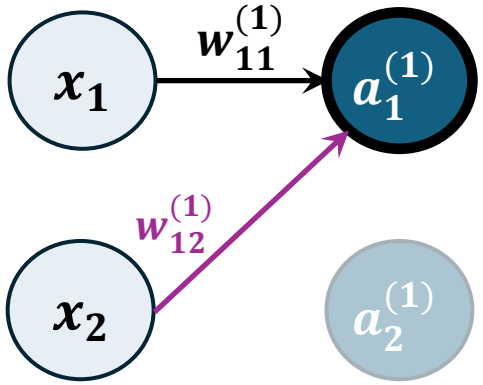
Questions?

Introducing > 1 neuron in a hidden layer



This is our output ($\hat{y}$)

Introducing > 1 neuron in a hidden layer



$a_1^{(2)}$

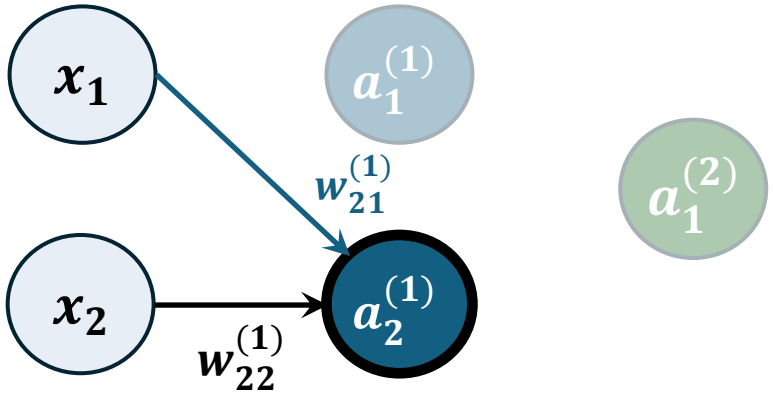
$$z_1^{(1)} = w_{11}^{(1)} x_1 + w_{12}^{(1)} x_2 + b_1^{(1)}$$

Weighted sum

$$a_1^{(1)} = g^{(1)}(z_1^{(1)})$$

Nonlinear activation

Introducing > 1 neuron in a hidden layer



$a_1^{(2)}$

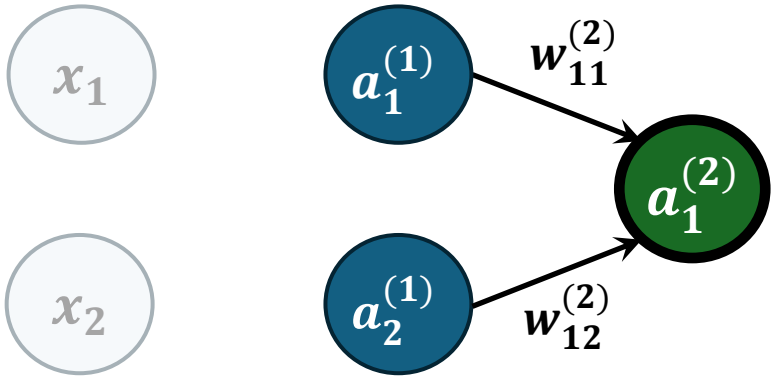
$$z_1^{(1)} = w_{11}^{(1)} x_1 + w_{12}^{(1)} x_2 + b_1^{(1)}$$

$$a_1^{(1)} = g^{(1)}(z_1^{(1)})$$

$$z_2^{(1)} = w_{21}^{(1)} x_1 + w_{22}^{(1)} x_2 + b_2^{(1)}$$

$$a_2^{(1)} = g^{(1)}(z_2^{(1)})$$

Introducing > 1 neuron in a hidden layer



$$z_1^{(1)} = w_{11}^{(1)} x_1 + w_{12}^{(1)} x_2 + b_1^{(1)}$$

$$a_1^{(1)} = g^{(1)}(z_1^{(1)})$$

$$z_2^{(1)} = w_{21}^{(1)} x_1 + w_{22}^{(1)} x_2 + b_2^{(1)}$$

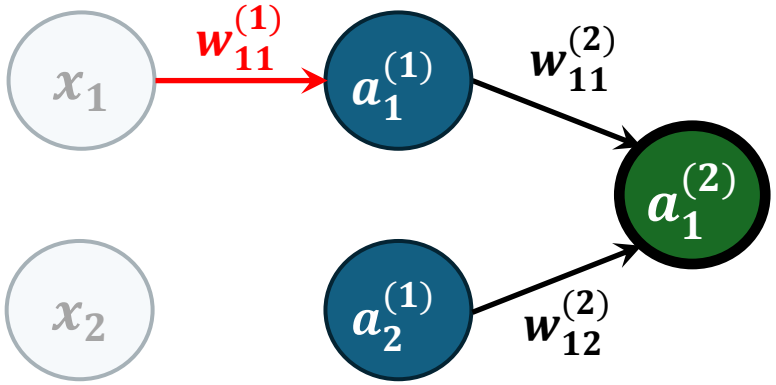
$$a_2^{(1)} = g^{(1)}(z_2^{(1)})$$

The final output is a linear combination of the hidden layer's activated outputs

$$z_1^{(2)} = w_{11}^{(2)} a_1^{(1)} + w_{12}^{(2)} a_2^{(1)} + b_1^{(2)}$$

$$a_1^{(2)} = g^{(2)}(z_1^{(2)})$$

Introducing > 1 neuron in a hidden layer



$$z_1^{(1)} = w_{11}^{(1)} x_1 + w_{12}^{(1)} x_2 + b_1^{(1)}$$

$$a_1^{(1)} = g^{(1)}(z_1^{(1)})$$

$$z_2^{(1)} = w_{21}^{(1)} x_1 + w_{22}^{(1)} x_2 + b_2^{(1)}$$

$$a_2^{(1)} = g^{(1)}(z_2^{(1)})$$

Now, suppose we want:

$$\frac{\partial \mathcal{L}}{\partial w_{11}^{(1)}}$$

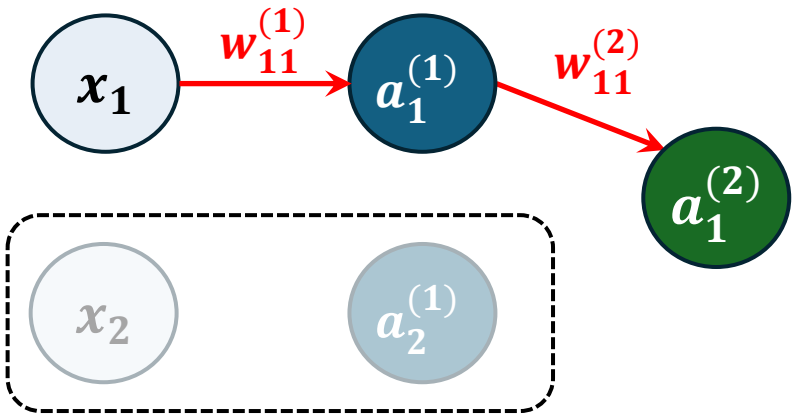
(from 0th → 1st layer)

$$z_1^{(2)} = w_{11}^{(2)} a_1^{(1)} + w_{12}^{(2)} a_2^{(1)} + b_1^{(2)}$$

$$a_1^{(2)} = g^{(2)}(z_1^{(2)})$$

Remember, $\mathcal{L}$ is some nested function of $w_{11}^{(1)}$

Which neurons depend on $w_{11}^{(1)}$



Not touched by $w_{11}^{(1)}$

$$z_1^{(1)} = \boldsymbol{w_{11}^{(1)}} x_1 + w_{12}^{(1)} x_2 + b_1^{(1)}$$
$$a_1^{(1)} = g^{(1)}(z_1^{(1)})$$

This neuron depends on $w_{11}^{(1)}$

$$z_2^{(1)} = w_{21}^{(1)} x_1 + w_{22}^{(1)} x_2 + b_2^{(1)}$$
$$a_2^{(1)} = g^{(1)}(z_2^{(1)})$$

Has no dependence on $w_{11}^{(1)}$

Now, suppose we want:

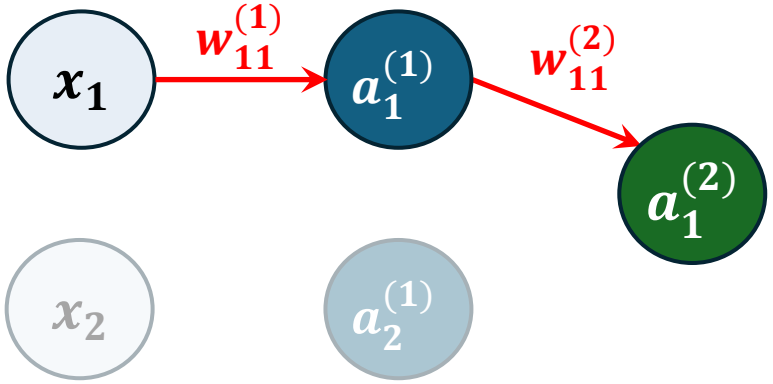
$$\frac{\partial \mathcal{L}}{\partial w_{11}^{(1)}}$$

Apply chain rule along this path
- Goal: connect output to input -

$$z_1^{(2)} = w_{11}^{(2)} \boldsymbol{a_1^{(1)}} + w_{12}^{(2)} a_2^{(1)} + b_1^{(2)}$$
$$a_1^{(2)} = g^{(2)}(z_1^{(2)})$$

This depends on a_1 which depends on $w_{11}^{(1)}$

Chain rule again, just longer…



$$z_1^{(1)} = \boldsymbol{w_{11}^{(1)}} x_1 + w_{12}^{(1)} x_2 + b_1^{(1)}$$

$$a_1^{(1)} = g^{(1)}(z_1^{(1)})$$

$$z_1^{(2)} = w_{11}^{(2)} \boldsymbol{a_1^{(1)}} + w_{12}^{(2)} a_2^{(1)} + b_1^{(2)}$$

$$\boxed{a_1^{(2)}} = g^{(2)}(z_1^{(2)})$$

$$\frac{\partial \mathcal{L}}{\partial w_{11}^{(1)}} = \frac{\partial \mathcal{L}}{\partial a_1^{(2)}} \frac{\partial a_1^{(2)}}{\partial w_{11}^{(1)}}$$

Need to get this 2nd term in terms of something I know

1st term I can calculate

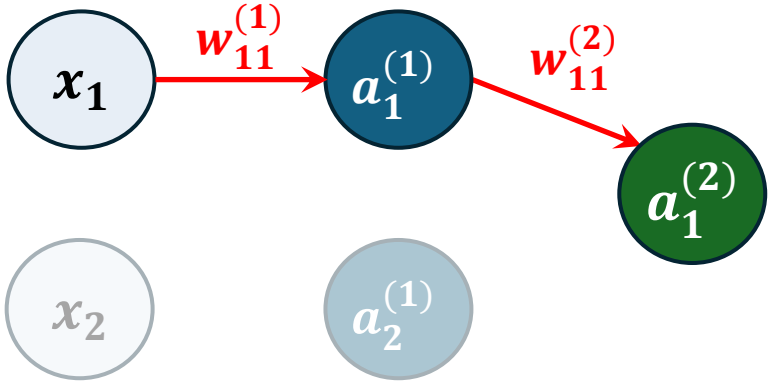
$$\frac{\partial a_1^{(2)}}{\partial z_1^{(2)}} \frac{\partial z_1^{(2)}}{\partial w_{11}^{(1)}}$$

Our output ($\hat{y}$) – start here and work our way "backwards"

$$\frac{\partial z_1^{(2)}}{\partial a_1^{(1)}} \frac{\partial a_1^{(1)}}{\partial w_{11}^{(1)}}$$

$$\frac{\partial a_1^{(1)}}{\partial z_1^{(1)}} \frac{\partial z_1^{(1)}}{\partial w_{11}^{(1)}}$$

Chain rule again, just longer…



$$\frac{\partial \mathcal{L}}{\partial w_{11}^{(1)}} = \frac{\partial \mathcal{L}}{\partial a_1^{(2)}} \frac{\partial a_1^{(2)}}{\partial z_1^{(2)}} \frac{\partial z_1^{(2)}}{\partial a_1^{(1)}} \frac{\partial a_1^{(1)}}{\partial z_1^{(1)}} \frac{\partial z_1^{(1)}}{\partial w_{11}^{(1)}}$$

We essentially go "backward" along this path:

$$w_{11}^{(1)} \leftarrow z_1^{(1)} \leftarrow a_1^{(1)} \leftarrow z_1^{(2)} \leftarrow a_1^{(2)} \leftarrow \mathcal{L}$$

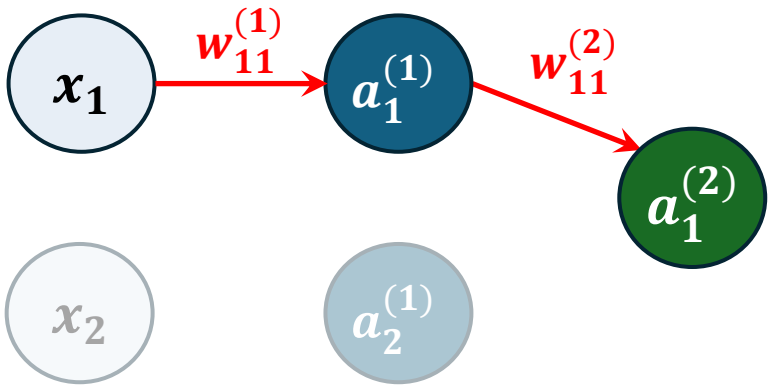
$$\frac{\partial \mathcal{L}}{\partial w_{11}^{(1)}} = \frac{\partial \mathcal{L}}{\partial a_1^{(2)}} \frac{\partial a_1^{(2)}}{\partial w_{11}^{(1)}}$$

$$\frac{\partial a_1^{(2)}}{\partial z_1^{(2)}} \frac{\partial z_1^{(2)}}{\partial w_{11}^{(1)}}$$

$$\frac{\partial z_1^{(2)}}{\partial a_1^{(1)}} \frac{\partial a_1^{(1)}}{\partial w_{11}^{(1)}}$$

$$\frac{\partial a_1^{(1)}}{\partial z_1^{(1)}} \frac{\partial z_1^{(1)}}{\partial w_{11}^{(1)}}$$

Chain rule again, just longer…



$$z_1^{(1)} = w_{11}^{(1)} x_1 + w_{12}^{(1)} x_2 + b_1^{(1)}$$

$$a_1^{(1)} = g^{(1)}(z_1^{(1)})$$

$$z_1^{(2)} = w_{11}^{(2)} a_1^{(1)} + w_{12}^{(2)} a_2^{(1)} + b_1^{(2)}$$

$$a_1^{(2)} = g^{(2)}(z_1^{(2)})$$

$$\frac{\partial \mathcal{L}}{\partial w_{11}^{(1)}} = \frac{\partial \mathcal{L}}{\partial a_1^{(2)}} \frac{\partial a_1^{(2)}}{\partial z_1^{(2)}} \frac{\partial z_1^{(2)}}{\partial a_1^{(1)}} \frac{\partial a_1^{(1)}}{\partial z_1^{(1)}} \frac{\partial z_1^{(1)}}{\partial w_{11}^{(1)}}$$

$g'(z_1^{(2)})$

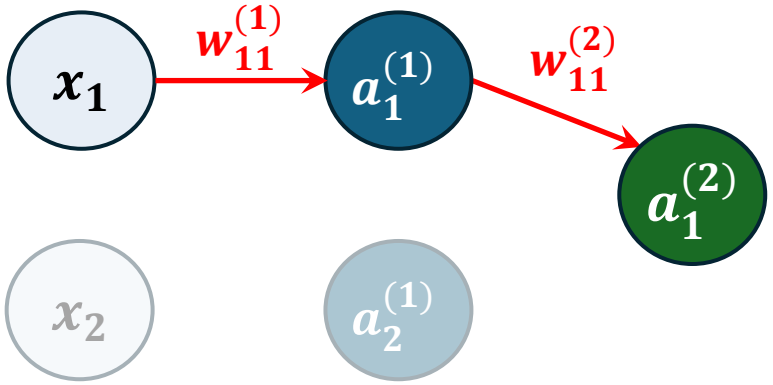
$w_{11}^{(2)}$ Known weight

$g'(z_1^{(1)})$

x_1 Known input

These are just the slopes of the activation functions

Chain rule again, just longer…



$$\frac{\partial \mathcal{L}}{\partial w_{11}^{(1)}} = \frac{\partial \mathcal{L}}{\partial a_1^{(2)}} \frac{\partial a_1^{(2)}}{\partial z_1^{(2)}} \frac{\partial z_1^{(2)}}{\partial a_1^{(1)}} \frac{\partial a_1^{(1)}}{\partial z_1^{(1)}} \frac{\partial z_1^{(1)}}{\partial w_{11}^{(1)}}$$

$$\frac{\partial \mathcal{L}}{\partial w_{11}^{(1)}} = \frac{\partial \mathcal{L}}{\partial a_1^{(2)}} g'(z_1^{(2)}) w_{11}^{(2)} g'(z_1^{(1)}) x_1$$

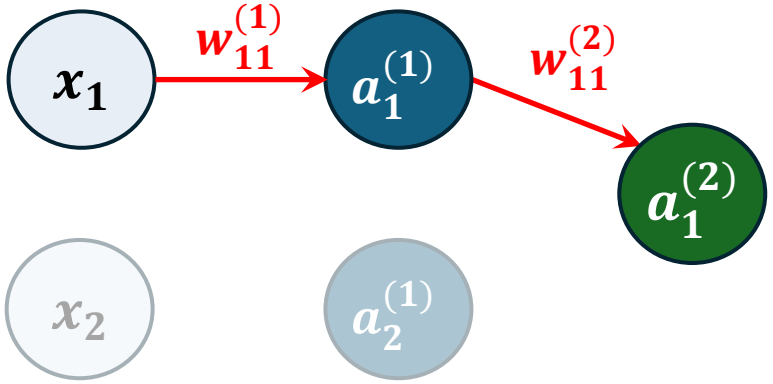
This expression is getting a bit long, so let's introduce an **"error signal" on the output:**

$$\frac{\partial \mathcal{L}}{\partial z_1^{(2)}} = \delta_1^{(2)} = \frac{\partial \mathcal{L}}{\partial a_1^{(2)}} \frac{\partial a_1^{(2)}}{\partial z_1^{(2)}} = \frac{\partial \mathcal{L}}{\partial a_1^{(2)}} g'(z_1^{(2)})$$

$$\frac{\partial \mathcal{L}}{\partial w_{11}^{(1)}} = \delta_1^{(2)} w_{11}^{(2)} g'(z_1^{(1)}) x_1$$

$\delta_1^{(2)}$ tells us how much the loss would change if we slightly changed the last neuron's input, $z_1^{(2)}$

Final expression for the loss of a weight



$$\frac{\partial \mathcal{L}}{\partial w_{11}^{(1)}} = \delta_1^{(2)} w_{11}^{(2)} g'(z_1^{(1)}) x_1$$

Similarly, let's define an **error signal on the hidden neuron:**

$$\delta_1^{(1)} = \frac{\partial \mathcal{L}}{\partial z_1^{(1)}}$$

The loss

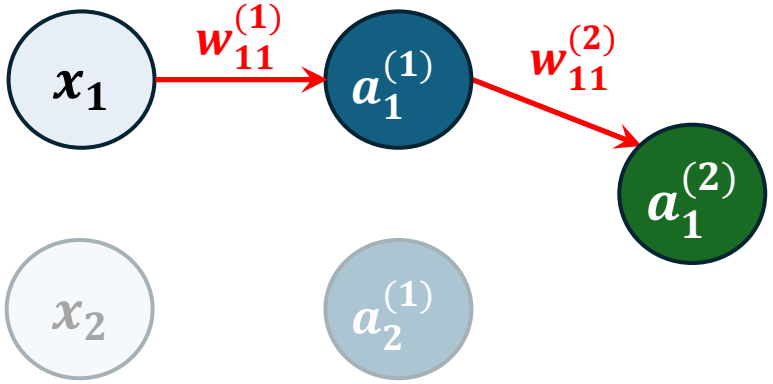
The neuron's input

How much did this neuron contribute to the final prediction error

$$\frac{\partial \mathcal{L}}{\partial z_1^{(1)}} = \frac{\partial \mathcal{L}}{\partial z_1^{(2)}} \frac{\partial z_1^{(2)}}{\partial a_1^{(1)}} \frac{\partial a_1^{(1)}}{\partial z_1^{(1)}} = \delta_1^{(2)} w_{11}^{(2)} g'(z_1^{(1)})$$

$$w_{11}^{(1)} \rightarrow z_1^{(1)} \rightarrow a_1^{(1)} \rightarrow z_1^{(2)} \rightarrow a_1^{(2)} \rightarrow \mathcal{L}$$

Final expression for the loss of a weight



$$\frac{\partial \mathcal{L}}{\partial w_{11}^{(1)}} = \delta_1^{(2)} w_{11}^{(2)} g'(z_1^{(1)}) x_1$$

Similarly, let's define an **error signal on the hidden neuron:**

$$\delta_1^{(1)} = \frac{\partial \mathcal{L}}{\partial z_1^{(1)}}$$

Putting it all together:

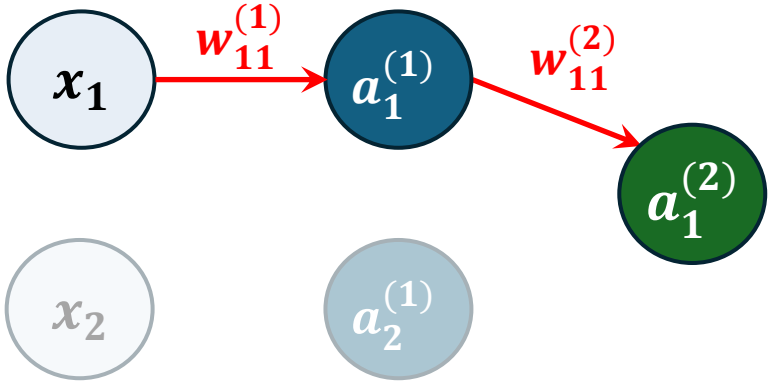
$$\frac{\partial \mathcal{L}}{\partial w_{11}^{(1)}} = \delta_1^{(1)} x_1$$

$$\frac{\partial \mathcal{L}}{\partial z_1^{(1)}} = \frac{\partial \mathcal{L}}{\partial z_1^{(2)}} \frac{\partial z_1^{(2)}}{\partial a_1^{(1)}} \frac{\partial a_1^{(1)}}{\partial z_1^{(1)}} = \delta_1^{(2)} w_{11}^{(2)} g'(z_1^{(1)})$$

Weight gradient = Error at receiving neuron × input entering the weight

$$w_{11}^{(1)} \rightarrow z_1^{(1)} \rightarrow a_1^{(1)} \rightarrow z_1^{(2)} \rightarrow a_1^{(2)} \rightarrow \mathcal{L}$$

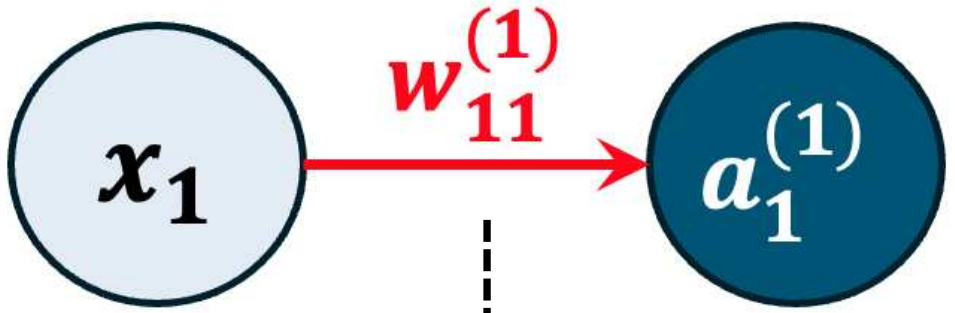
Final expression for the loss of a weight



Putting it all together:

$$\frac{\partial \mathcal{L}}{\partial w_{11}^{(1)}} = \delta_1^{(1)} x_1$$

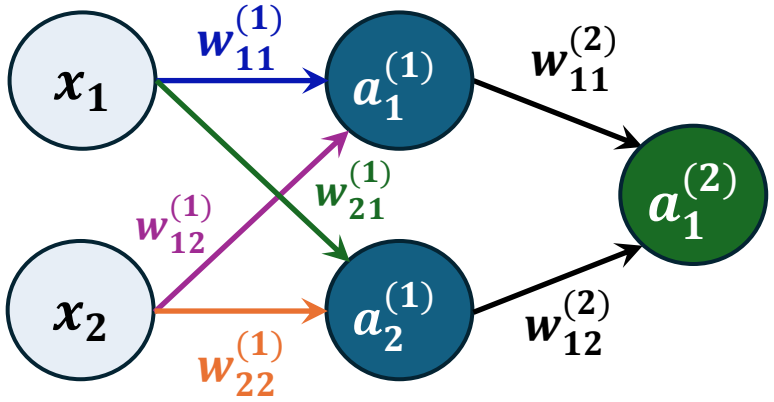
Weight gradient = Error at receiving neuron × input entering the weight



x_1: how big of a value gets fed to that neuron

$\delta_1^{(1)}$: how much effect this neuron has on the prediction error

Other weights get similar expressions



$$\frac{\partial \mathcal{L}}{\partial w_{11}^{(1)}} = \delta_1^{(1)} x_1$$

$$\frac{\partial \mathcal{L}}{\partial w_{12}^{(1)}} = \delta_1^{(1)} x_2$$

$$\frac{\partial \mathcal{L}}{\partial w_{21}^{(1)}} = \delta_2^{(1)} x_1$$

$$\frac{\partial \mathcal{L}}{\partial w_{22}^{(1)}} = \delta_2^{(1)} x_2$$

In general, for any weight entering the hidden layer:

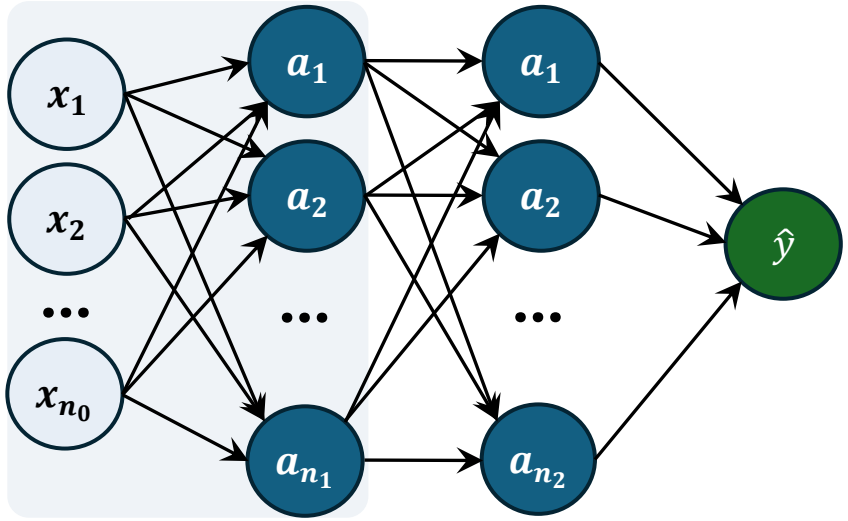
$$\frac{\partial \mathcal{L}}{\partial w_{ji}^{(1)}} = \delta_j^{(1)} x_i$$

Where the error signal is:

$$\delta_j^{(1)} = \frac{\partial \mathcal{L}}{\partial z_j^{(1)}}$$

Each weight's gradient is the **signal entering that weight**, multiplied by **how strongly the receiving neuron contributes to the final loss**.

Generalizing to multiple hidden layers



$$\frac{\partial \mathcal{L}}{\partial w_{ji}^{(1)}} = \delta_j^{(1)} x_i$$

The general version for any weight connecting neurons in subsequent layers:

$$\frac{\partial \mathcal{L}}{\partial w_{ji}^{(l)}} = \delta_j^{(l)} a_i^{(l-1)}$$

Error signal: how sensitive the loss is to that neuron's pre-activation value

$$\delta_j^{(l)} = \frac{\partial \mathcal{L}}{\partial z_j^{(l)}}$$

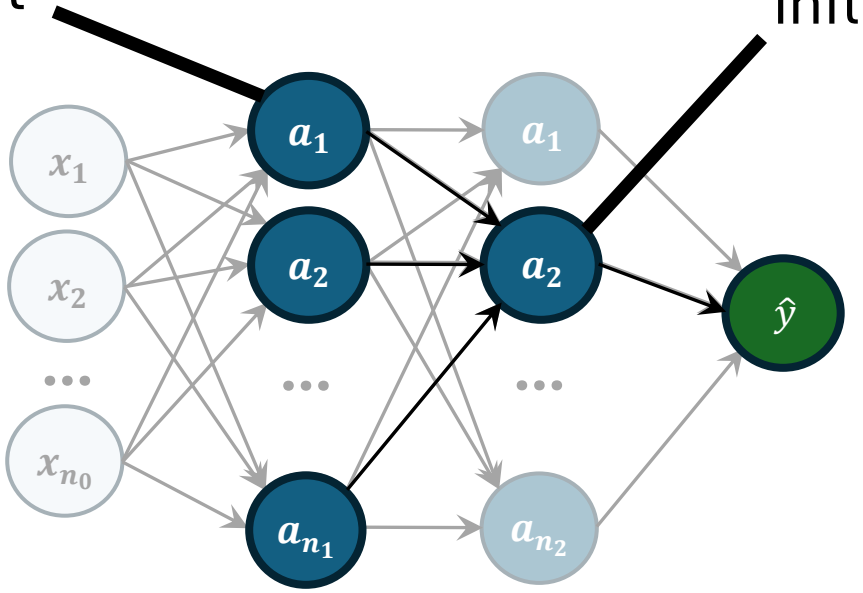
$$\delta_j^{(l)} = g'\left(z_j^{(l)}\right) \sum_k w_{kj}^{(l+1)} \delta_k^{(l+1)}$$

Error signal at a neuron = the errors *from the next layer*, weighted by the connections to those neurons, then multiplied by the derivative of the current neuron's activation.

Backpropagation of error signals

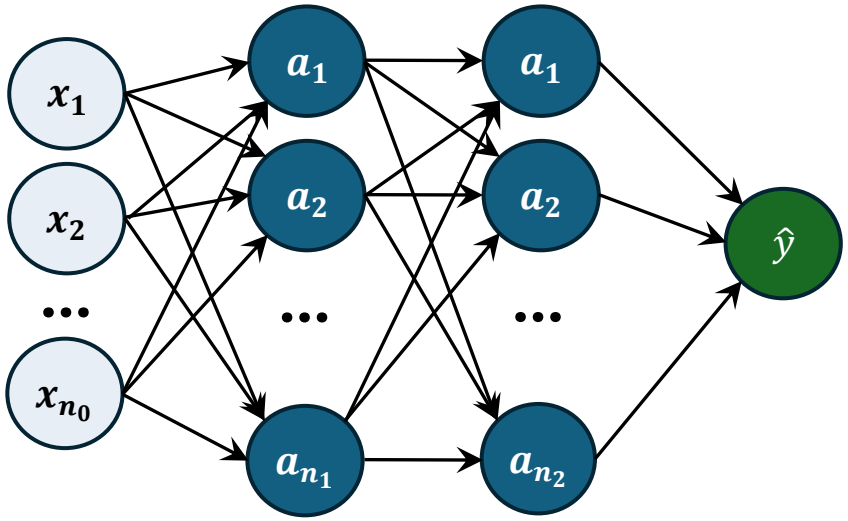
Then these are more likely to also have a large influence on the output

If this one has a large influence on the output



Backpropagation on a model with L layers

Forward pass



Backward pass

Make a prediction and compute the error signal at the output:

$$\delta_j^{(L)} = \frac{\partial \mathcal{L}}{\partial \hat{y}}$$

This usually simplifies. For example, MSE loss:

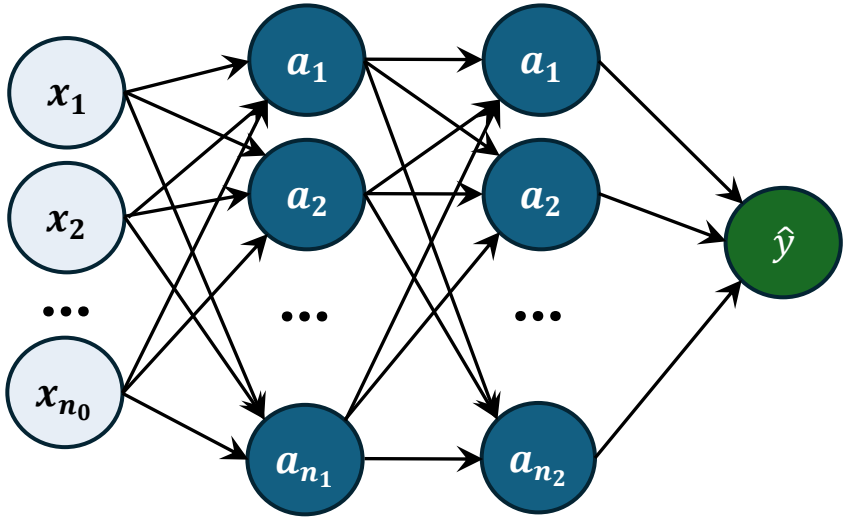
$$\mathcal{L} = (\hat{y} - y)^2 \rightarrow \partial \mathcal{L} / \partial \hat{y} = 2(\hat{y} - y)$$

Then we **propagate the error backward:**

$$\delta_j^{(l)} = g'\left(z_j^{(l)}\right) \sum_k w_{kj}^{(l+1)} \delta_k^{(l+1)}$$

Do this over and over until we reach the inputs, computing all error signals along the way

Backpropagation on a model with L layers



Backward pass

Use each error signal to calculate parameter gradients:

$$\frac{\partial \mathcal{L}}{\partial w_{ji}^{(l)}} = \delta_j^{(l)} a_i^{(l-1)}$$

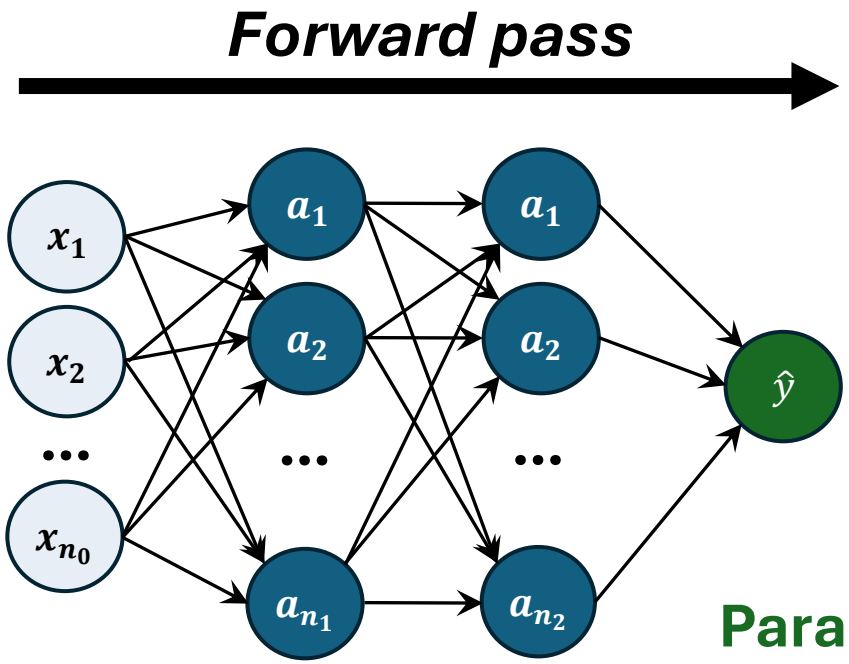
$$\frac{\partial \mathcal{L}}{\partial b_j^{(l)}} = \delta_j^{(l)}$$

Then we **propagate the error backward:**

$$\delta_j^{(l)} = g'\left(z_j^{(l)}\right) \sum_k w_{kj}^{(l+1)} \delta_k^{(l+1)}$$

Do this over and over until we reach the inputs, computing all error signals along the way

Backpropagation on a model with L layers



$$\frac{\partial \mathcal{L}}{\partial w_{ji}^{(l)}} = \delta_j^{(l)} a_i^{(l-1)}$$

$$\frac{\partial \mathcal{L}}{\partial b_j^{(l)}} = \delta_j^{(l)}$$

Parameter gradients are then used in gradient descent:

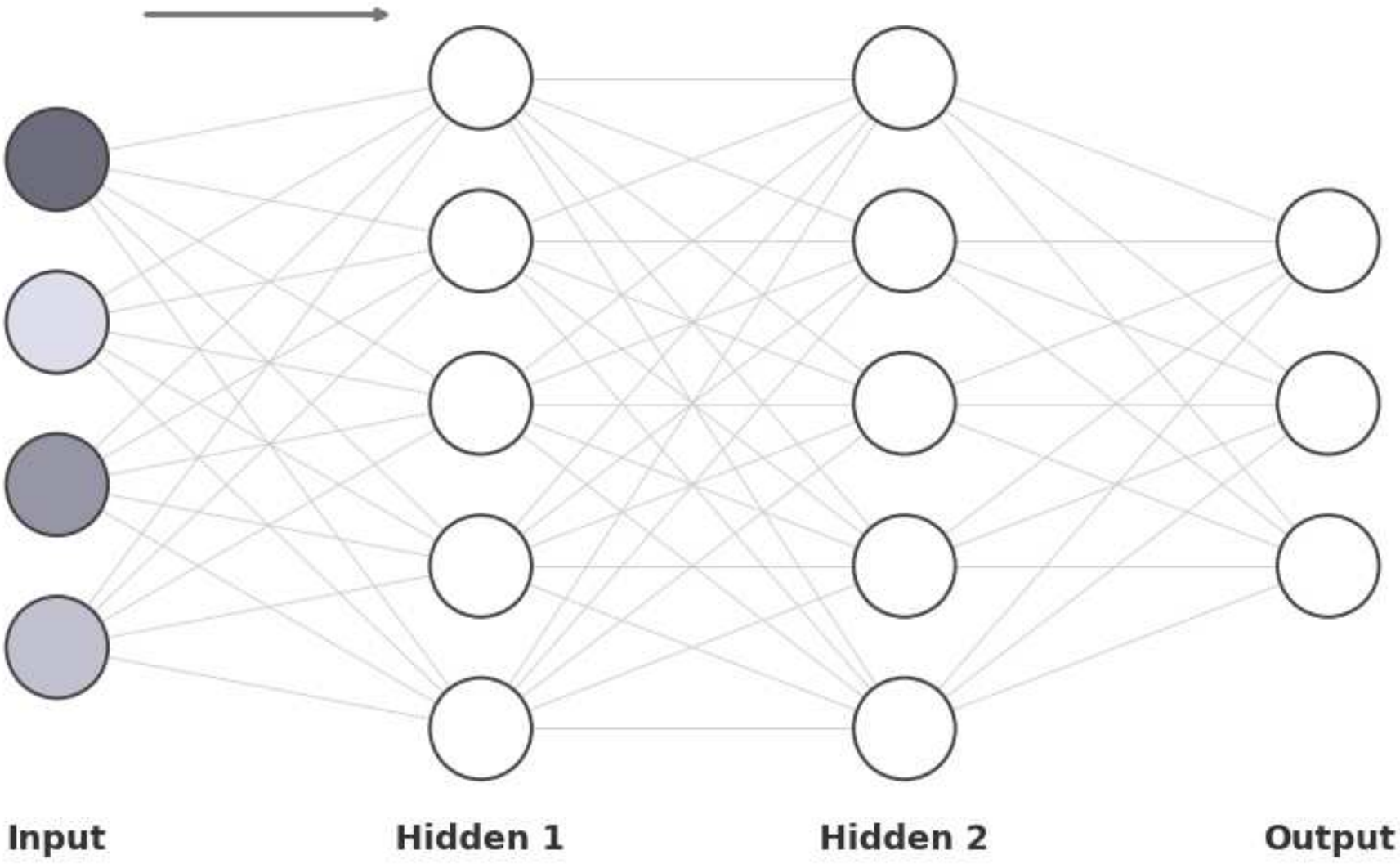
$$\text{New } w_{ji}^{(l)} = \text{Old } w_{ji}^{(l)} - \eta \left(\frac{d\mathcal{L}}{dw_{ji}^{(l)}} \right)$$

$$\text{New } b_j^{(l)} = \text{Old } b_j^{(l)} - \eta \left(\frac{d\mathcal{L}}{db_j^{(l)}} \right)$$

Do another forward pass with these updated params, backprop, and repeat!

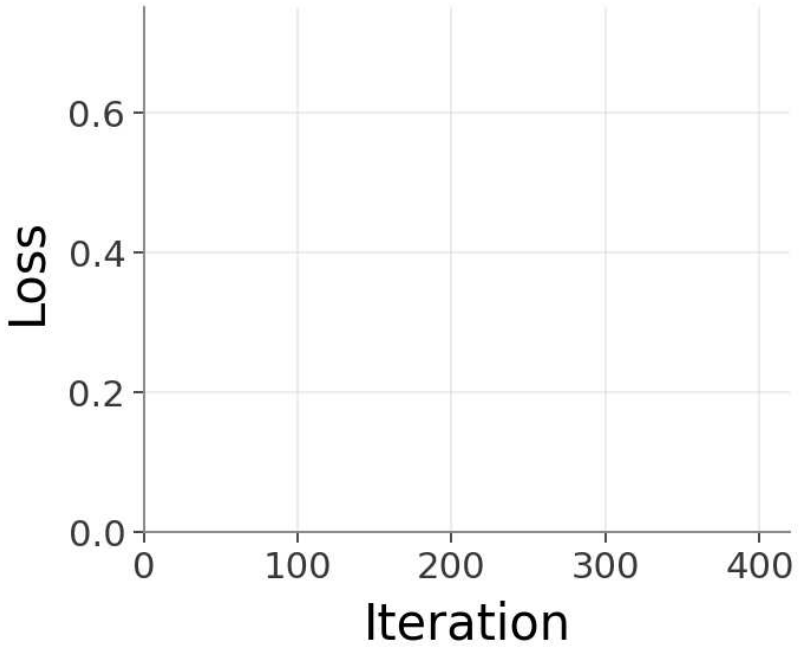
Backpropagation on a model with L layers

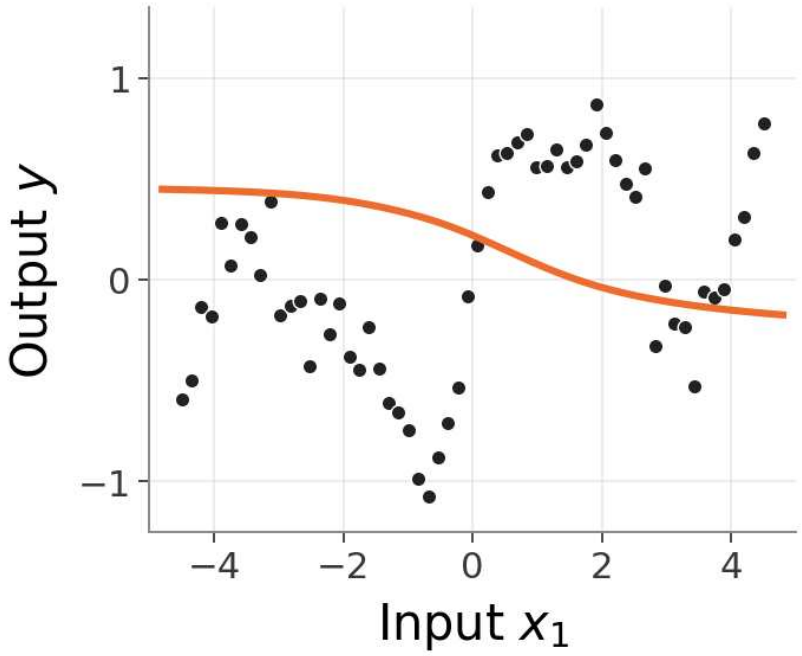
Courtesy of Claude

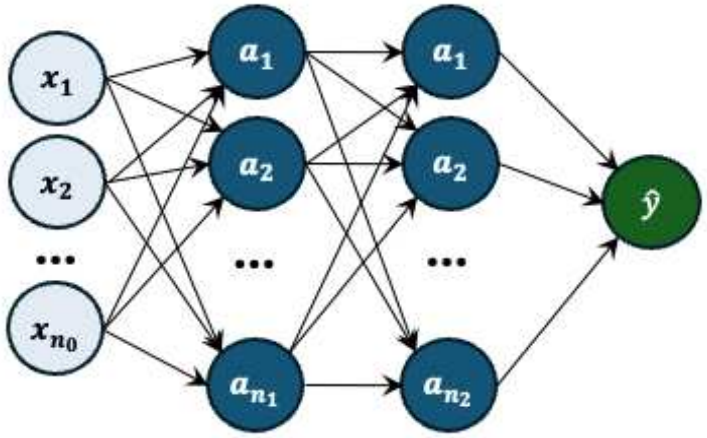


Questions?

The training process







Each pass through the “loop” is one iteration

Initialize parameters

↓

Forward pass

↓

Compute loss

↓

Backpropagate

↓

Update parameters

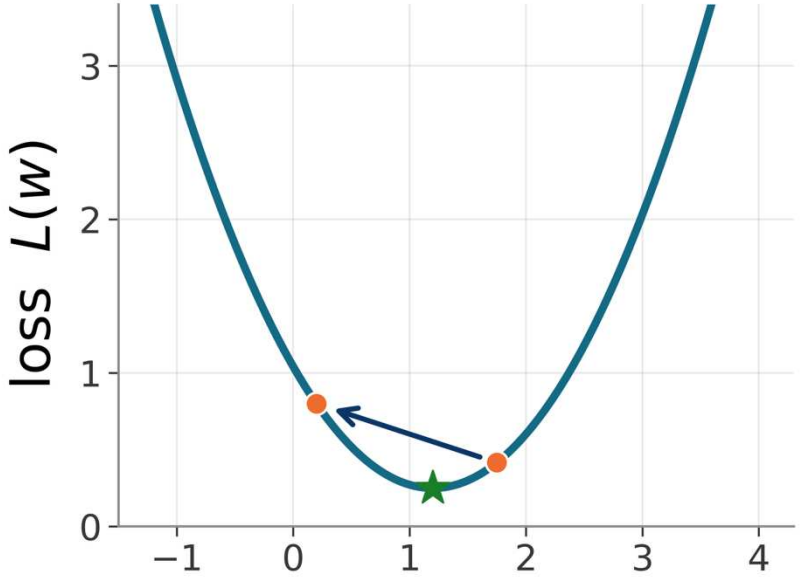
Repeat

Loss landscapes and the learning rate



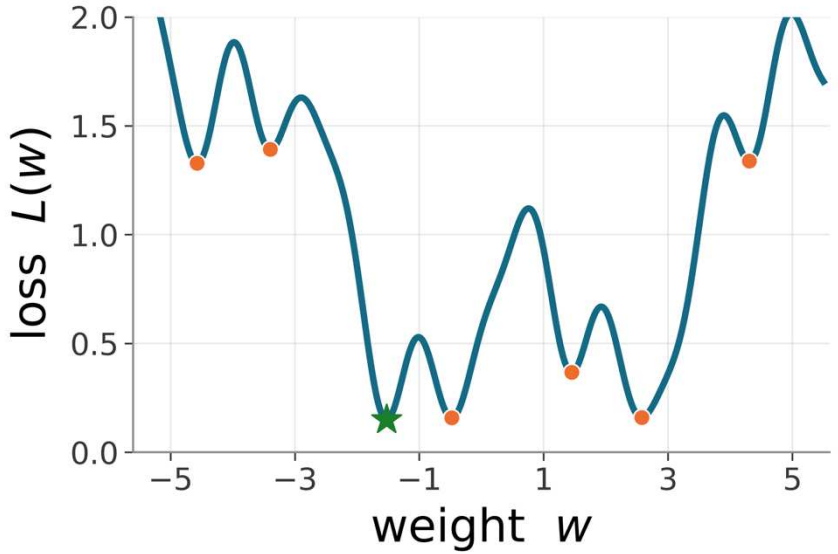
Why does loss sometimes increase?

Neural networks have non-convex loss landscapes



$$w' = w - \eta\left(\frac{d\mathcal{L}}{dw}\right)$$

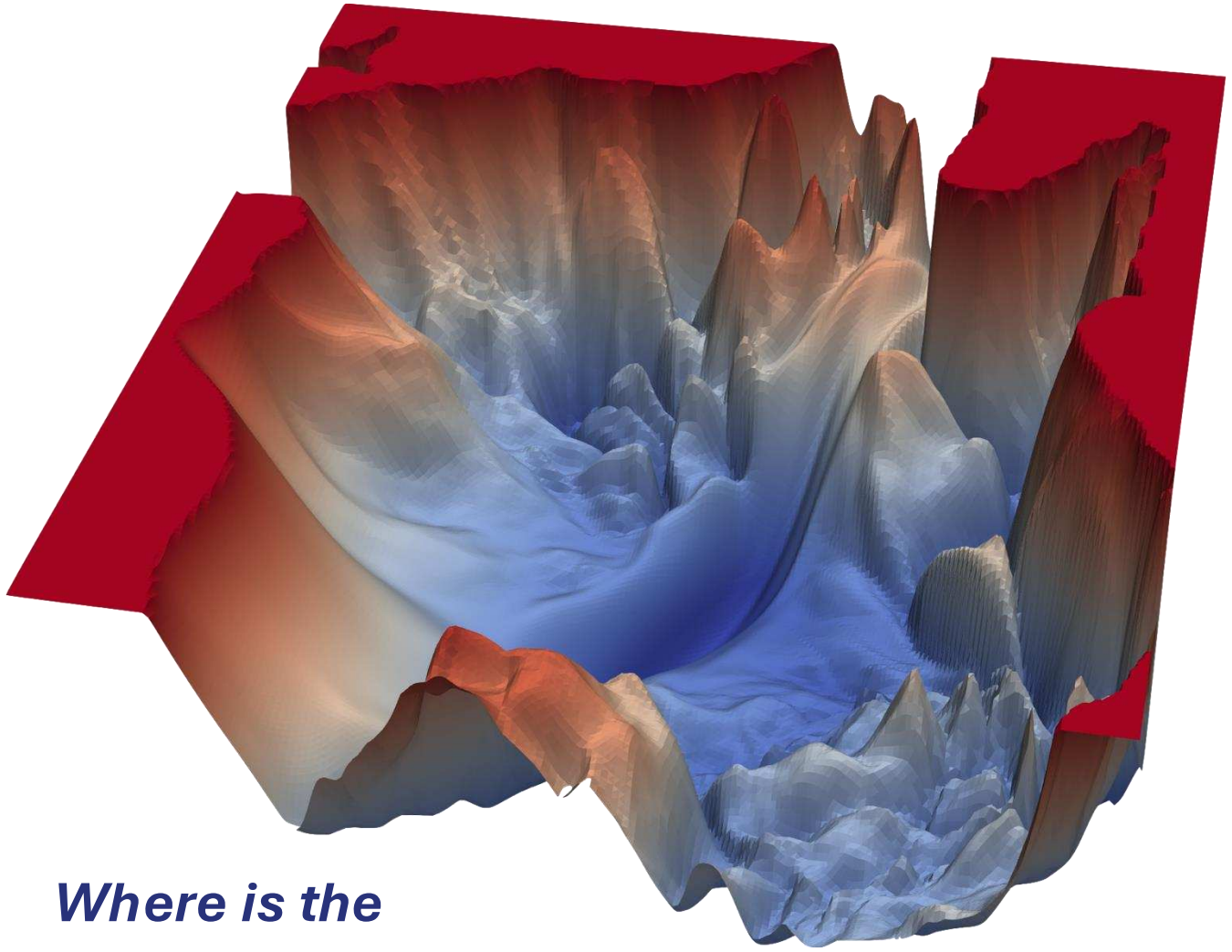
With too large a learning rate, we can overshoot the minimum



Too large a rate → overshoot min

Too small a rate → get "stuck"

Loss landscapes and the learning rate



Where is the global min?

Visualizing the Loss Landscape of Neural Nets

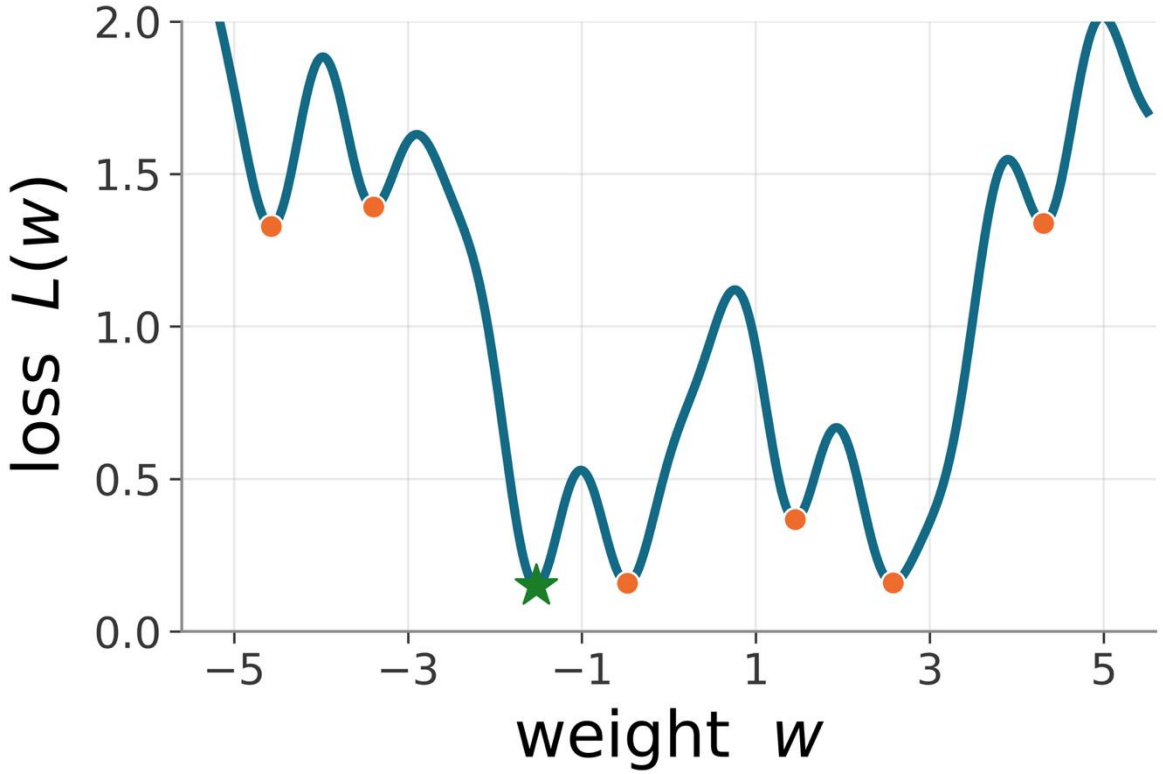
Hao Li[1], **Zheng Xu**[1], **Gavin Taylor**[2], **Christoph Studer**[3], **Tom Goldstein**[1]
[1]University of Maryland, College Park [2]United States Naval Academy [3]Cornell University
{haoli,xuzh,tomg}@cs.umd.edu, taylor@usna.edu, studer@cornell.edu

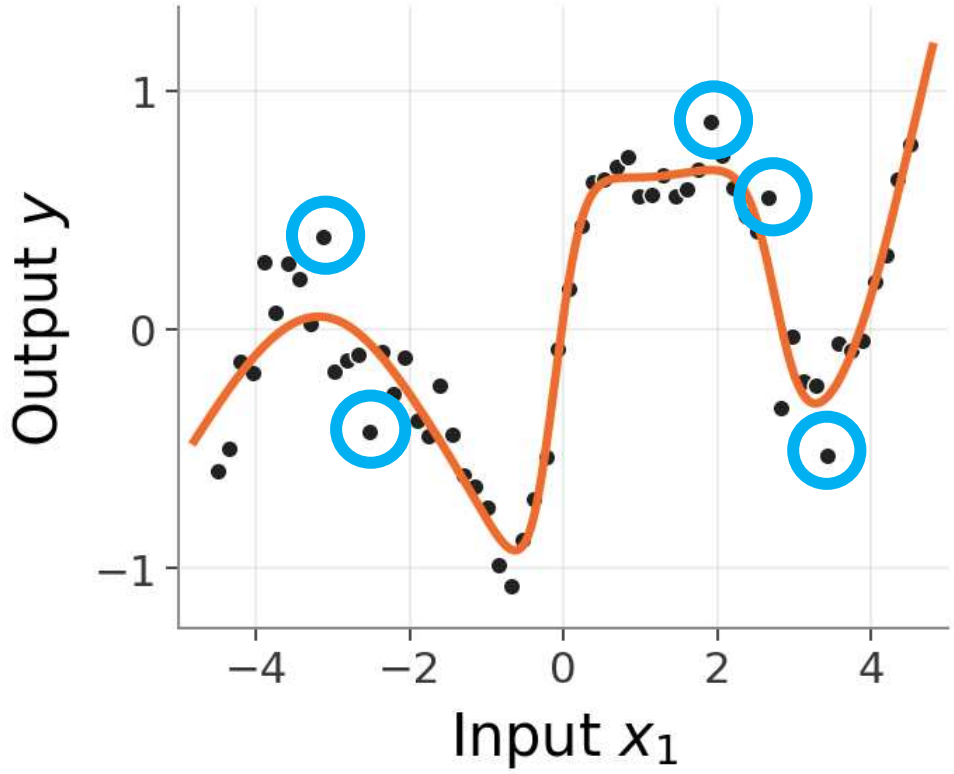
Deep neural networks have **many weights** that we need to optimize, and without regularization*, the loss is **highly non-convex**

*We'll cover this next time

One solution: Stochastic gradient descent

This loss is computed over the **entire training dataset**

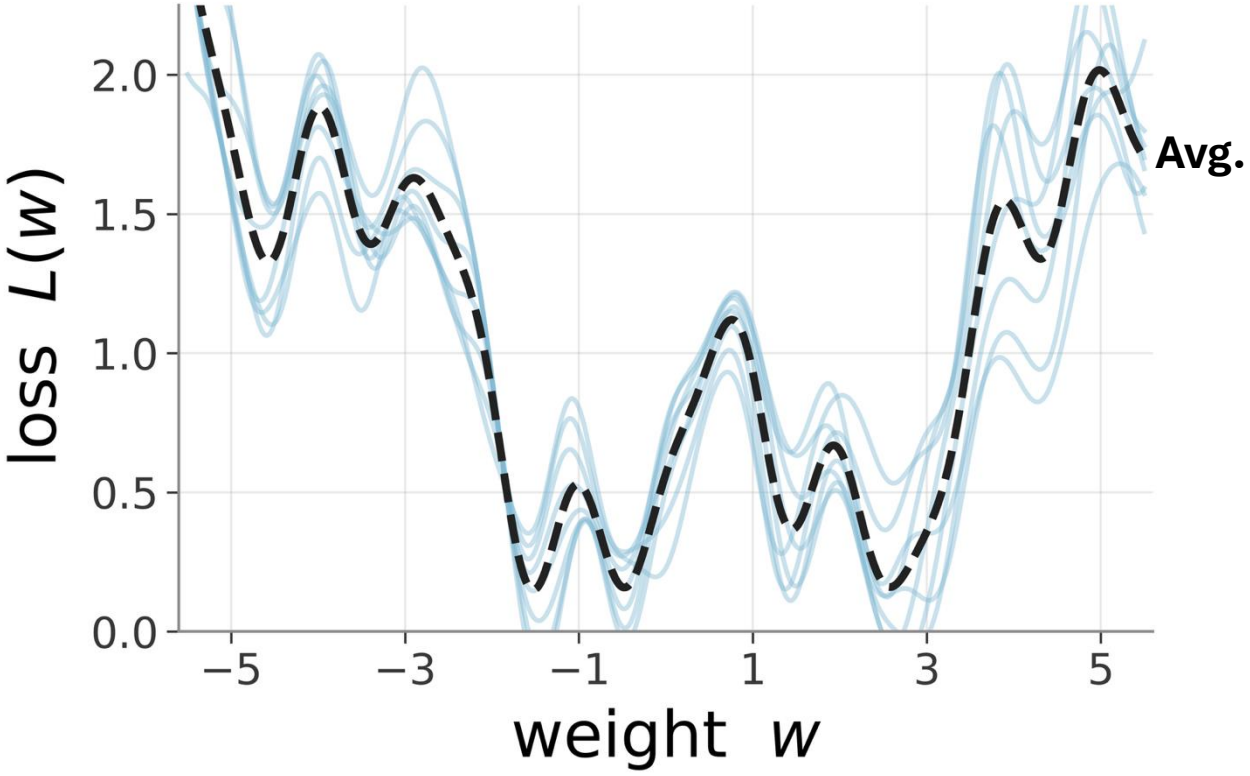




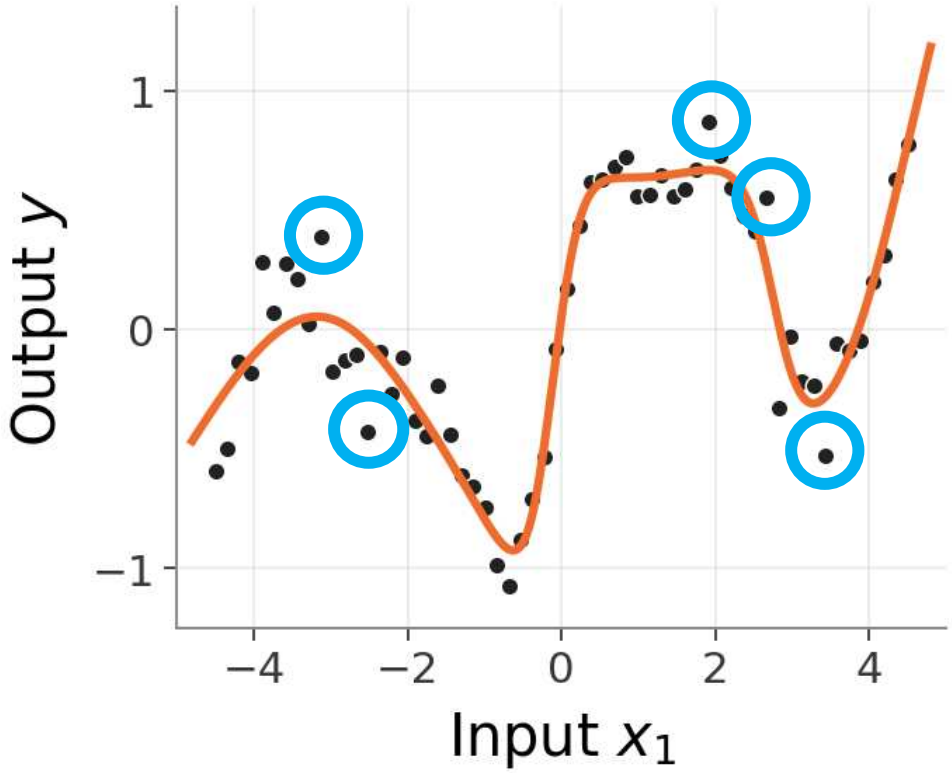
What if we instead computed loss, and made updates, **one sample at a time?**

One solution: Stochastic gradient descent

Each blue curve is the loss *vs.* weight **for one sample**



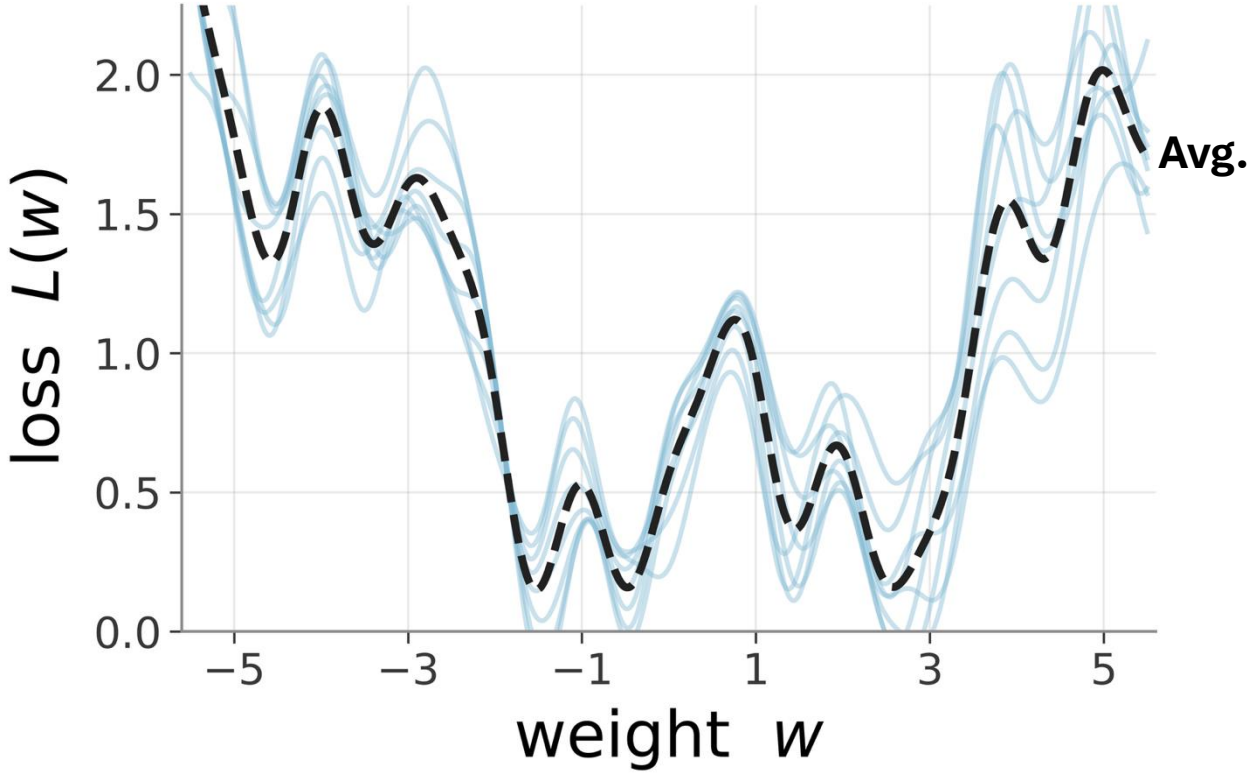
In reality, it would look even more diverse than this

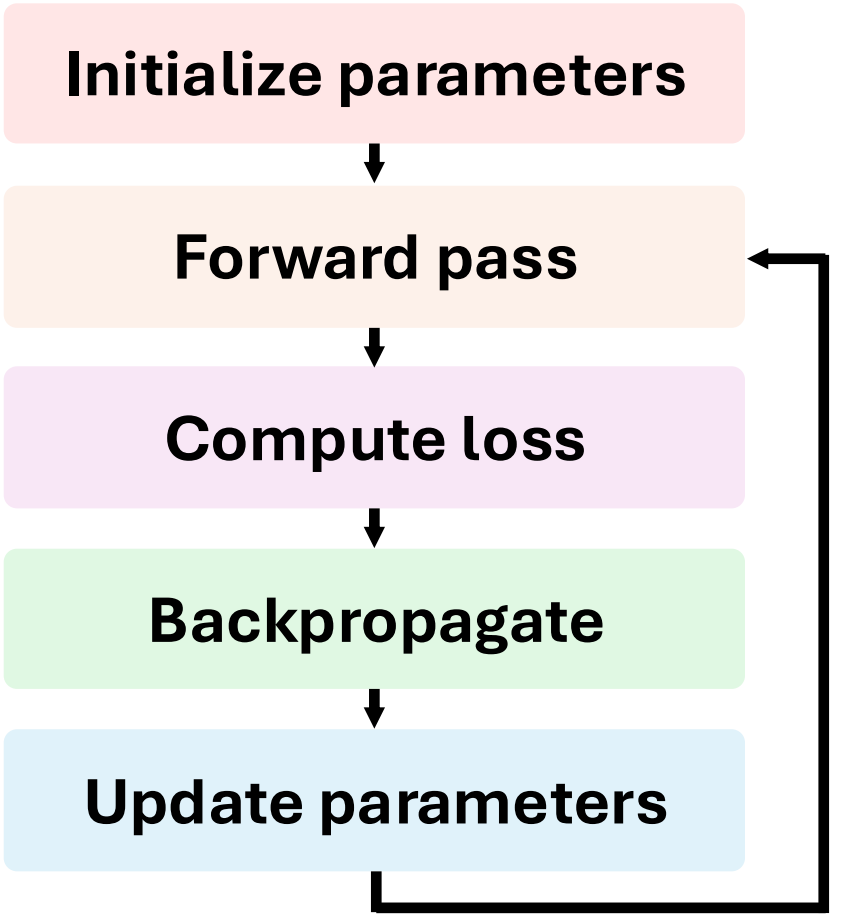


What if we instead computed loss, and made updates, **one sample at a time?**

One solution: Stochastic gradient descent

Each blue curve is the loss *vs.* weight **for one sample**

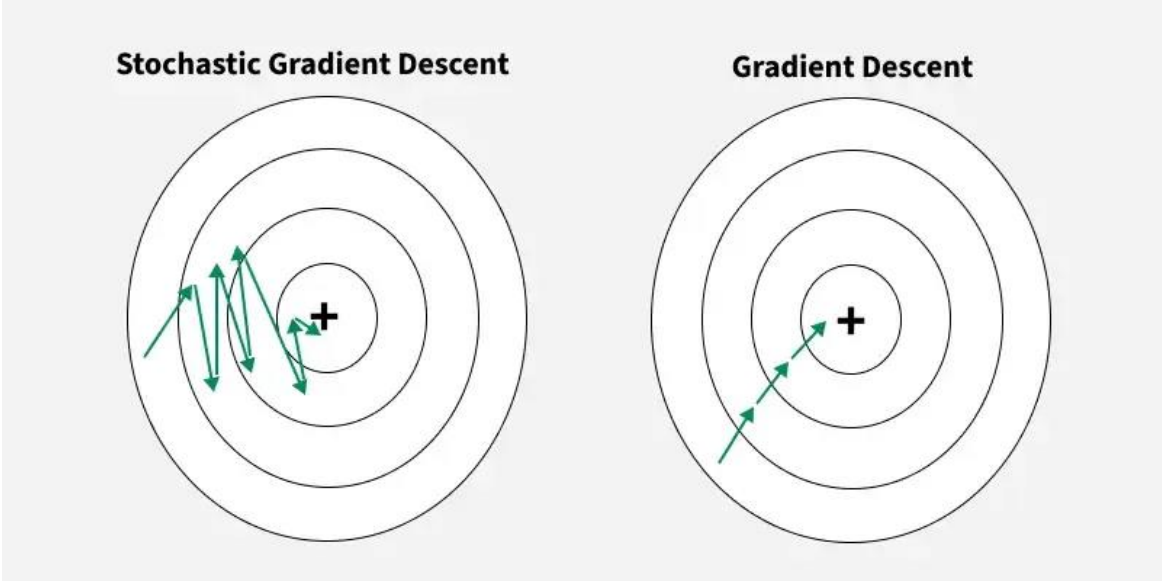


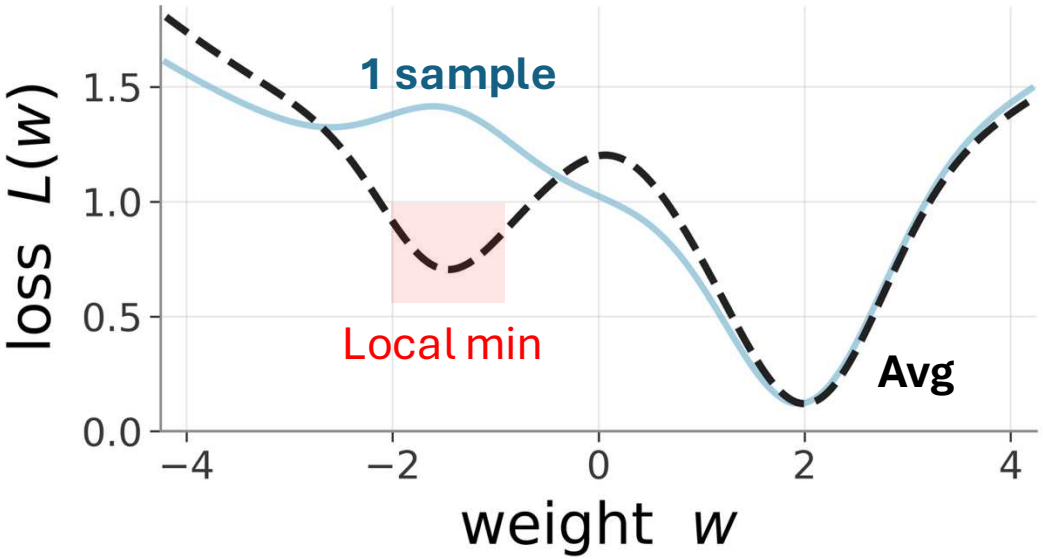


Repeat one iteration per sample. This is ***stochastic gradient descent (SGD)***

One solution: Stochastic gradient descent

SGD is more **"noisy"** than full gradient descent





The noisiness can help us escape local minima!

Initialize parameters

↓

Forward pass

↓

Compute loss

↓

Backpropagate

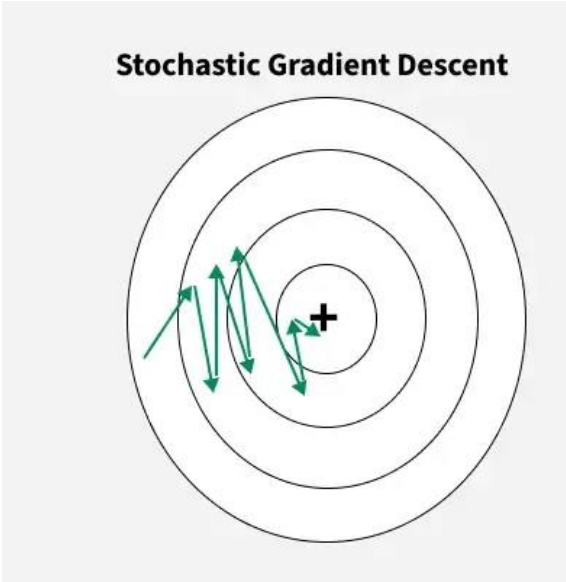
↓

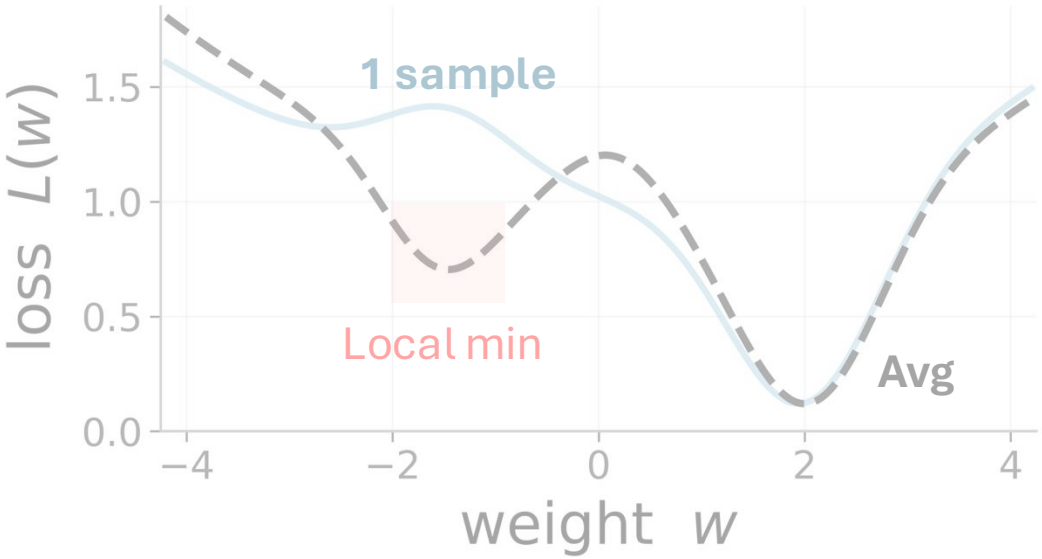
Update parameters

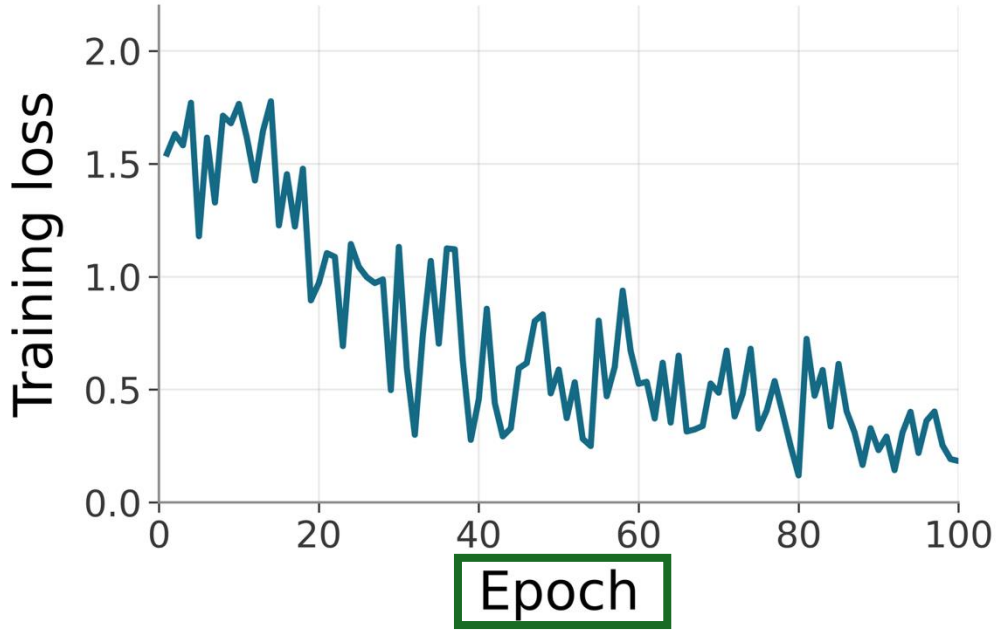
Repeat one iteration per sample. This is ***stochastic gradient descent (SGD)***

A better solution: Mini batch GD

But SGD can sometimes be *too* noisy

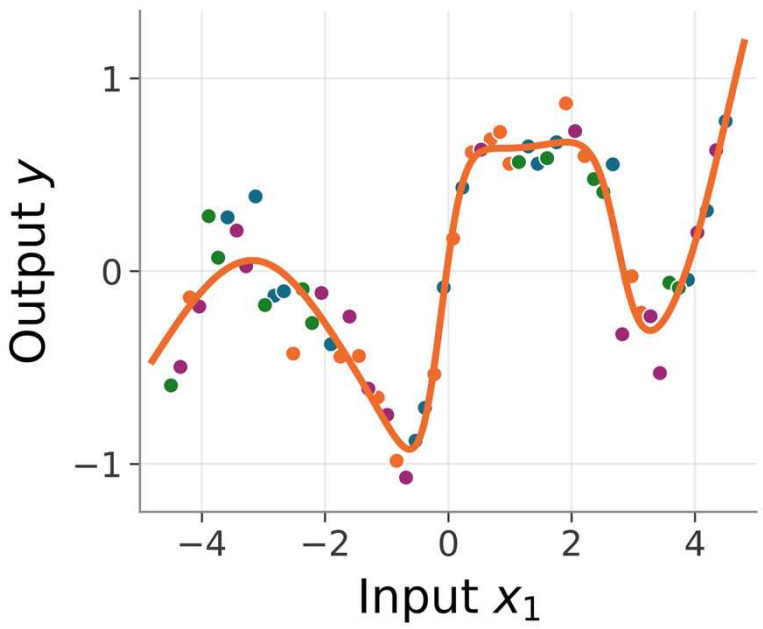




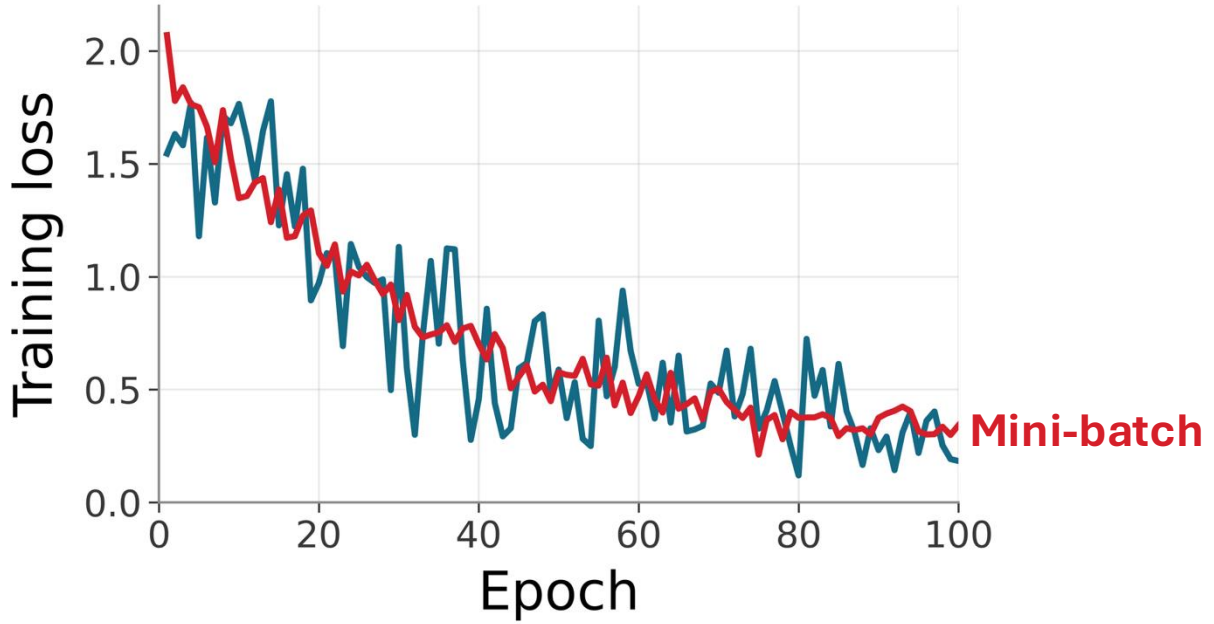


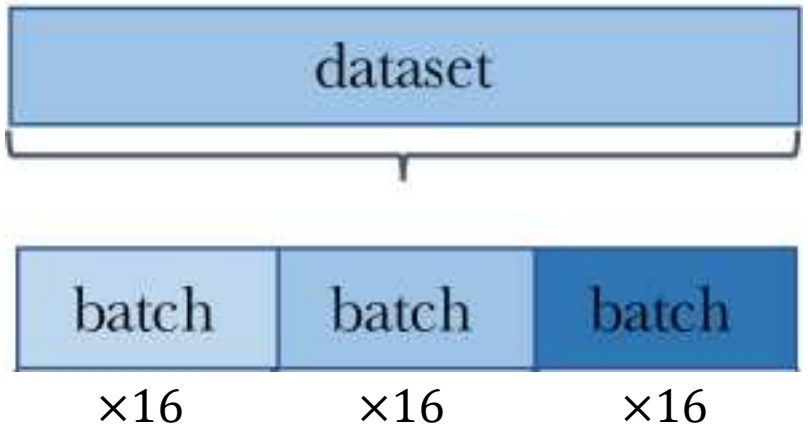
1 epoch = 1 complete pass through the training dataset. In SGD, there is N_{samples} weight updates per epoch.

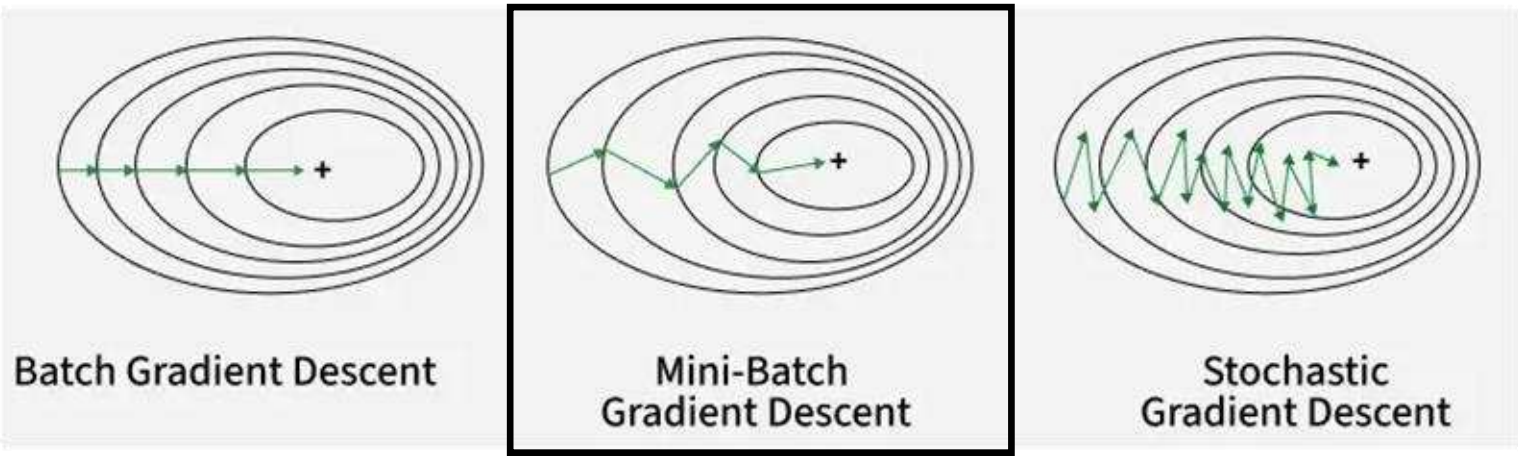
A better solution: Mini batch GD



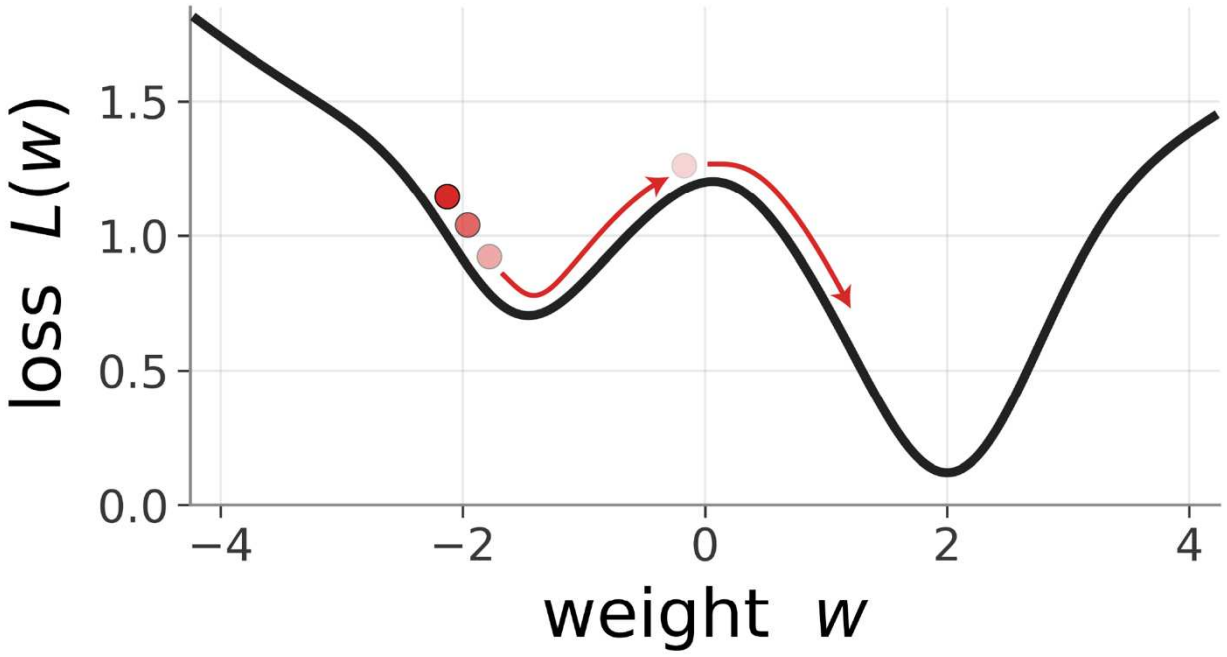
Compute loss and updates weights based on ***batches*** (usually each with **16-256 samples**)







GD can also benefit from *momentum*



$$w_t = w_{t-1} - \eta\left(\frac{d\mathcal{L}}{dw}\right)$$

Without momentum: Update is based on current gradient only

With momentum: Update is based on a **running avg of recent gradients**

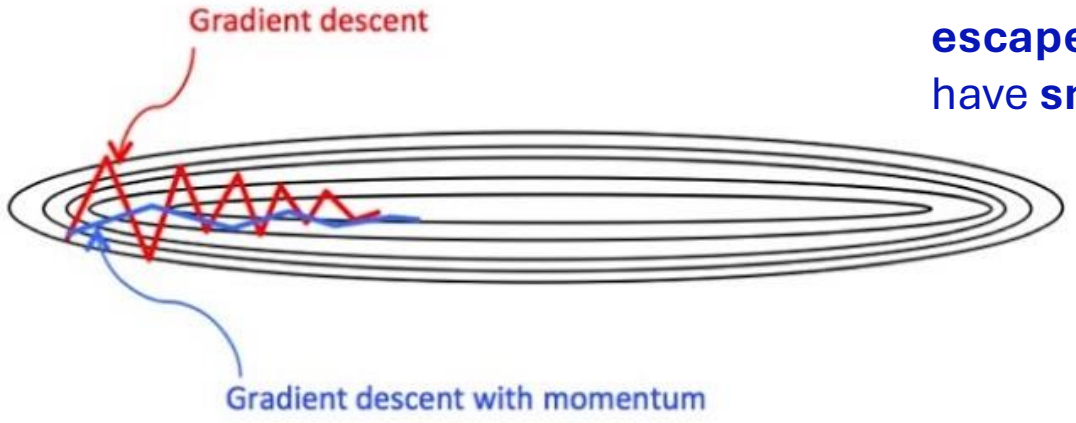
$$w_t = w_{t-1} - \eta v_t$$

$$v_t = \beta v_{t-1} + (1-\beta)\left(\frac{d\mathcal{L}}{dw}\right)$$

Controls how much we are influenced by past gradients

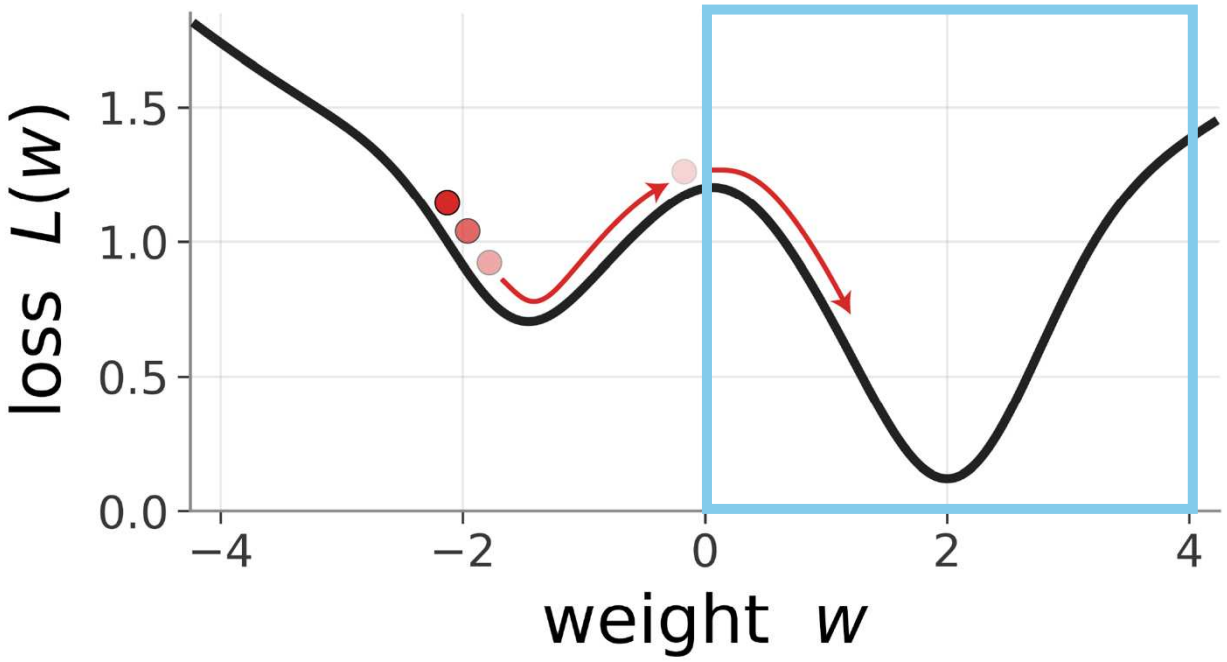
$\beta \sim 1$ → strong influence

$\beta \sim 0$ → weak influence



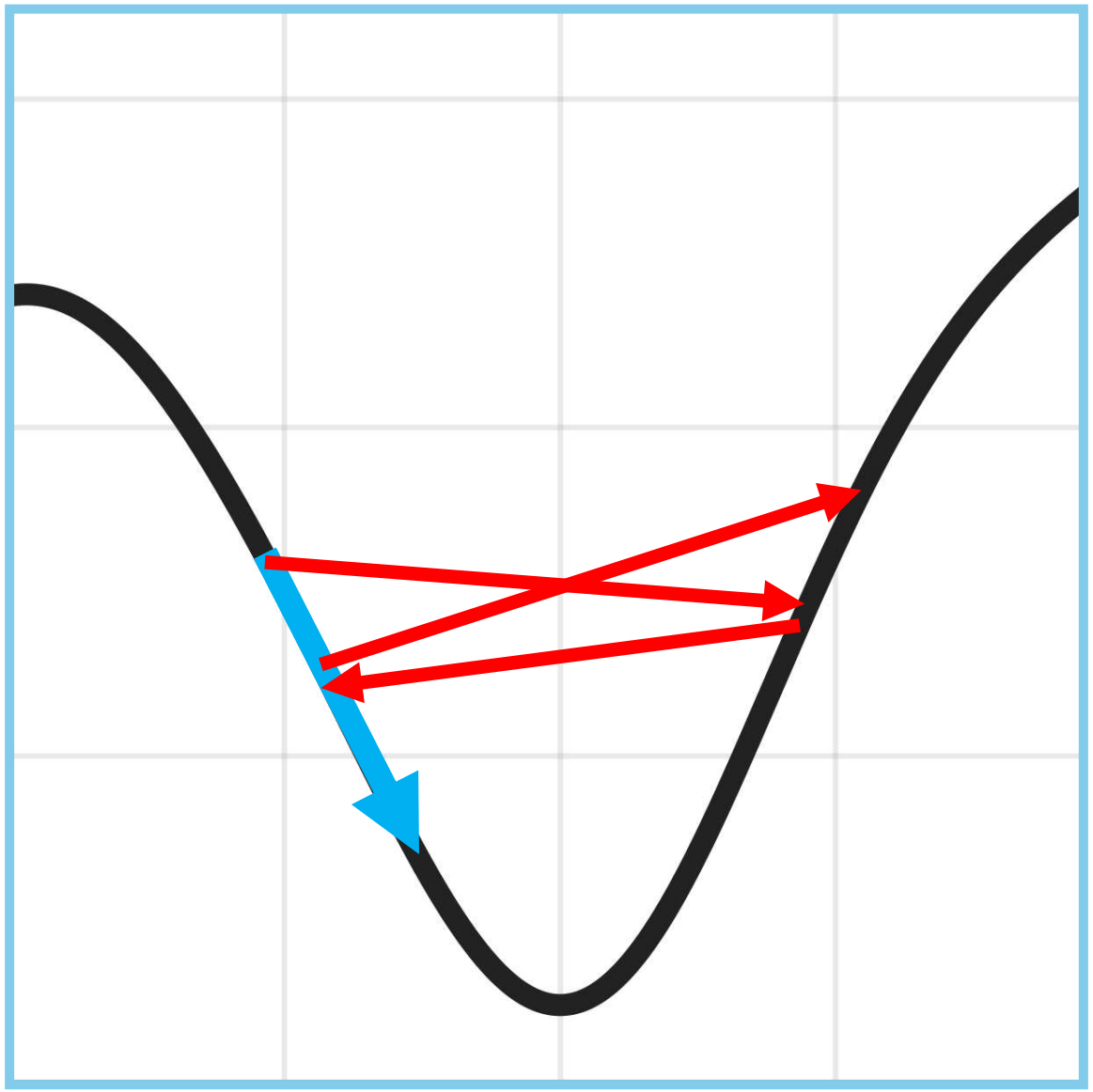
Momentum helps us **escape local minima** and have **smoother loss**

One more trick: An adaptive learning rate



When we are far from the minimum, a high learning rate is beneficial to "quickly descend" downward

But making the learning rate too large will cause wild fluctuations



One more trick: An adaptive learning rate

RMSProp: Root-mean squared propagation

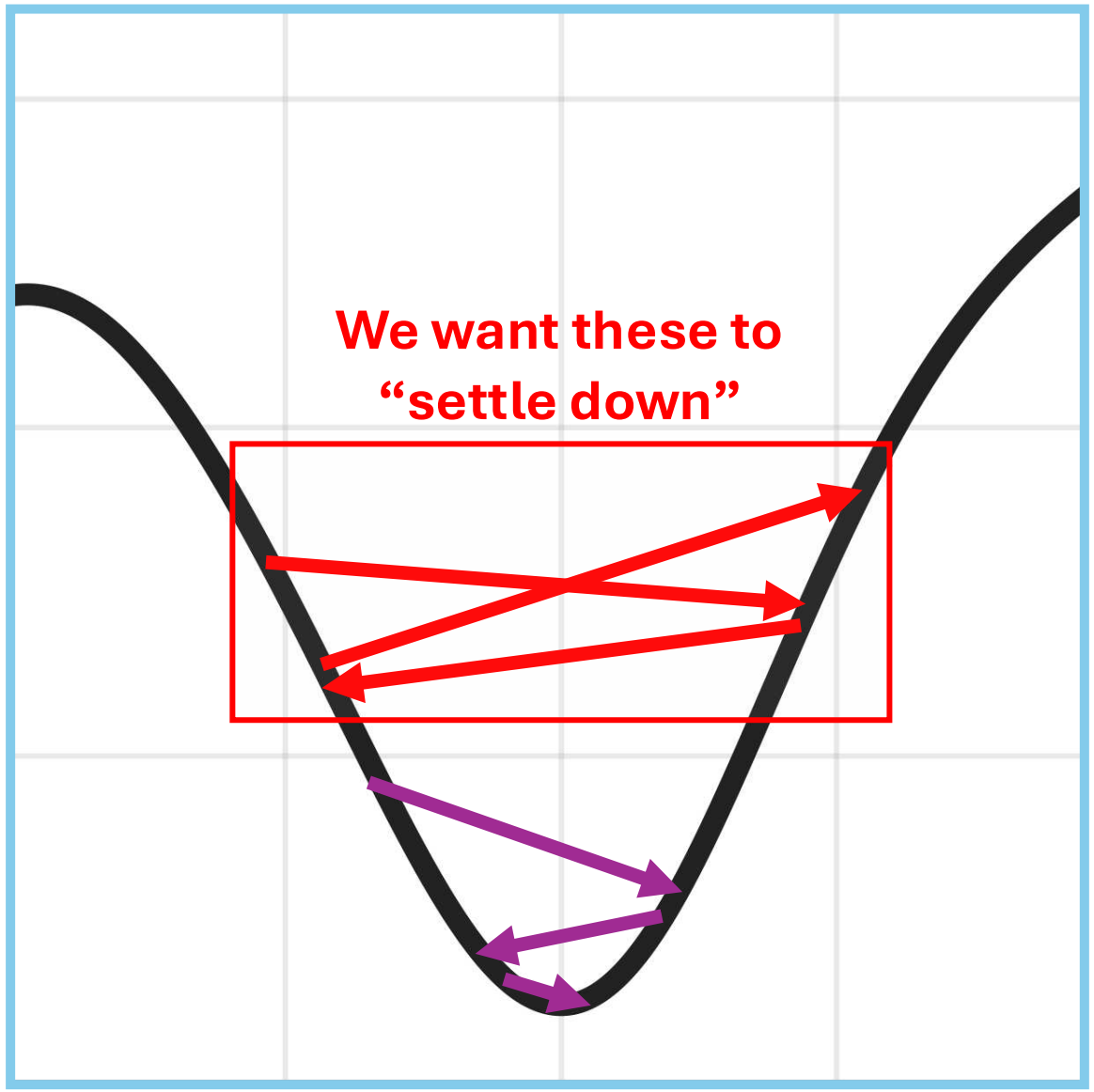
→ Parameters with repeatedly large gradients get smaller updates

$$w_t = w_{t-1} - \frac{\eta}{\sqrt{s_t} + \epsilon}\left(\frac{d\mathcal{L}}{dw}\right)$$

This is a running avg of the squares of recent gradients

This is a small factor to prevent division by zero

Large gradients → divide by larger $\sqrt{s_t}$
→ updates are made smaller



Adam optimizer is the go-to method

Published as a conference paper at ICLR 2015

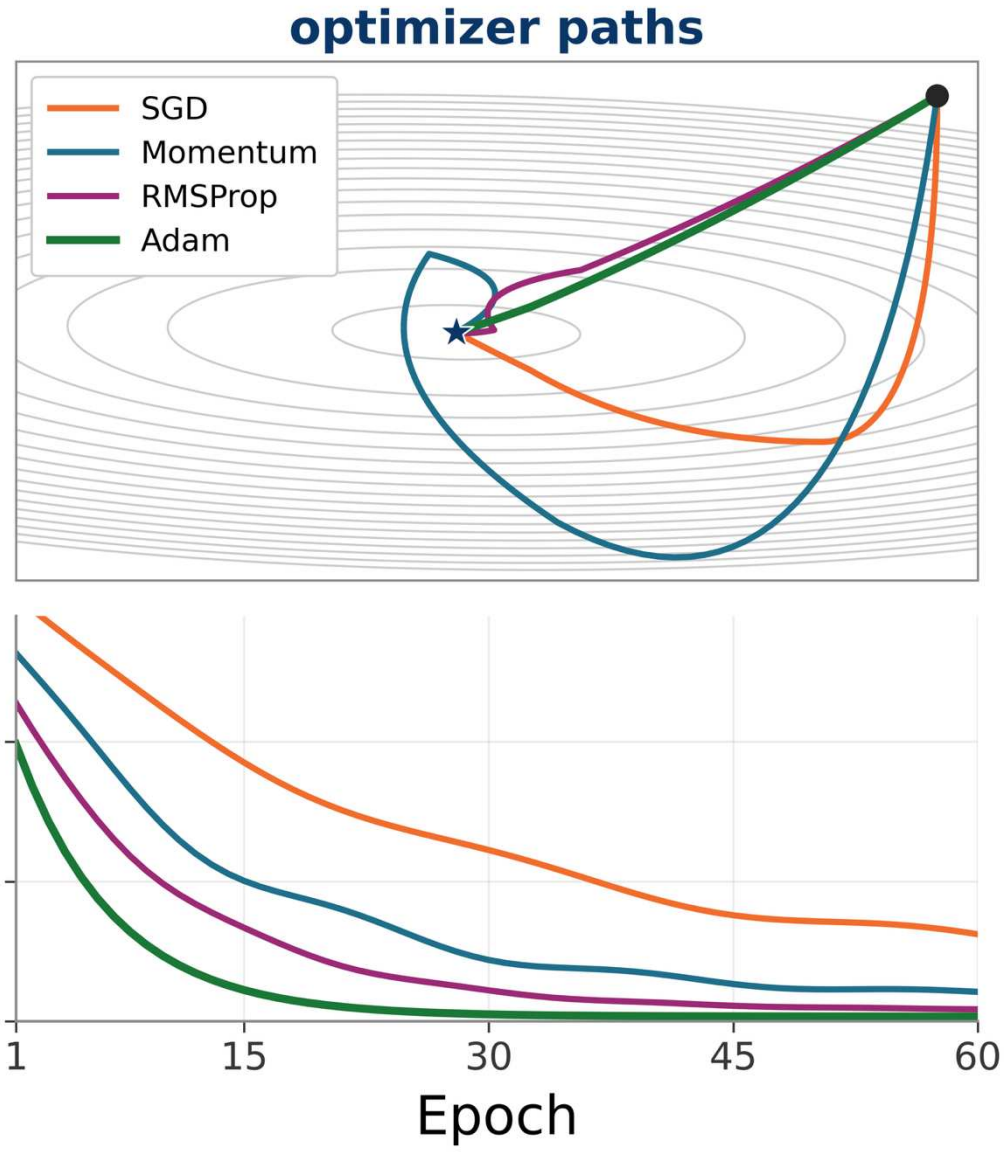
ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION

Diederik P. Kingma*
University of Amsterdam, OpenAI
dpkingma@openai.com

Jimmy Lei Ba*
University of Toronto
jimmy@psi.utoronto.ca

Most neural networks are trained using the Adam optimizer, which **combines momentum + RMSProp**

$$w_t = w_{t-1} - \frac{\eta v_t}{\sqrt{s_t} + \epsilon}$$



Questions?

What to know for the final exam

- You should be able to do a **backprop calculation by hand**
  - There will be practice in discussion section and on HW
- Understand the **error signal** intuitively
- How to use **gradient descent** (and its variants) to train NNs
  - Stochastic (SGD) and mini-batch gradient descent
  - Momentum and (adaptive) learning rate