AI in Mathematics
Lecture 4
Deep Learning in Mathematics

Bar-Ilan University
Nebius Academy | Stevens Institute of Technology
April 8, 2025

About This Course

~~1 week: Intro~~

~~2 weeks: Classic ML~~

2 weeks: Deep Learning in Mathematics

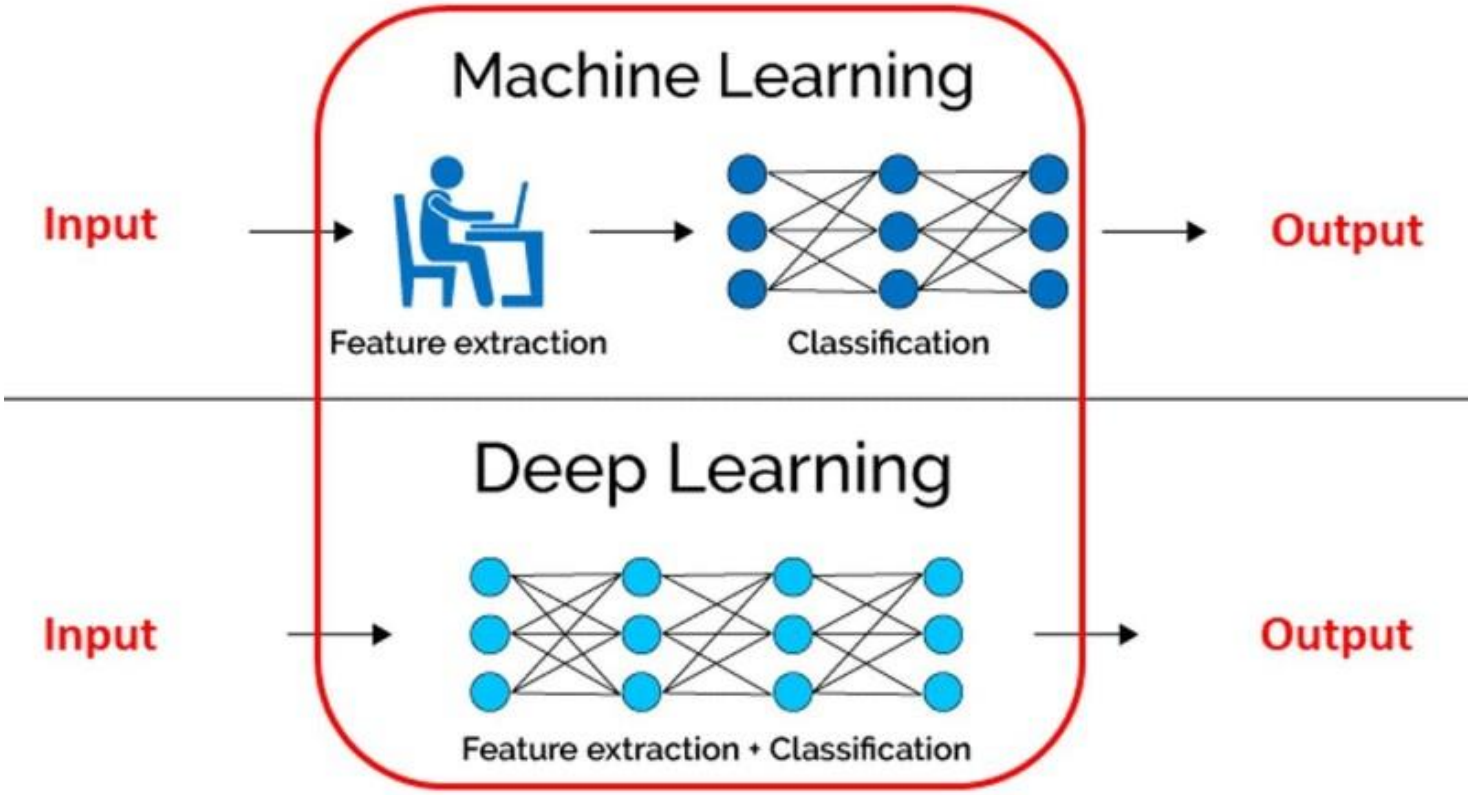
3 weeks: Math as an NLP problem (LLMs etc.)

3 weeks: Reinforcement Learning (RL) in Math

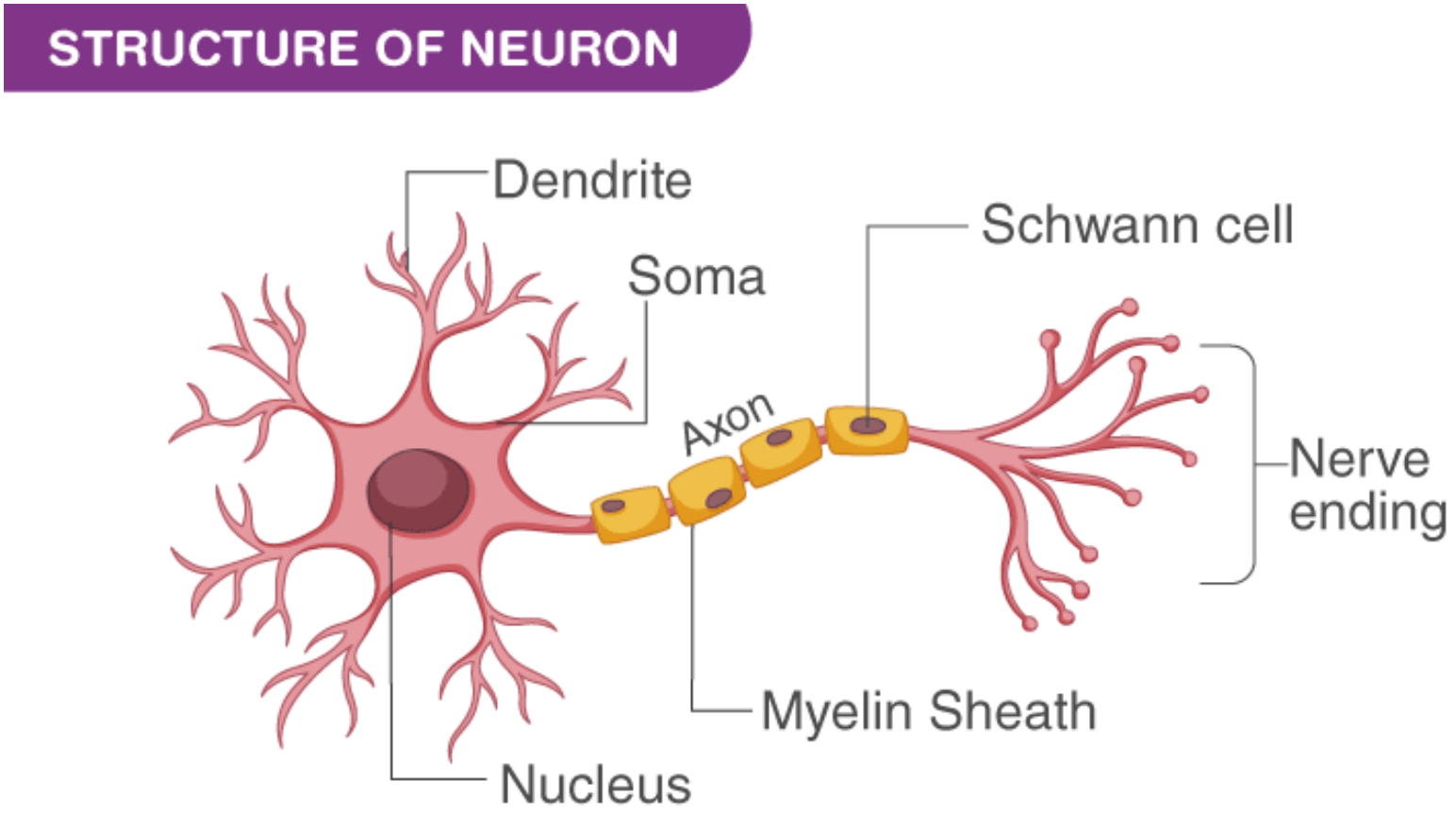
1 week: Advanced AI topics

1 week: Project Presentations

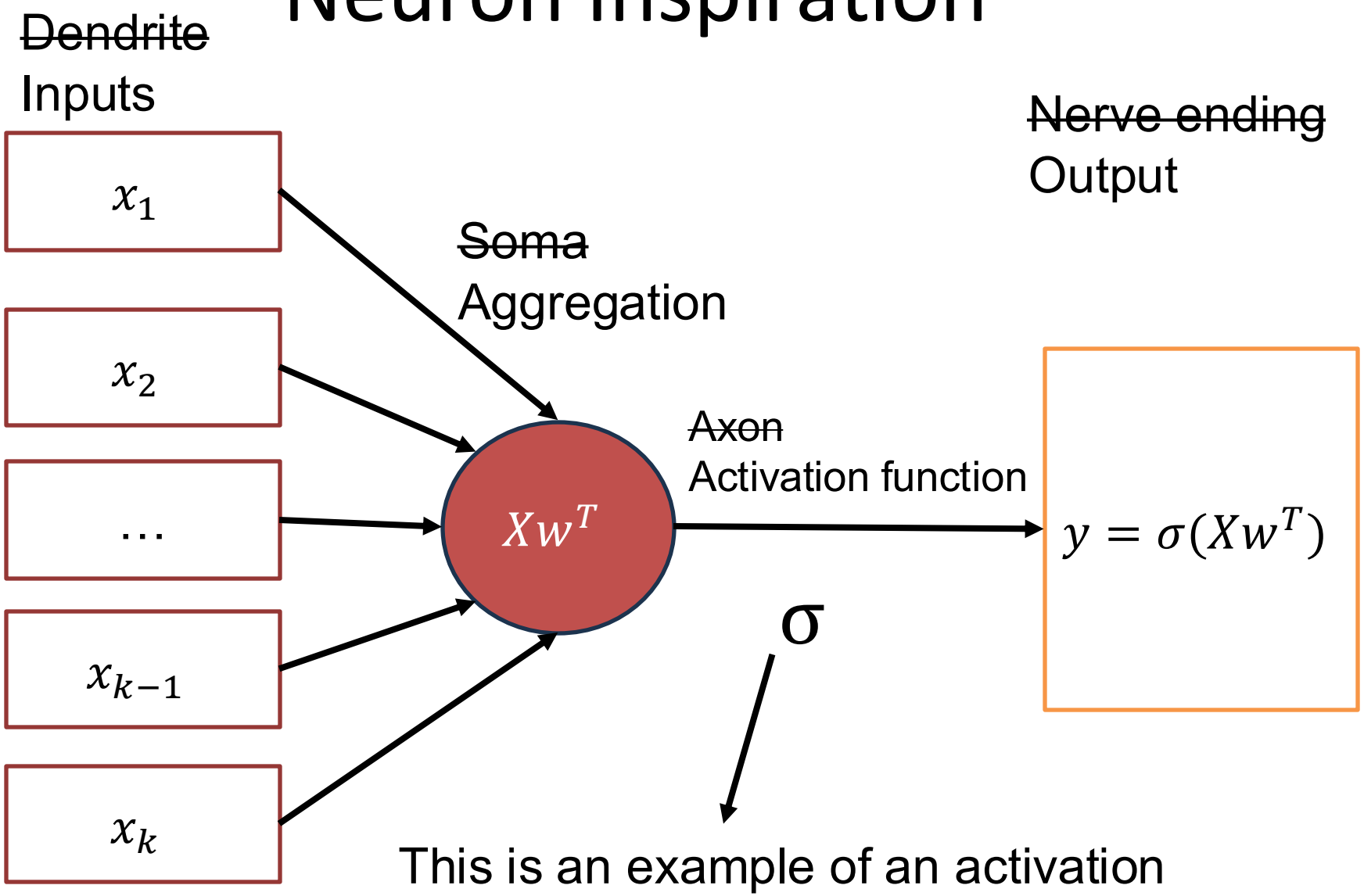
What is deep learning?



Neural Network Inspiration



Neuron Inspiration

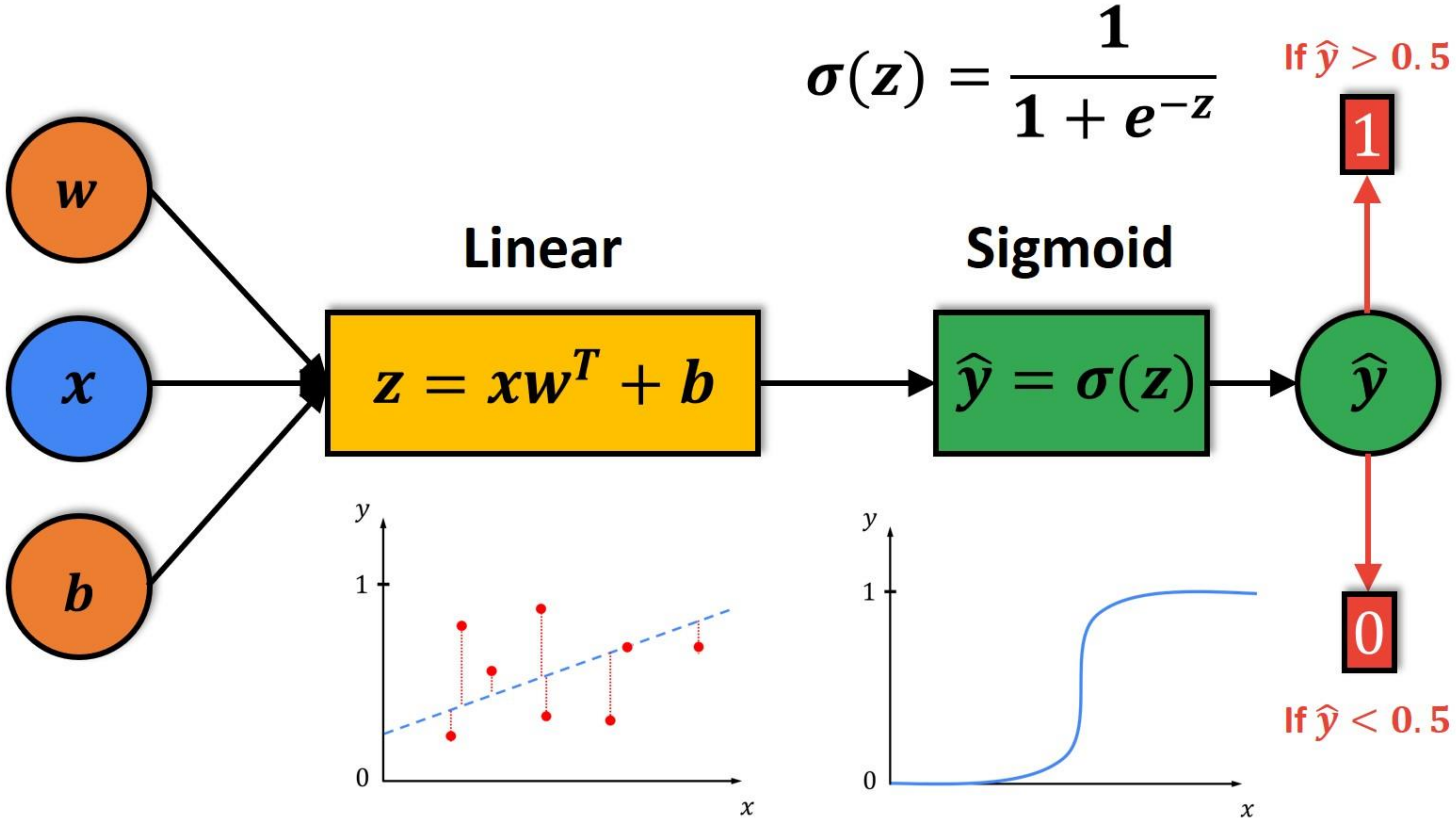


~~Nerve ending~~
Output

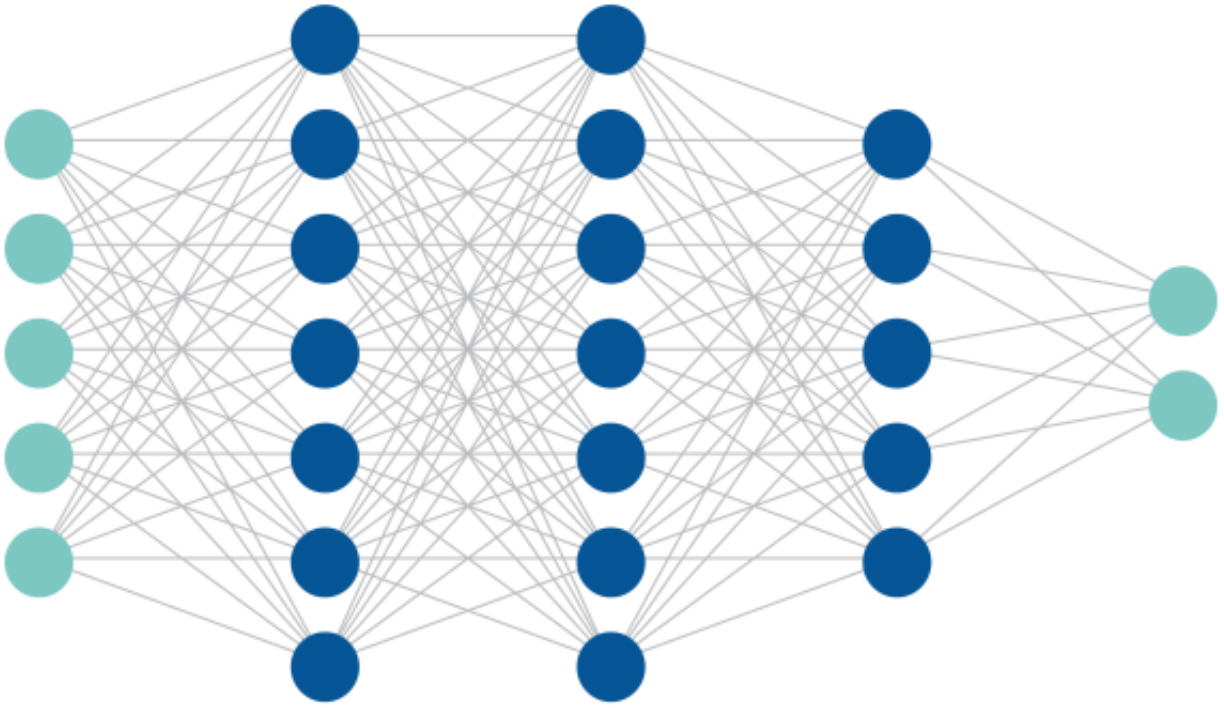
This is an example of an activation function; we will see more examples later.

Logistic Regression

Seems like it was a neuron



Neural Network



$$f(x) = f_L \circ \cdots \circ f_1(x), \qquad f_i(x) = \sigma_i(xW_i^T + b_i)$$

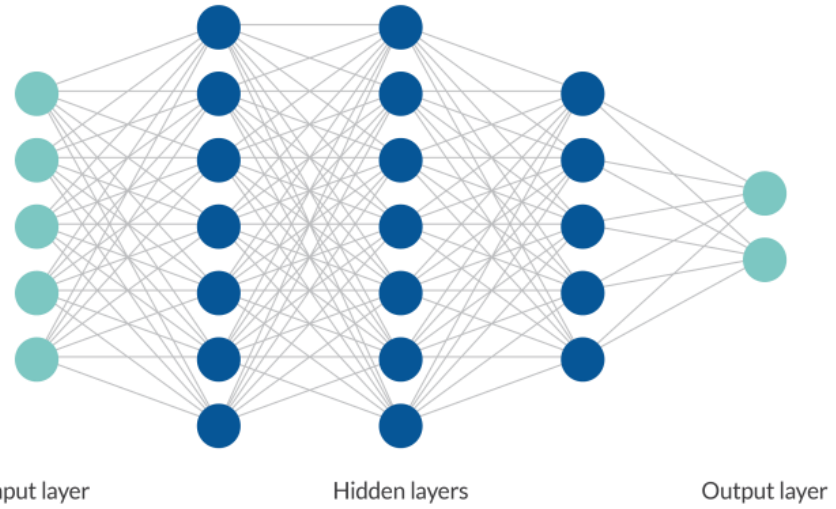
$$W_i \in \mathbb{R}^{n_{i-1} \times n_i}, \qquad b_i \in \mathbb{R}^{n_i}, \qquad x \in \mathbb{R}^{n_0}$$

Neural Network

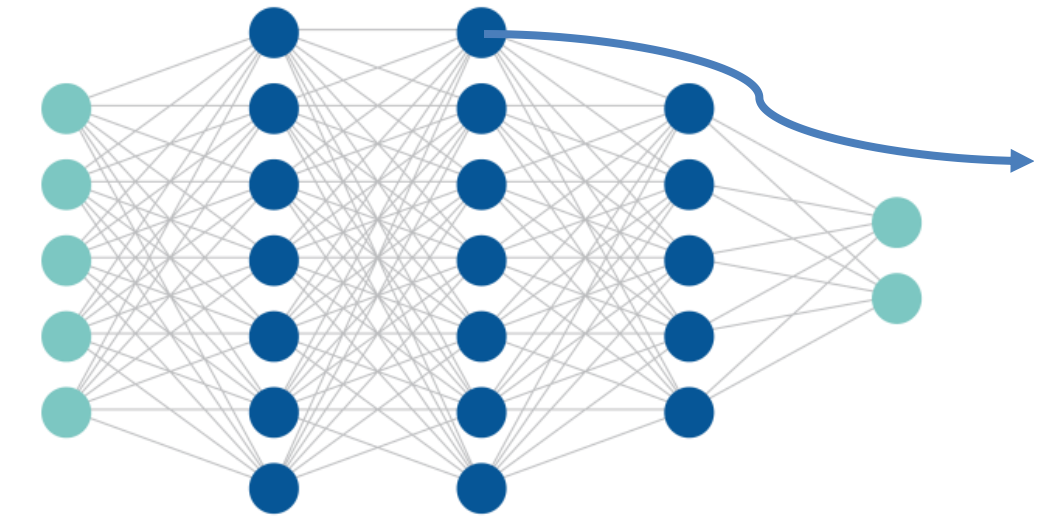
Input layer neurons take features as inputs and pass them inside the model.

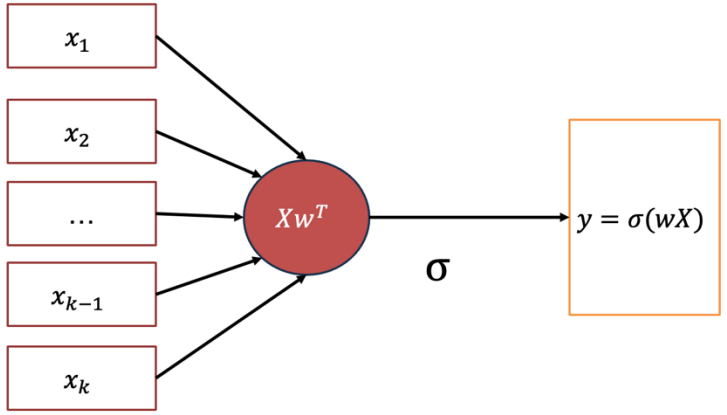
Output layer neurons return values. It is our prediction.

Hidden layer neurons (and calculation of **Output** layer) are what we considered a black box previously.



Neural Network

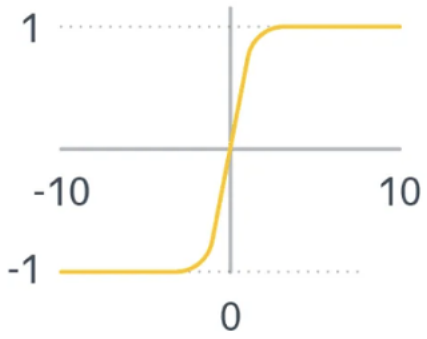




What if we do not have activation function?

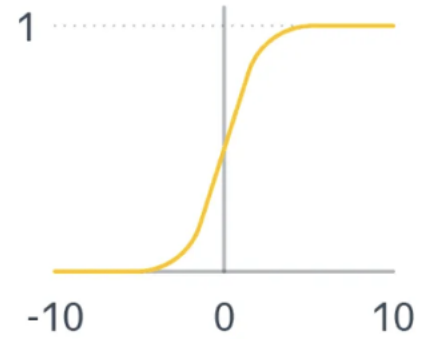
Then neural network reduces to linear model

Activation Functions



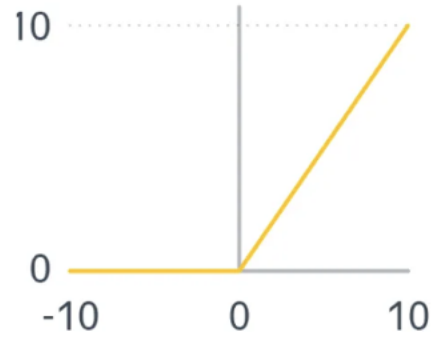
tanh

$tanh(x)$



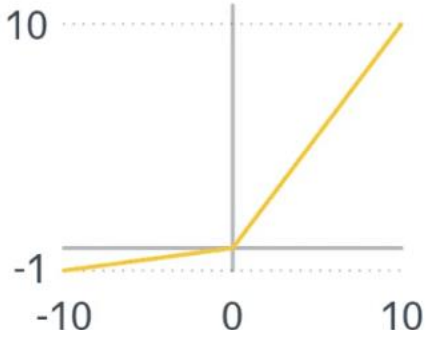
Sigmoid

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



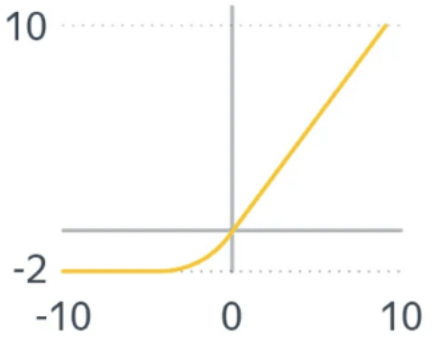
ReLU

$max(0, x)$



Leaky ReLU

$max(0.1x, x)$



ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

Functional Analysis

Neural network with **one** hidden-layer and **non-polynomial activation function** is capable of always approximating a multi-variant continuous function.

Cybenko theorem (1989)

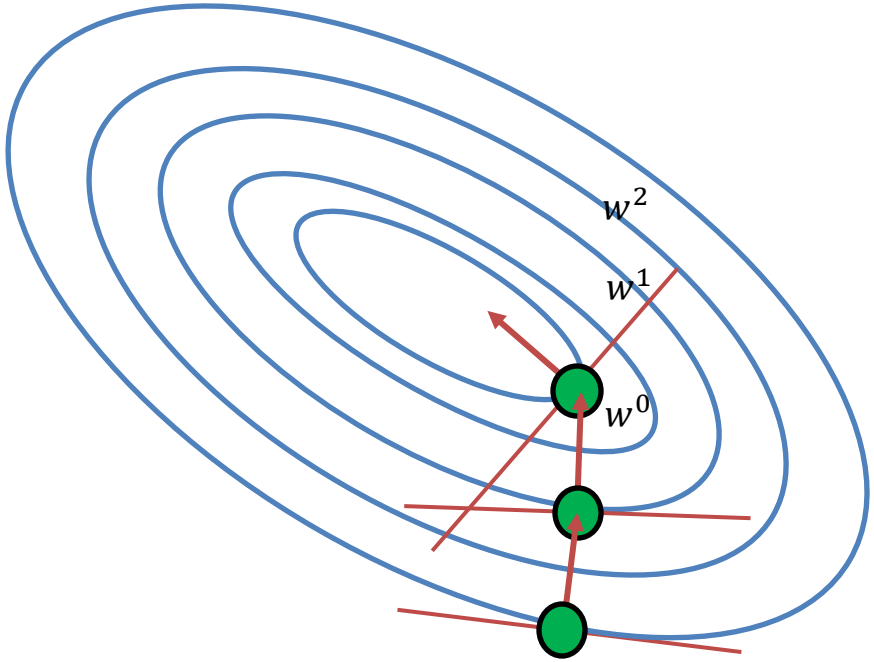
$\forall\ continious\ f: \mathbb{R}^m \rightarrow \mathbb{R}$ and $\varepsilon > 0\ \exists N > 0$ and parameters $w_1 \ldots w_N \in \mathbb{R}^m, b_1 \ldots b_N, \alpha_1 \ldots \alpha_N \in \mathbb{R}$ such that:

$$\left| f(x) - \sum_{i=1}^{N} \alpha_i \sigma(w_i x - b_i) \right| < \varepsilon$$

Why is a **non-polynomial** activation function essential in neural networks?

What limitations do polynomial activation functions have in function approximation?

Gradient Descent Recap



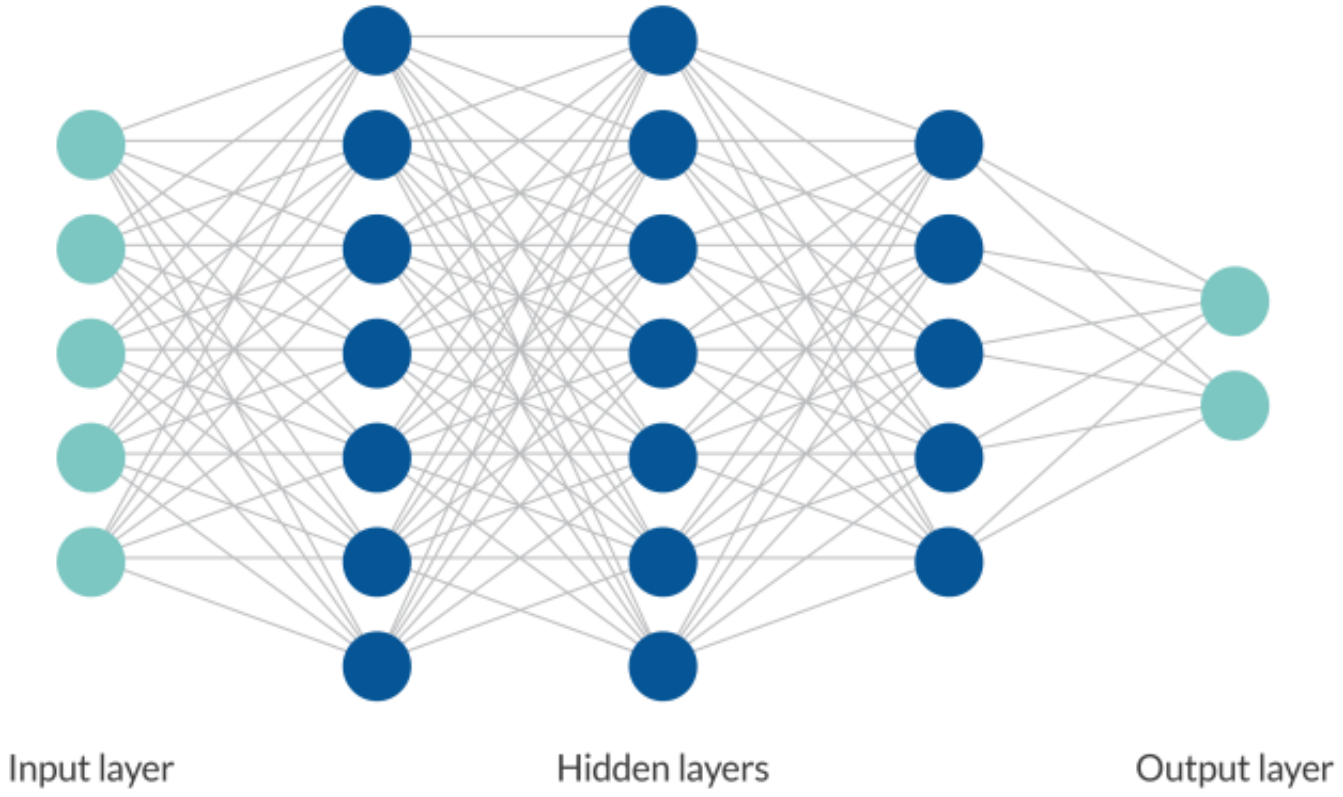
Iterative algorithm:

$$w^{k+1} = w^k - \alpha \nabla_w L\left(w^k\right)$$

α – learning rate (LR).

Backpropagation Overview

We want to update weights of a model, we can do it by a chain rule for each weight of a model.



Backpropagation Motivation

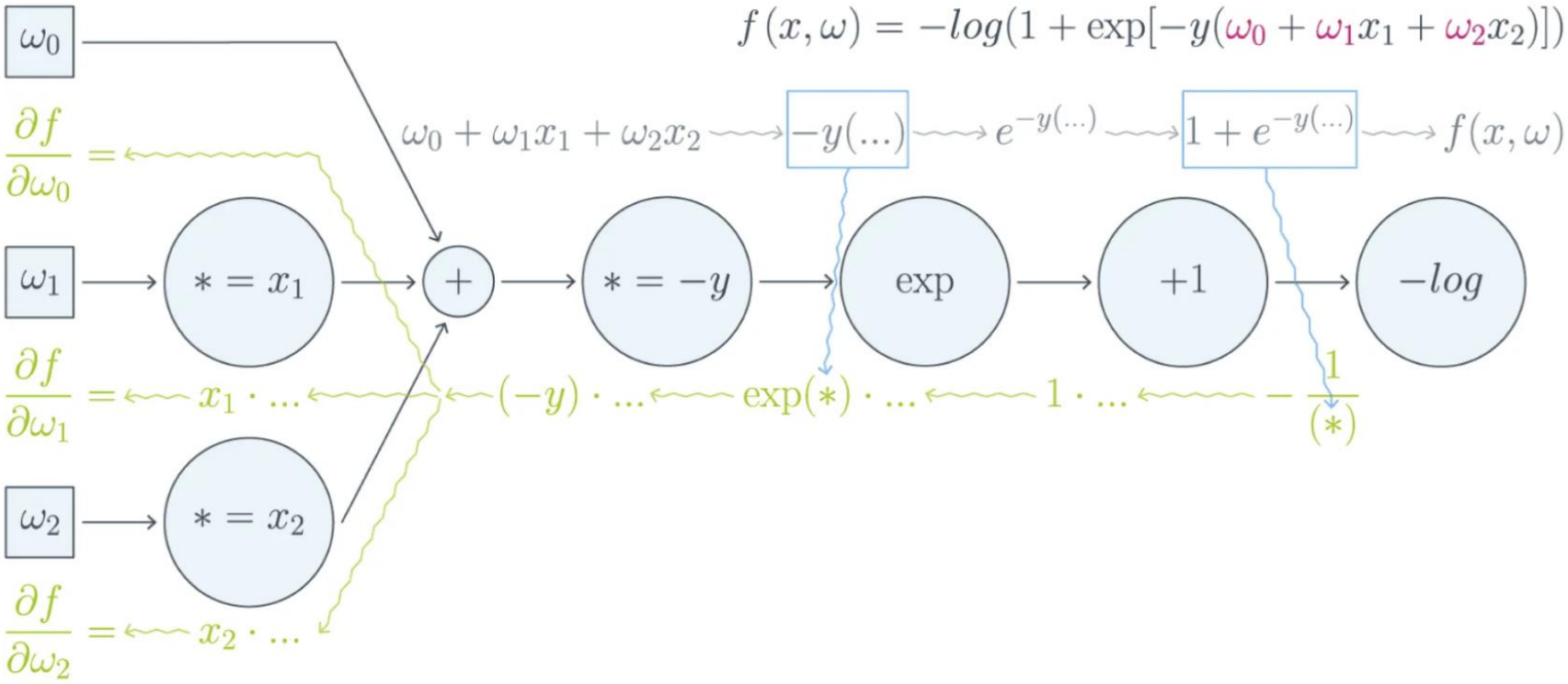
$$f(x) = f_L \circ \cdots \circ f_1(x), \qquad f_i(x) = \sigma_i(xW_i^T + b_i)$$

$W_i \in \mathbb{R}^{n_{i+1} \times n_i}$, $b_i \in \mathbb{R}^{n_i}$, $x \in \mathbb{R}^{n_0}$ – one example.

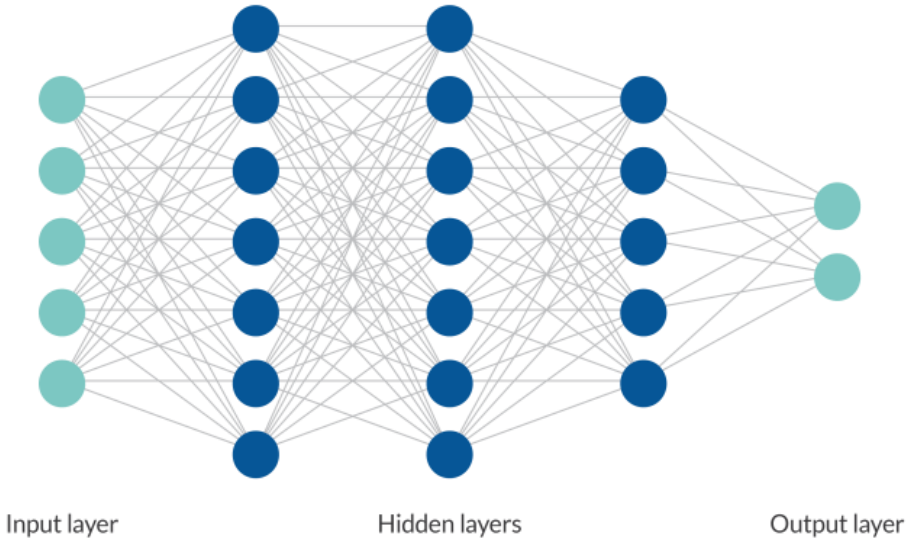
$L(f(x), y)$ – our loss.

In order to compute $\nabla_{W_i} L(f(x), y)$ we can compute each value by chain rule.

Backpropagation Motivation



Backpropagation Motivation

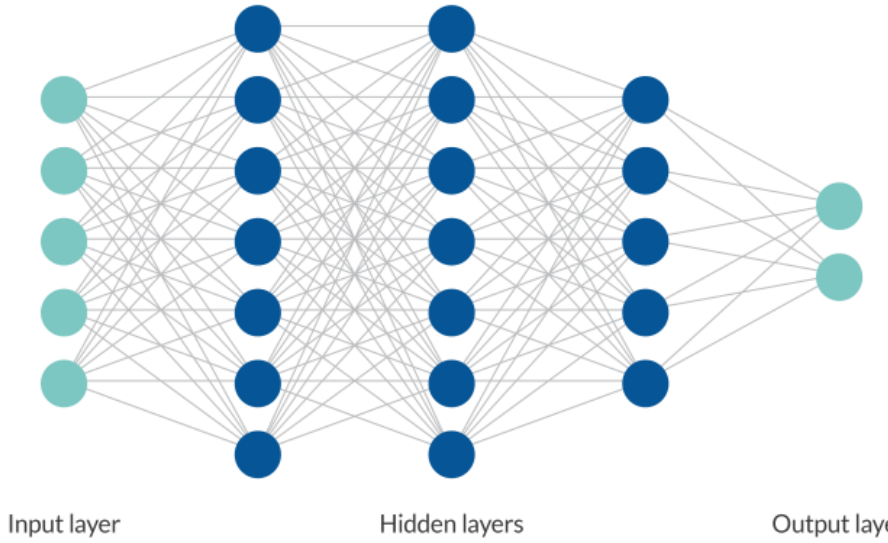


In a neural network, we have many variables and **many paths** through which a weight can influence the final output.

A simple application of the chain rule **doesn't tell us**:

- In what order to compute derivatives,
- Which intermediate values to store,
- How to avoid redundant calculations.

Backpropagation Motivation



Suppose we have a nested function:

$$L(x) = f_4\Big(f_3\Big(f_2\big(f_1(x)\big)\Big)\Big)$$

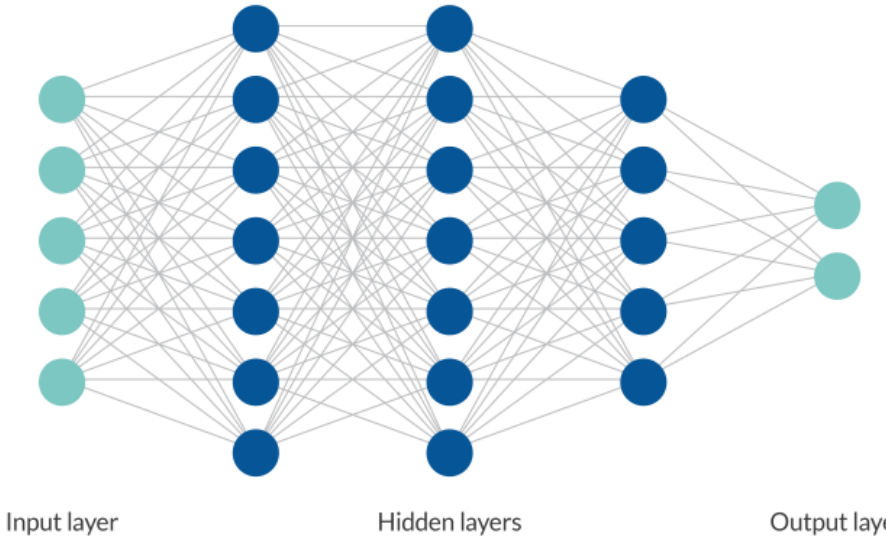
Where W_i – coefficients of f_i.

🚫 **Naive approach is** to compute $\frac{\partial L}{\partial W_i}$ directly, we could expand the full chain, but this is **expensive** and leads to **redundant calculations**.

Backpropagation does a reverse in some sense:

The application of the chain rule in the **correct order**, with **caching of intermediate results**, to compute all gradients **efficiently in linear time**.

Backpropagation



Suppose X_1, X_2, X_3, X_4 - are output variables of $f_1,\ f_2,\ f_3, f_4$.

Backpropagation:

1. **Forward pass:**

Compute and store all intermediate values:

$$f_1(x),\ f_2(f_1(x)),\ f_3(\dots), f_4(\dots)$$

2. **Backward pass**:

• Start from $\nabla_{X_4} L$

• Use cached values to apply the chain rule efficiently and get next layer's parameters gradients:

$$\nabla_{X_4} L \ \rightarrow\ \nabla_{X_3} L$$

Linear Layer Example

Forward Pass

$$y = xW^T + b$$

- $x \in \mathbb{R}^{\mathrm{n}}\ (input\ vector)$
- $W \in \mathbb{R}^{\mathrm{mxn}}\ (weight\ matrix), b \in \mathbb{R}^{\mathrm{m}}\ (bias\ vector)$
- $y \in \mathbb{R}^{\mathrm{m}}\ (output\ vector)$

Backward Pass

Given $\frac{\partial L}{\partial y}$, compute:

- $\frac{\partial L}{\partial b}, \frac{\partial L}{\partial W}$ – parameters of model
- $\boldsymbol{\frac{\partial L}{\partial x}}$ – **Backpropagation step.**

Linear Layer Example

$$y = xW^T + b$$

$$\frac{\partial L}{\partial W_{ij}} = \sum \frac{\partial L}{\partial y_k} \cdot \frac{\partial y_k}{\partial W_{ij}} = \frac{\partial L}{\partial y_j} \cdot x_i$$

Which implies:

$$\boldsymbol{\frac{\partial L}{\partial W}} = x^T \frac{\partial L}{\partial y}$$

Linear Layer Example

$$y = xW^T + b$$

$$\frac{\partial L}{\partial b_j} = \sum_i \frac{\partial L}{\partial y_i} \cdot \frac{\partial y_i}{\partial b_j} = \frac{\partial L}{\partial y_j}$$

Which implies:

$$\boldsymbol{\frac{\partial L}{\partial b}} = \frac{\partial L}{\partial y}$$

Linear Layer Example

$$y = xW^T + b$$

$$\frac{\partial L}{\partial x_j} = \sum_i \frac{\partial L}{\partial y_i} \cdot \frac{\partial y_i}{\partial x_j} = \sum_i \frac{\partial L}{\partial y_i} W_{ij}$$

Which implies:

$$\frac{\boldsymbol{\partial L}}{\boldsymbol{\partial x}} = \frac{\partial L}{\partial y} W$$

Linear Layer Example

$$y = xW^T + b$$

$$\frac{\partial \boldsymbol{L}}{\partial \boldsymbol{W}} = \frac{\partial L}{\partial y} x^T$$

$$\frac{\partial \boldsymbol{L}}{\partial \boldsymbol{b}} = \frac{\partial L}{\partial y}$$

$$\frac{\partial \boldsymbol{L}}{\partial \boldsymbol{x}} = W^T \frac{\partial L}{\partial y}$$

Calculating $\frac{\partial \boldsymbol{L}}{\partial \boldsymbol{x}}$ allows us to iterate further.

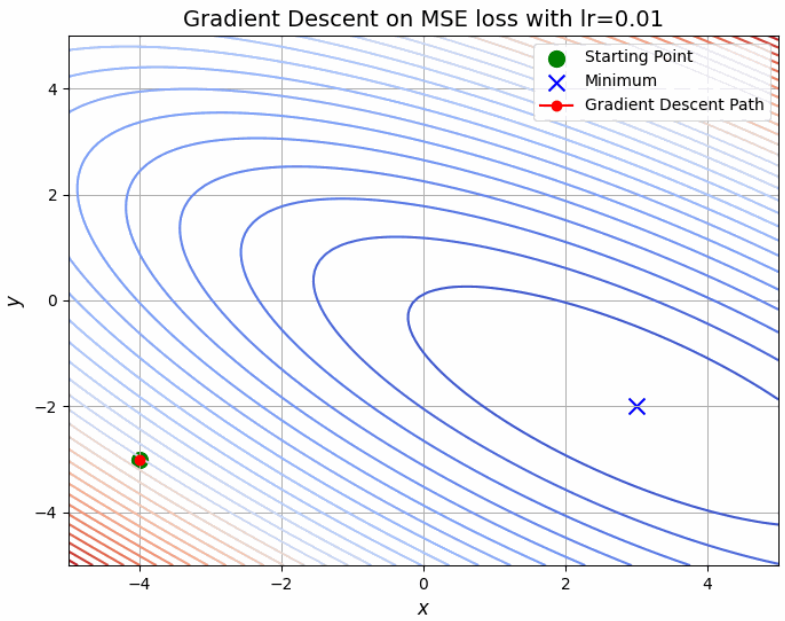
Optimization

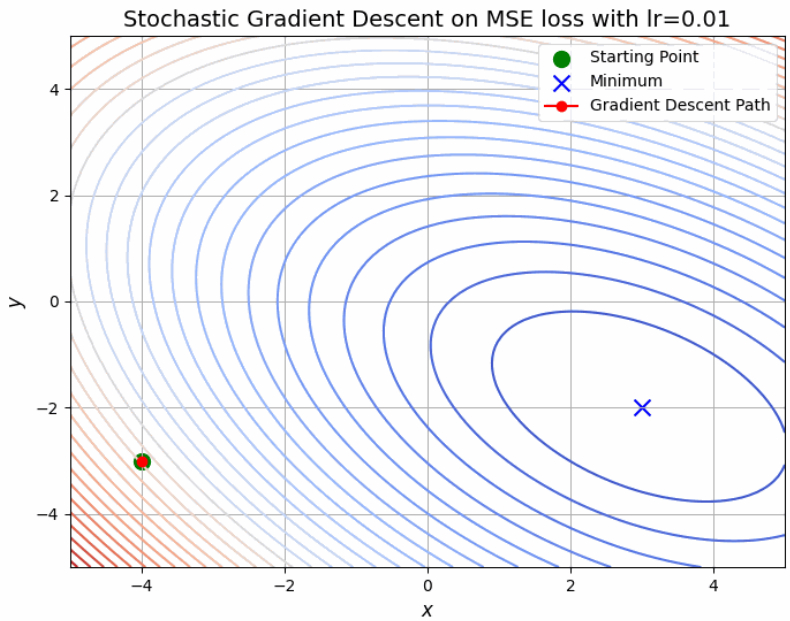
Calculating each iteration of Backpropagation is costly for **large** datasets.

So we need to have a good

Stochastic Gradient Descent techniques.

Reminder: SGD





SGD is erratic; Smaller batches make it worse.

Use SGD only when full GD is impractical.

Vanilla SGD is rarely used nowadays, people prefer modified versions.
We will discuss them **next time. Next time has come!**

Momentum

$$g_k = \nabla_w \mathcal{L}(w_k)$$

$$\mu_k = \beta_1 \cdot \mu_{k-1} + (1 - \beta_1) g_k$$

$$w_k = w_{k-1} - \alpha_k \cdot \mu_k$$

Accumulates previous directions
Makes GD less erratic

RMSProp
(Root Mean Square Propagation)

$$g_k = \nabla_w \mathcal{L}(w_k)$$

$$\nu_k = \beta_2 \cdot \nu_{k-1} + (1 - \beta_2) g_k^2$$

Adaptive step
for each coordinate
(second-order methods)

$$w_k = w_{k-1} - \alpha_k \cdot \frac{g_k}{\sqrt{\nu_k + \epsilon}}$$

Adam
(Adaptive Moment Estimation)

$$g_k = \nabla_w \mathcal{L}(w_k)$$

$$\mu_k = \beta_1 \cdot \mu_{k-1} + (1 - \beta_1) g_k$$

$$\nu_k = \beta_2 \cdot \nu_{k-1} + (1 - \beta_2) g_k^2$$

$$\hat{\mu}_k = \frac{\mu_k}{1 - \beta_1^k}, \qquad \hat{\nu}_k = \frac{\nu_k}{1 - \beta_2^k}$$

$$w_k = w_{k-1} - \alpha_k \cdot \frac{\hat{\mu}_k}{\sqrt{\hat{\nu}_k + \epsilon}}$$