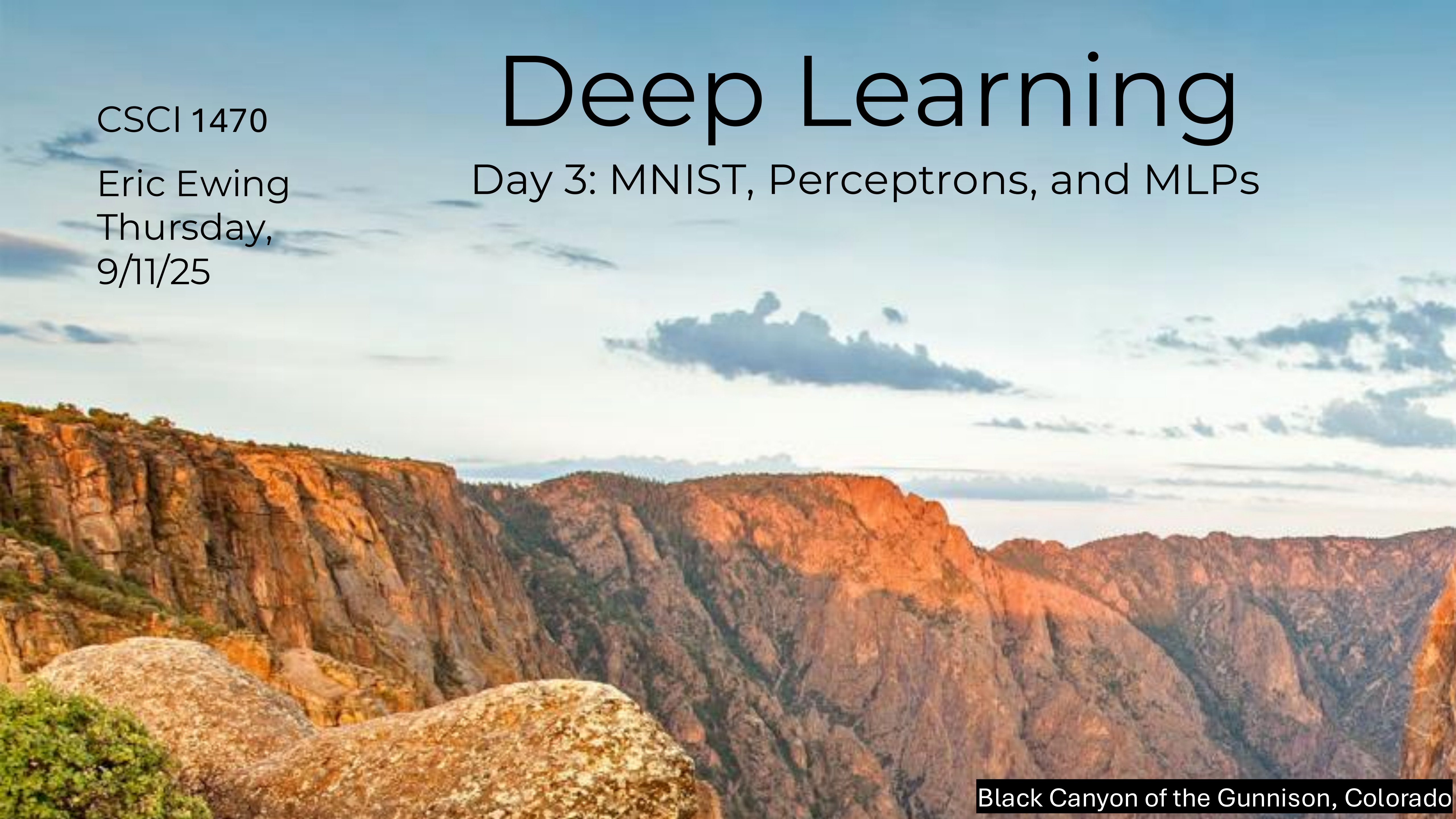
Deep Learning

Day 3: MNIST, Perceptrons, and MLPs

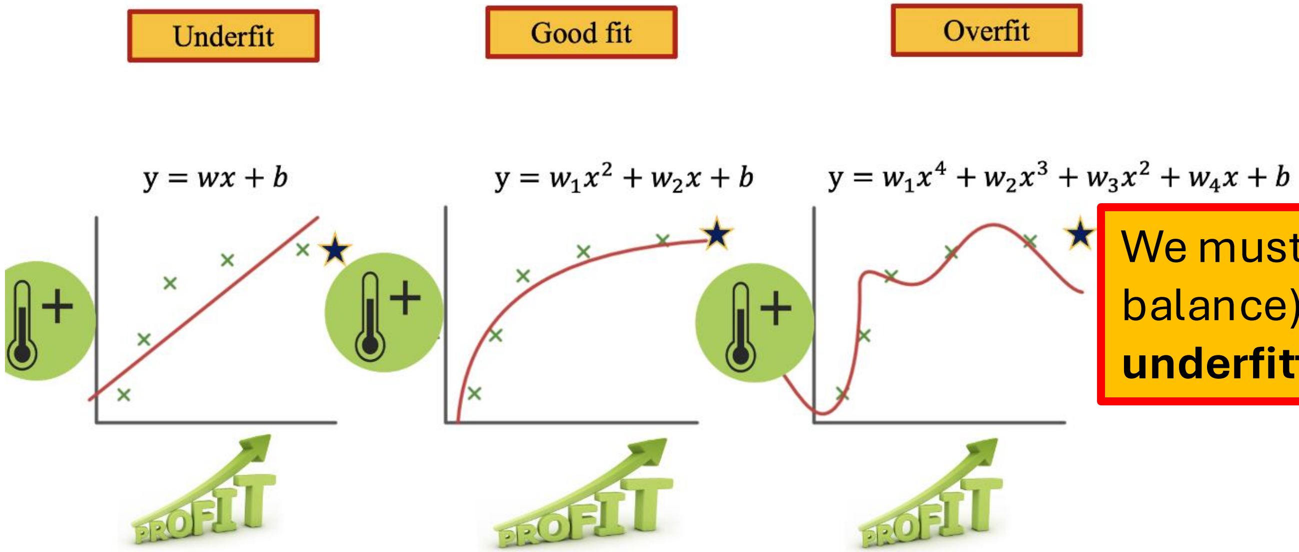
CSCI 1470

Eric Ewing
Thursday, 9/11/25



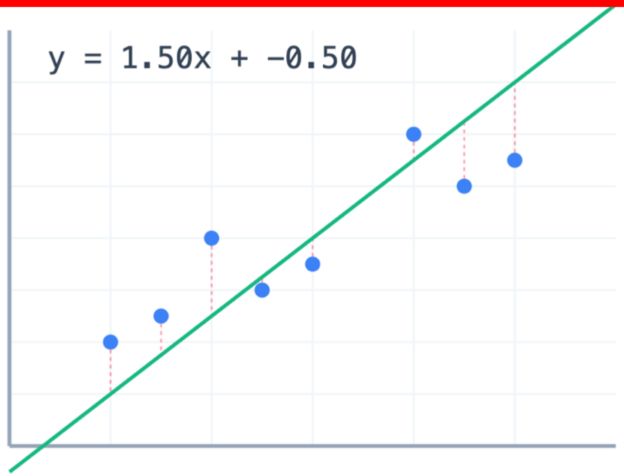
Black Canyon of the Gunnison, Colorado

Recap

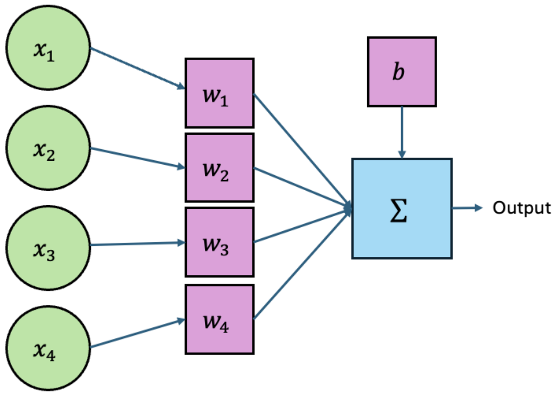


We must always test for (and balance) **overfitting and underfitting**

Loss Functions tell us about the performance of the model (which we will also optimize for)

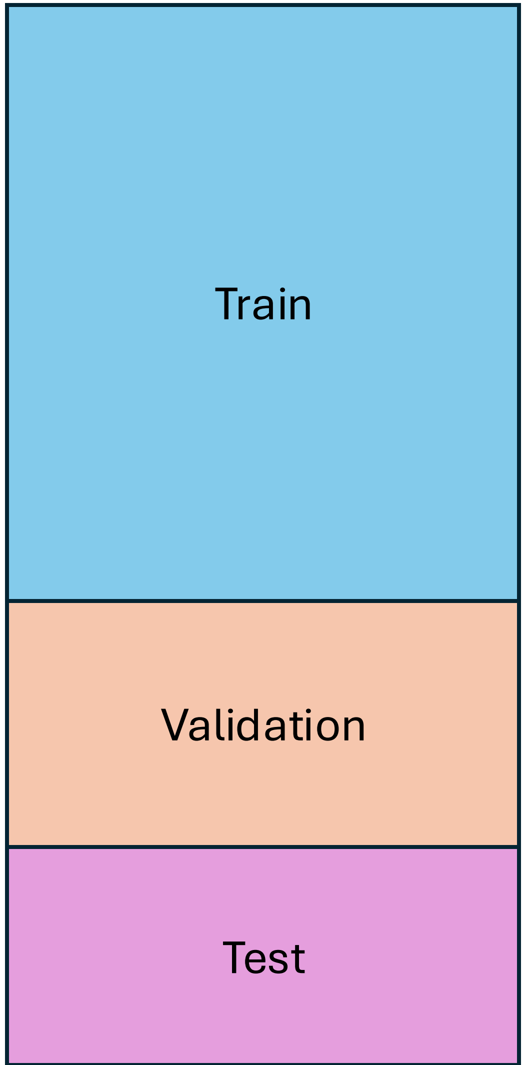


A perceptron/neuron works just like a linear regression, but has a different **activation function**



Train, validation, and test sets

- **Training Set**: Used to adjust parameters of model
- ***Validation set*** — used to test how well we're doing as we develop
  - Prevents ***overfitting***
- ***Test Set*** — used to evaluate the model once the model is done



MNIST

The most famous dataset in Deep Learning

Modified **N**ational **I**nstitute of **S**tandards and **T**echnology database

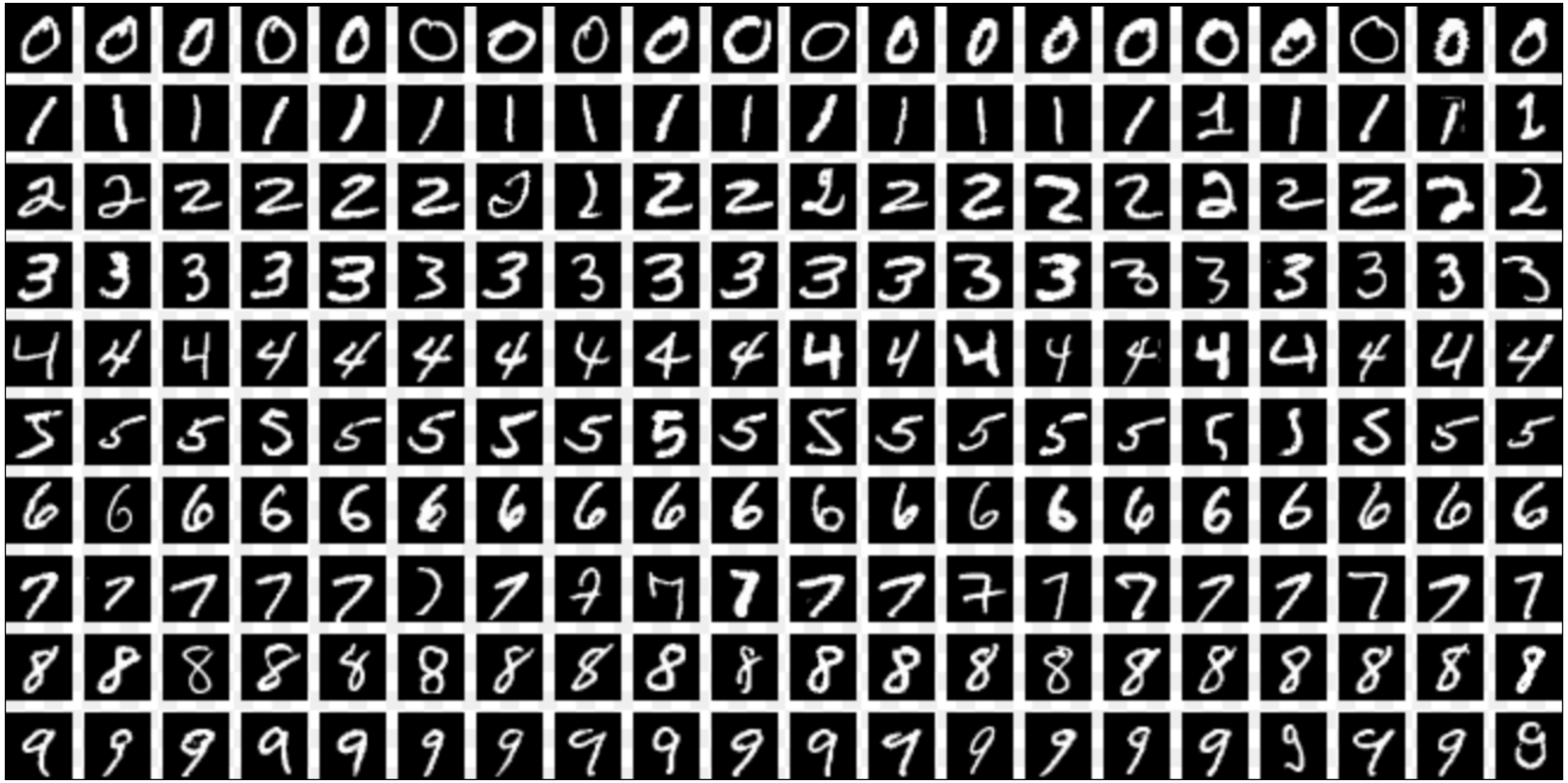
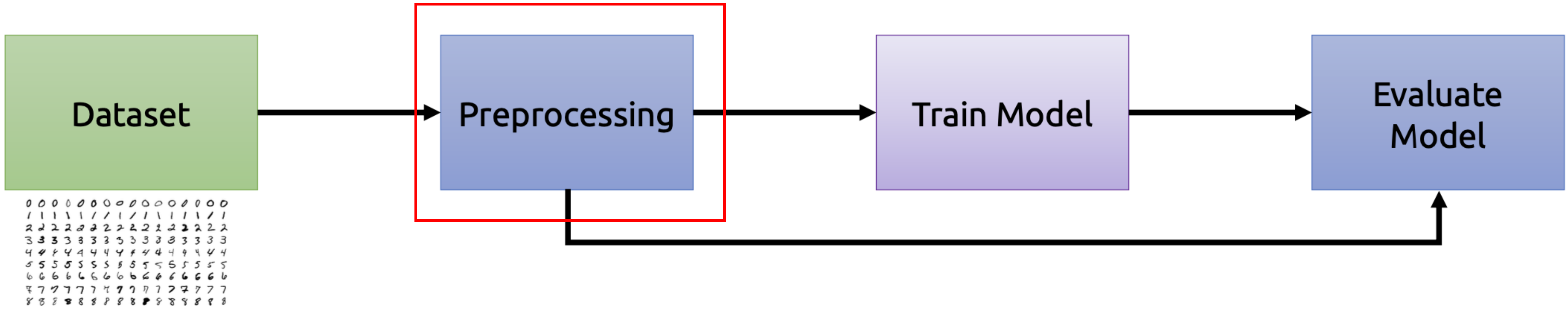


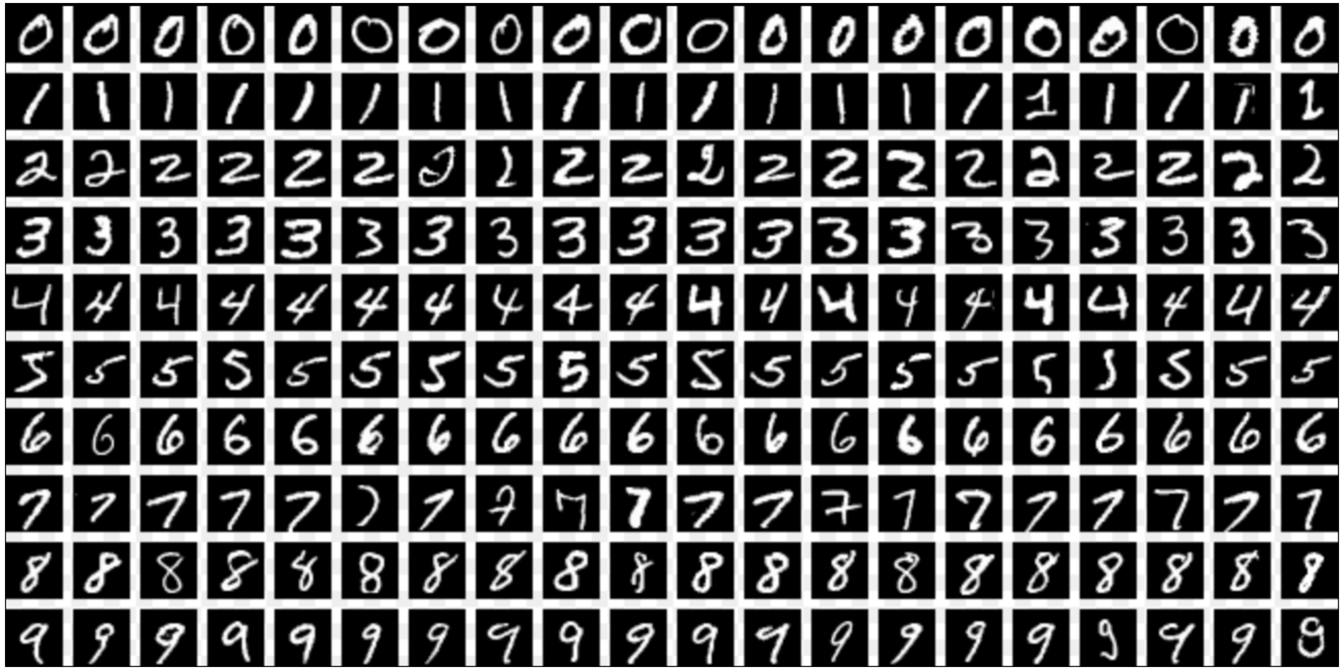
Image courtesy of Wikipedia

Machine Learning Pipeline for Digit Recognition



MNIST

- 60,000 Images in training set
- 10,000 Images in test set
- No explicit validation set

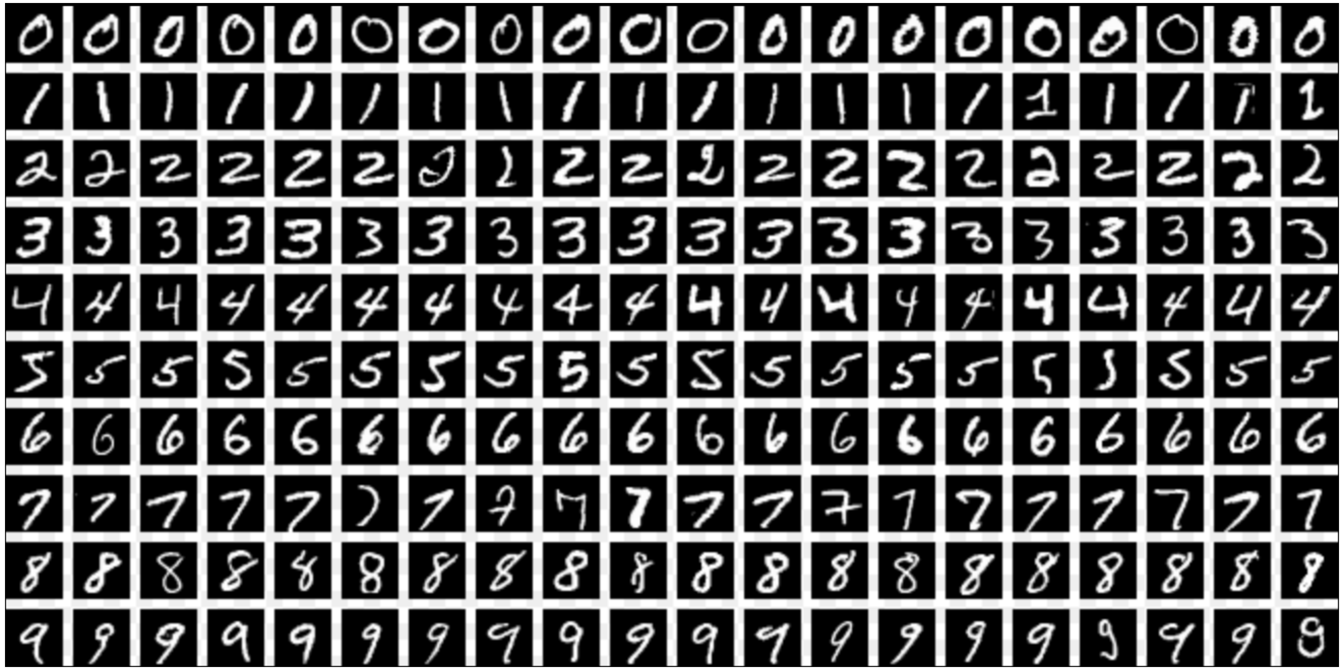


MNIST

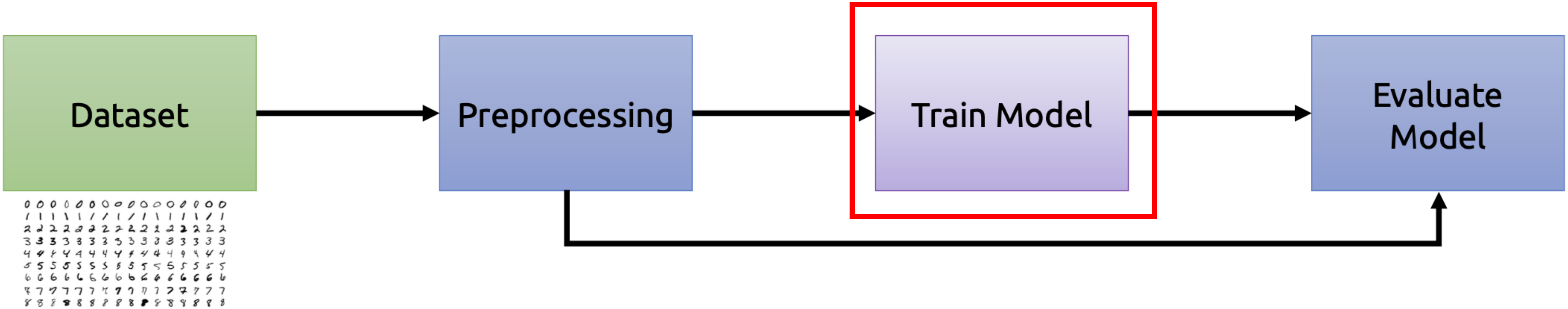
- 60,000 Images in training set
- 10,000 Images in test set
- No explicit validation set

What do you suggest we do?

80/20 train/validation splits are common



Machine Learning Pipeline for Digit Recognition



Our Problem:

Classifying MNIST digits requires predicting 1 of 10 possible values

Input: $\mathbb{X}$

Target: $\mathbb{Y}$

Pixel Grid

Which digit is it?

$x^{(1)} =$ 

28x28 pixels

Function: f

$\boxed{f(\mathbb{X}) \rightarrow \mathbb{Y}}$

$y^{(1)} = \text{“2”}$

$x^{(2)} =$ 

$y^{(2)} = \text{“0”}$

Our Problem:

Classifying MNIST digits requires predicting 1 of 10 possible values

What is our input space?

Input: $\mathbb{X}$

Target: $\mathbb{Y}$

Pixel Grid

Which digit is it?

$x^{(1)} =$ 

28x28 pixels

Function: f

$f(\mathbb{X}) \rightarrow \mathbb{Y}$

$y^{(1)} = \text{“2”}$

$x^{(2)} =$ 

$y^{(2)} = \text{“0”}$

Our Problem:

Classifying MNIST digits requires predicting 1 of 10 possible values

Input: $\mathbb{X}$

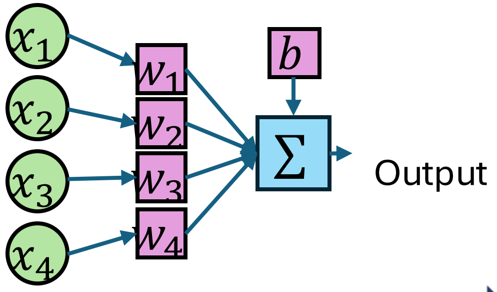
What is our input space?

Target: $\mathbb{Y}$

Pixel Grid

$x^{(1)} =$ 

28x28 pixels



Function: f

$f(\mathbb{X}) \rightarrow \mathbb{Y}$

Which digit is it?

$y^{(1)} =$ "2"

$x^{(2)} =$ 

$y^{(2)} =$ "0"

Our Problem:

Classifying MNIST digits requires predicting 1 of 10 possible values

Input: $\mathbb{X}$

What is our input space?

What is our output space?

Target: $\mathbb{Y}$

Pixel Grid

Which digit is it?

$x^{(1)} =$ 

28x28 pixels

Function: f

$f(\mathbb{X}) \rightarrow \mathbb{Y}$

$y^{(1)} =$ "2"

$x^{(2)} =$ 

$y^{(2)} =$ "0"

Our Problem:

Classifying MNIST digits requires predicting 1 of 10 possible values

Input: $\mathbb{X}$

What is our input space?

What is our output space?

What is our prediction task?

Target: $\mathbb{Y}$

Pixel Grid

Which digit is it?

$x^{(1)} =$ 

Function: f

$y^{(1)} =$ "2"

28x28 pixels

$\boxed{f(\mathbb{X}) \rightarrow \mathbb{Y}}$

$x^{(2)} =$ 

$y^{(2)} =$ "0"

Our simplified problem:

What is our input space?

What is our output space?

What is our prediction task?

Input: $\mathbb{X}$

Target: $\mathbb{Y}$

Pixel Grid

Is it digit 2?

$x^{(1)} =$ 

28x28 pixels

Function: f

$f(\mathbb{X}) \rightarrow \mathbb{Y}$

$y^{(1)} = 1$ 

$x^{(2)} =$ 

$y^{(2)} = 0$ 

The Perceptron Algorithm

Loop Over Dataset (until no weights change)

- For each misclassified example
  - update weights to make better prediction for example

The Perceptron Algorithm

1. Initialize $\vec{\theta} = \vec{0}$
2. For N iterations or until $\vec{\theta}$ does not change
    1. For each example $x^{(k)}$ with label $y^{(k)}$
        1. If $y^{(k)} = f(x^{(k)})$, continue
        2. Else, for all parameters $\theta_i \in \vec{\theta}$, $\theta_i = \theta_i + \left(y^{(k)} - f\left(x^{(k)}\right)\right) \cdot x_i^{(k)}$

w: weights
b: bias
θ: parameters (weights and biases), $\vec{\theta} = \{\vec{w} \cup b\}$

$x^{(k)}$: k’th training example, $y^{(k)}$ k’th training label
$x_i^{(k)}$: i’th feature for k’th example

The Perceptron Algorithm

1. $\boxed{\text{Initialize } \vec{\theta} = \vec{0}}$

Need to start somewhere…
any initial setting will work

2. For N iterations or until $\vec{\theta}$ does not change
    1. For each example $x^{(k)}$ with label $y^{(k)}$
        1. If $y^{(k)} = f(x^{(k)})$, continue
        2. Else, for all parameters $\theta_i \in \vec{\theta}$, $\theta_i = \theta_i + \left(y^{(k)} - f\left(x^{(k)}\right)\right) \cdot x_i^{(k)}$

w: weights
b: bias
θ: parameters (weights and biases), $\vec{\theta} = \{\vec{w} \cup b\}$

$x^{(k)}$: k'th training example, $y^{(k)}$ k'th training label
$x_i^{(k)}$: i'th feature for k'th example

The Perceptron Algorithm

N is referred to as **"epochs"**: Number of times the entire dataset is iterated through

1. Initialize $\vec{\theta} = \vec{0}$
2. $\boxed{\text{For N iterations}}$ or until $\vec{\theta}$ does not change
   1. For each example $x^{(k)}$ with label $y^{(k)}$
      1. If $y^{(k)} = f(x^{(k)})$, continue
      2. Else, for all parameters $\theta_i \in \vec{\theta}$, $\theta_i = \theta_i + \left(y^{(k)} - f\left(x^{(k)}\right)\right) \cdot x_i^{(k)}$

w: weights
b: bias
θ: parameters (weights and biases), $\vec{\theta} = \{\vec{w} \cup b\}$

$x^{(k)}$: k'th training example, $y^{(k)}$ k'th training label
$x_i^{(k)}$: i'th feature for k'th example

The Perceptron Algorithm

1. Initialize $\vec{\theta} = \vec{0}$
2. For N iterations or until $\vec{\theta}$ does not change
   1. $\boxed{\text{For each example } x^{(k)} \text{ with label } y^{(k)}}$ $\boxed{\text{Loop over every example in dataset}}$
      1. If $y^{(k)} = f(x^{(k)})$, continue
      2. Else, for all parameters $\theta_i \in \vec{\theta}$, $\theta_i = \theta_i + \left(y^{(k)} - f\left(x^{(k)}\right)\right) \cdot x_i^{(k)}$

w: weights
b: bias
θ: parameters (weights and biases), $\vec{\theta} = \{\vec{w} \cup b\}$

$x^{(k)}$: k'th training example, $y^{(k)}$ k'th training label
$x_i^{(k)}$: i'th feature for k'th example

The Perceptron Algorithm

1. Initialize $\vec{\theta} = \vec{0}$
2. For N iterations or until $\vec{\theta}$ does not change
    1. For each example $x^{(k)}$ with label $y^{(k)}$
        1. If $y^{(k)} = f(x^{(k)})$, continue
        2. Else, for all parameters $\theta_i \in \vec{\theta}$, $\theta_i = \theta_i + (y^{(k)} - f(x^{(k)})) \cdot x_i$

Look only at examples that are misclassified (i.e., $y^{(k)} \neq f(x^{(k)})$)

w: weights
b: bias
θ: parameters (weights and biases), $\vec{\theta} = \{\vec{w} \cup b\}$

$x^{(k)}$: k'th training example, $y^{(k)}$ k'th training label
$x_i^{(k)}$: i'th feature for k'th example

The Perceptron Algorithm

For every parameter in our perceptron…

1. Initialize $\vec{\theta} = \vec{0}$
2. For N iterations or until $\vec{\theta}$ does not change
   1. For each example $x^{(k)}$ with label $y^{(k)}$
      1. If $y^{(k)} = f(x^{(k)})$, continue
      2. Else, for all parameters $\theta_i \in \vec{\theta}$ $\boxed{\theta_i = \theta_i + \left(y^{(k)} - f\left(x^{(k)}\right)\right) \cdot x_i^{(k)}}$

w: weights
b: bias
θ: parameters (weights and biases), $\vec{\theta} = \{\vec{w} \cup b\}$

$x^{(k)}$: k'th training example, $y^{(k)}$ k'th training label
$x_i^{(k)}$: i'th feature for k'th example

The Perceptron Algorithm

For every parameter in our perceptron…

If $y^{(k)} = 1$ and $f\left(x^{(k)}\right) = 0$ and $x_i^{(k)}$>0…

1. Initialize $\vec{\theta} = \vec{0}$
2. For N iterations or until $\vec{\theta}$ does not chang
   1. For each example $x^{(k)}$ with label $y^{(k)}$
      1. If $y^{(k)} = f(x^{(k)})$, continue
      2. Else, for all parameters $\theta_i \in \vec{\theta}$ $\boxed{\theta_i = \theta_i + \left(y^{(k)} - f\left(x^{(k)}\right)\right) \cdot x_i^{(k)}}$

$x^{(k)}$: k’th training example, $y^{(k)}$ k’th training label
$x_i^{(k)}$: i’th feature for k’th example

w: weights
b: bias
θ: parameters (weights and biases), $\vec{\theta} = \{\vec{w} \cup b\}$

The Perceptron Algorithm

1. Initialize $\vec{\theta} = \vec{0}$
2. For N iterations or until $\vec{\theta}$ does not chang
   1. For each example $x^{(k)}$ with label $y^{(k)}$
      1. If $y^{(k)} = f(x^{(k)})$, continue
      2. Else, for all parameters $\theta_i \in \vec{\theta}$ $\boxed{\theta_i = \theta_i + \left(y^{(k)} - f\left(x^{(k)}\right)\right) \cdot x_i^{(k)}}$

For every parameter in our perceptron…

If $y^{(k)} = 1$ and $f\left(x^{(k)}\right) = 0$ and $x_i^{(k)}$>0…

θ_i increases

w: weights
b: bias
θ: parameters (weights and biases), $\vec{\theta} = \{\vec{w} \cup b\}$

$x^{(k)}$: k'th training example, $y^{(k)}$ k'th training label
$x_i^{(k)}$: i'th feature for k'th example

The Perceptron Algorithm

If no parameters change, then we know that $y^{(k)} = f\left(x^{(k)}\right) \forall k$

1. Initialize $\vec{\theta} = \vec{0}$
2. For N iterations or until $\boxed{\vec{\theta} \text{ does not change}}$
    1. For each example $x^{(k)}$ with label $y^{(k)}$
        1. If $y^{(k)} = f(x^{(k)})$, continue
        2. Else, for all parameters $\theta_i \in \vec{\theta}$, $\theta_i = \theta_i + \left(y^{(k)} - f\left(x^{(k)}\right)\right) \cdot x_i^{(k)}$

w: weights
b: bias
θ: parameters (weights and biases), $\vec{\theta} = \{\vec{w} \cup b\}$

$x^{(k)}$: k'th training example, $y^{(k)}$ k'th training label
$x_i^{(k)}$: i'th feature for k'th example

Converting Perceptrons to Multi-Class Classification

Our Problem:

Classifying MNIST digits requires predicting 1 of 10 possible values

Input: $\mathbb{X}$

Target: $\mathbb{Y}$

Pixel Grid

$x^{(1)} =$ 

28x28 pixels

Function: f

$f(\mathbb{X}) \rightarrow \mathbb{Y}$

Which digit is it?

$y^{(1)} =$ “2”

$x^{(2)} =$ 

$y^{(2)} =$ “0”

Our Problem:

Classifying MNIST digits requires predicting 1 of 10 possible values

Target: $\mathbb{Y}$

Input: $\mathbb{X}$

How do we do this?

Pixel Grid

Which digit is it?

$x^{(1)} =$ 

Function: f

$y^{(1)} =$ "2"

28x28 pixels

$f(\mathbb{X}) \rightarrow \mathbb{Y}$

$x^{(2)} =$ 

$y^{(2)} =$ "0"

Our Problem:

Classifying MNIST digits requires predicting 1 of 10 possible values

Input: $\mathbb{X}$

How do we do this?

Target: $\mathbb{Y}$

Pixel Grid

Which digit is it?

$x^{(1)} =$

Function: f

$y^{(1)} =$ "2"

28x28 pixels

$f(\mathbb{X}) \rightarrow \mathbb{Y}$

$x^{(2)} =$

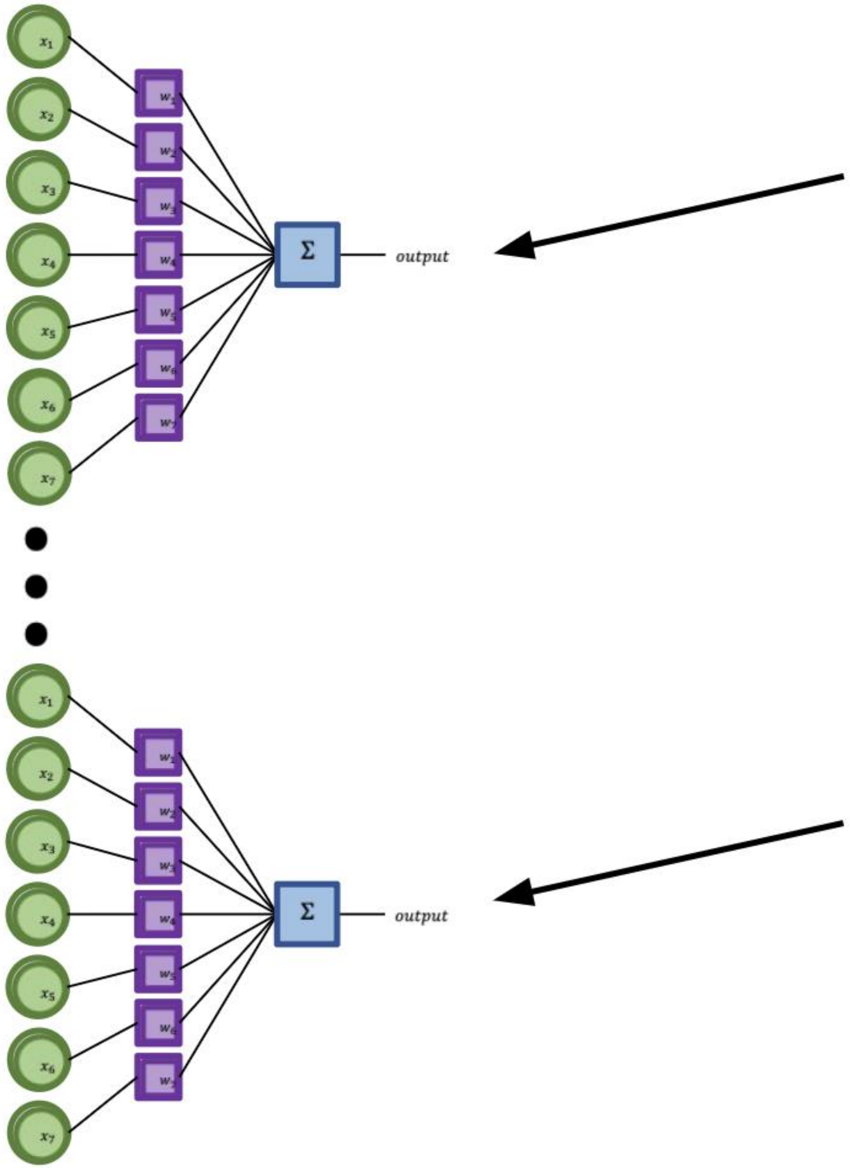
Instead of outputting a binary prediction, make an output for each class.

$y^{(2)} =$ "0"

Using Multiple Perceptrons

- We can use *m* perceptrons (where *m* is the number of output classes)
- For MNIST, this would be 10 perceptrons
- Each individual perceptron will need to return a value, our model will return the class with the highest value
  - Here, value refers to the weighted sum before the threshold is applied

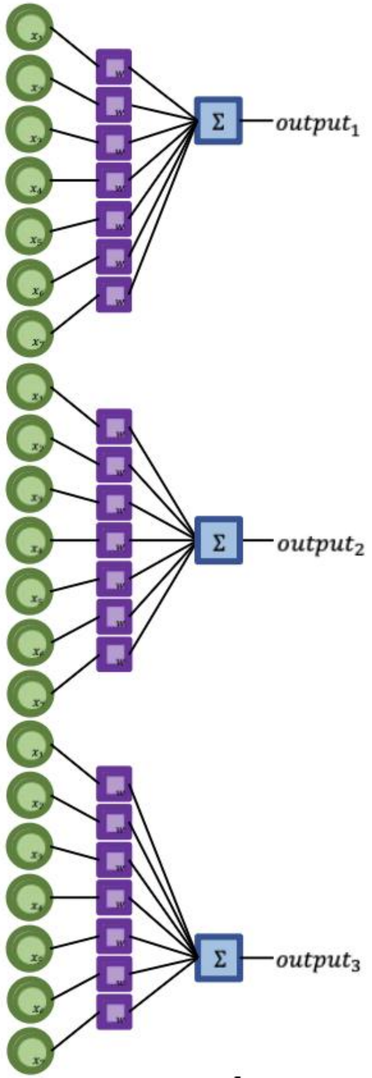
Using multiple perceptrons



Perceptron for predicting whether handwritten digit is a 0

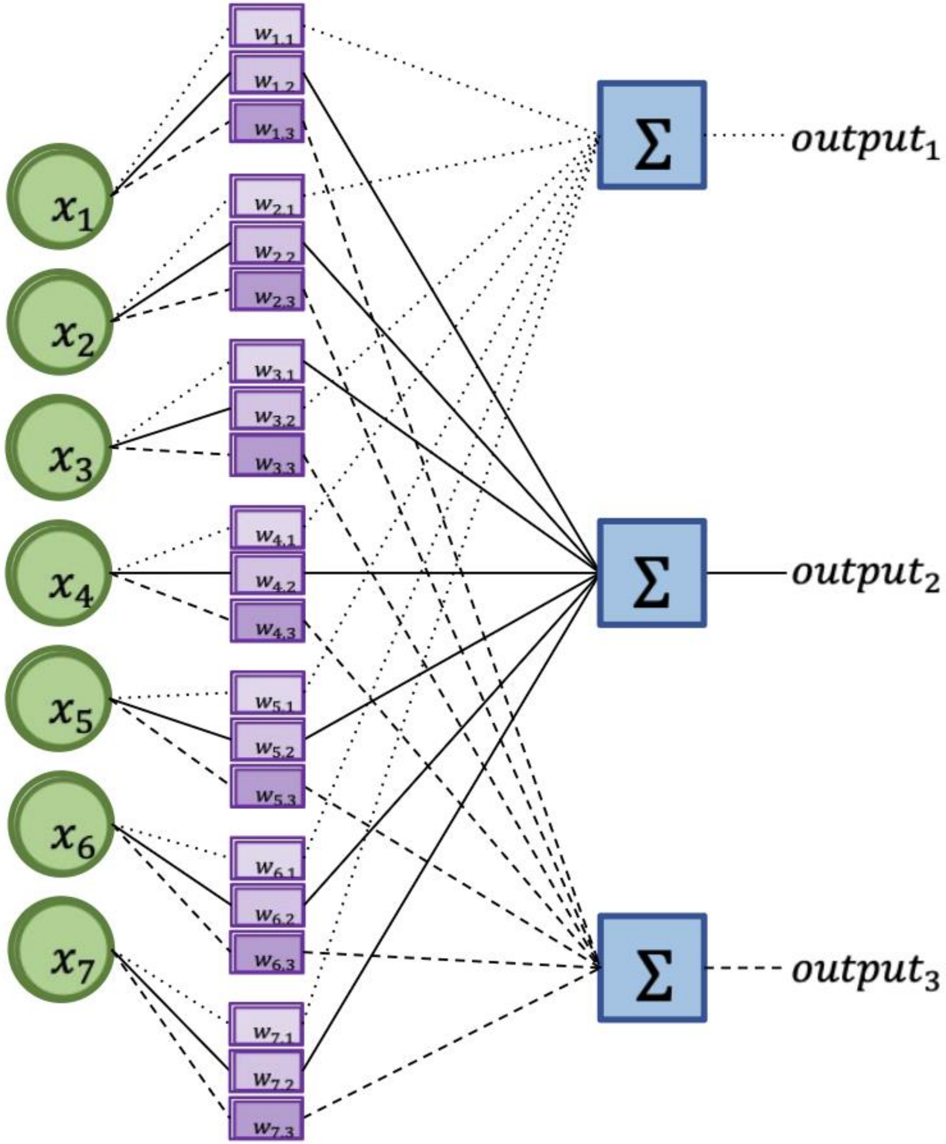
Perceptron for predicting whether handwritten digit is a 9

Multi-class Perceptron



Three separate perceptrons

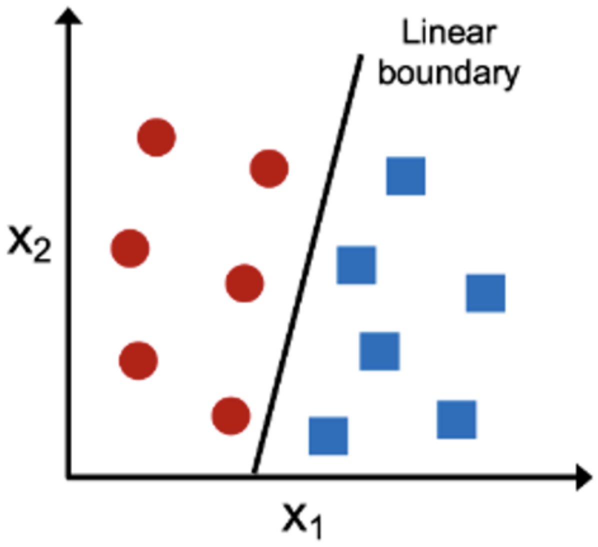
=



Three perceptrons sharing inputs

MNIST Performance

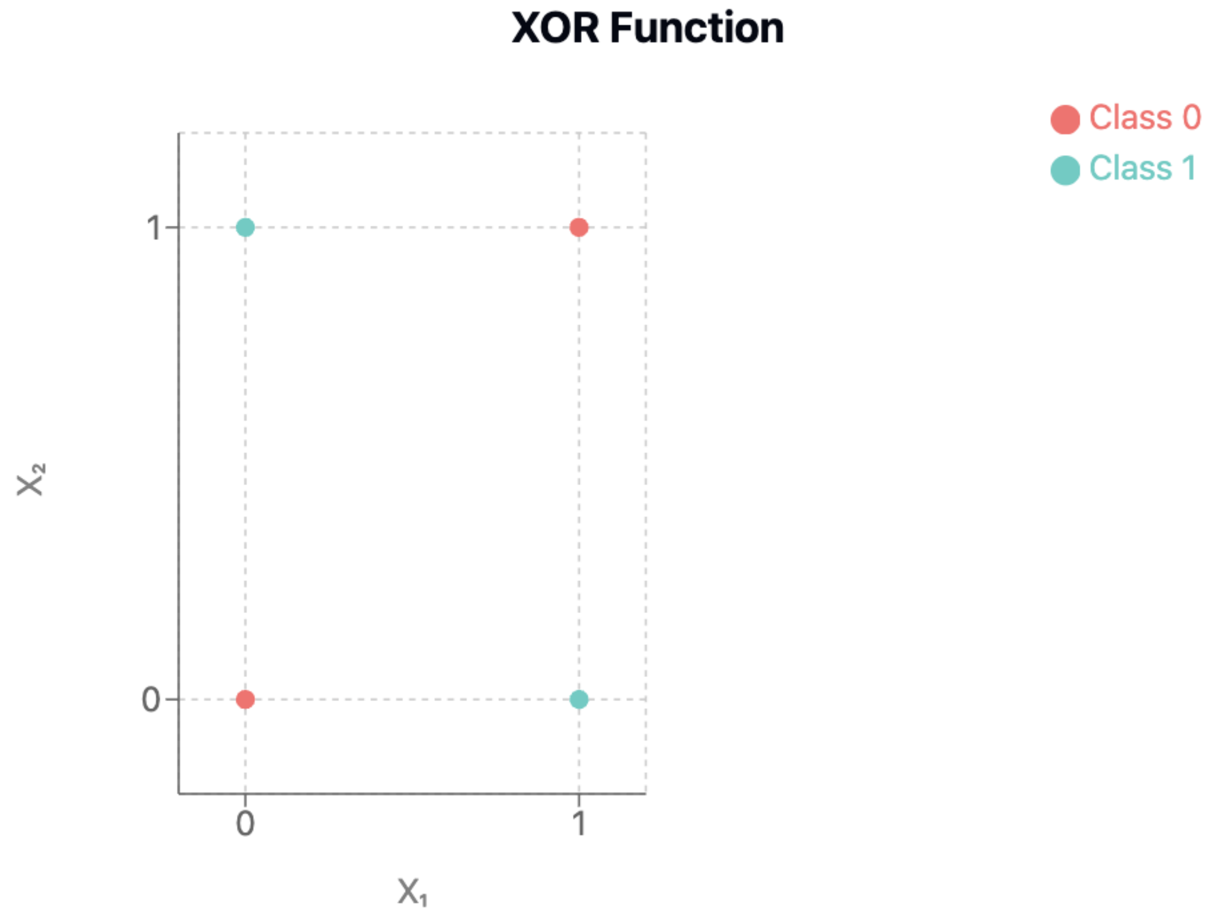
Perceptrons can perform quite well on MNIST, with around 85% accuracy



But they will always be linear classifiers…

Perceptrons

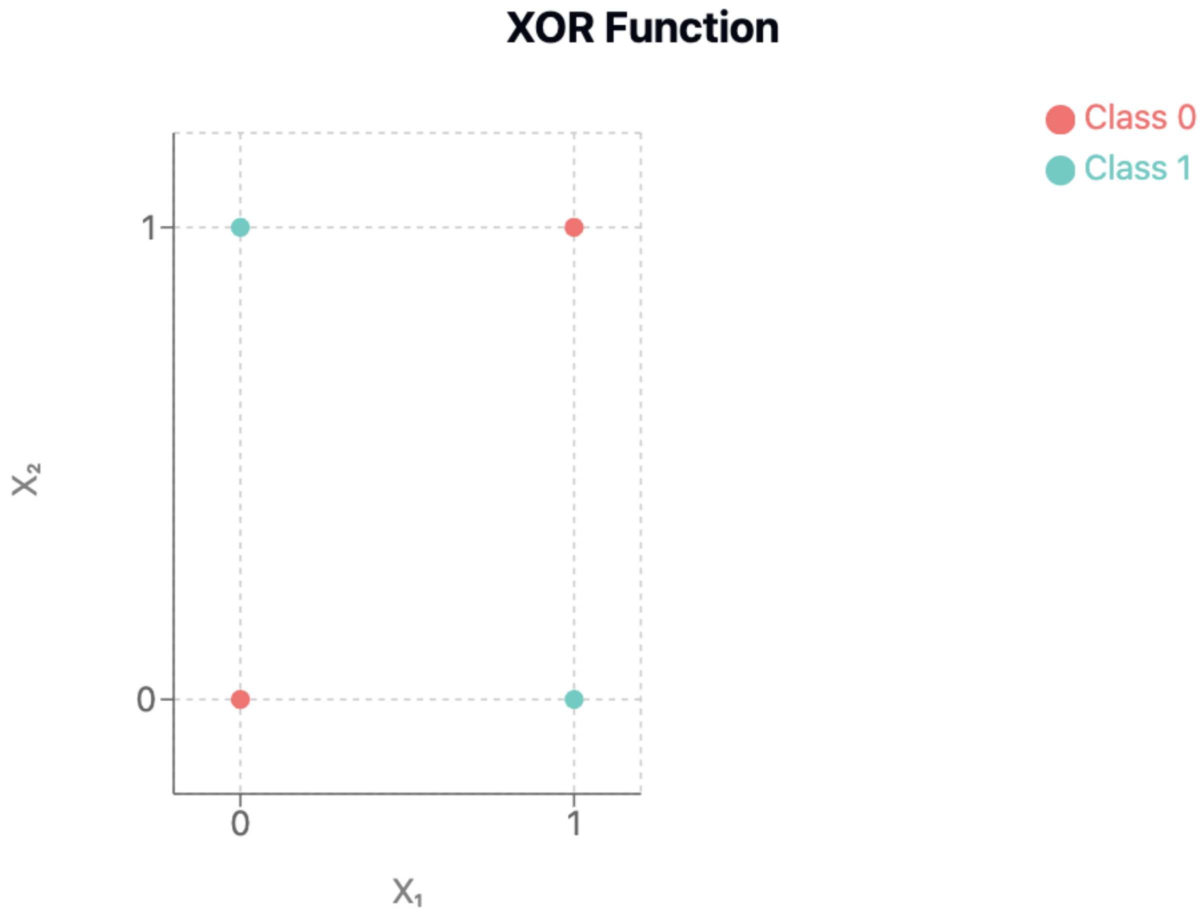
Are Perceptrons guaranteed to achieve 100% accuracy?



Perceptrons

Are Perceptrons guaranteed to achieve 100% accuracy?

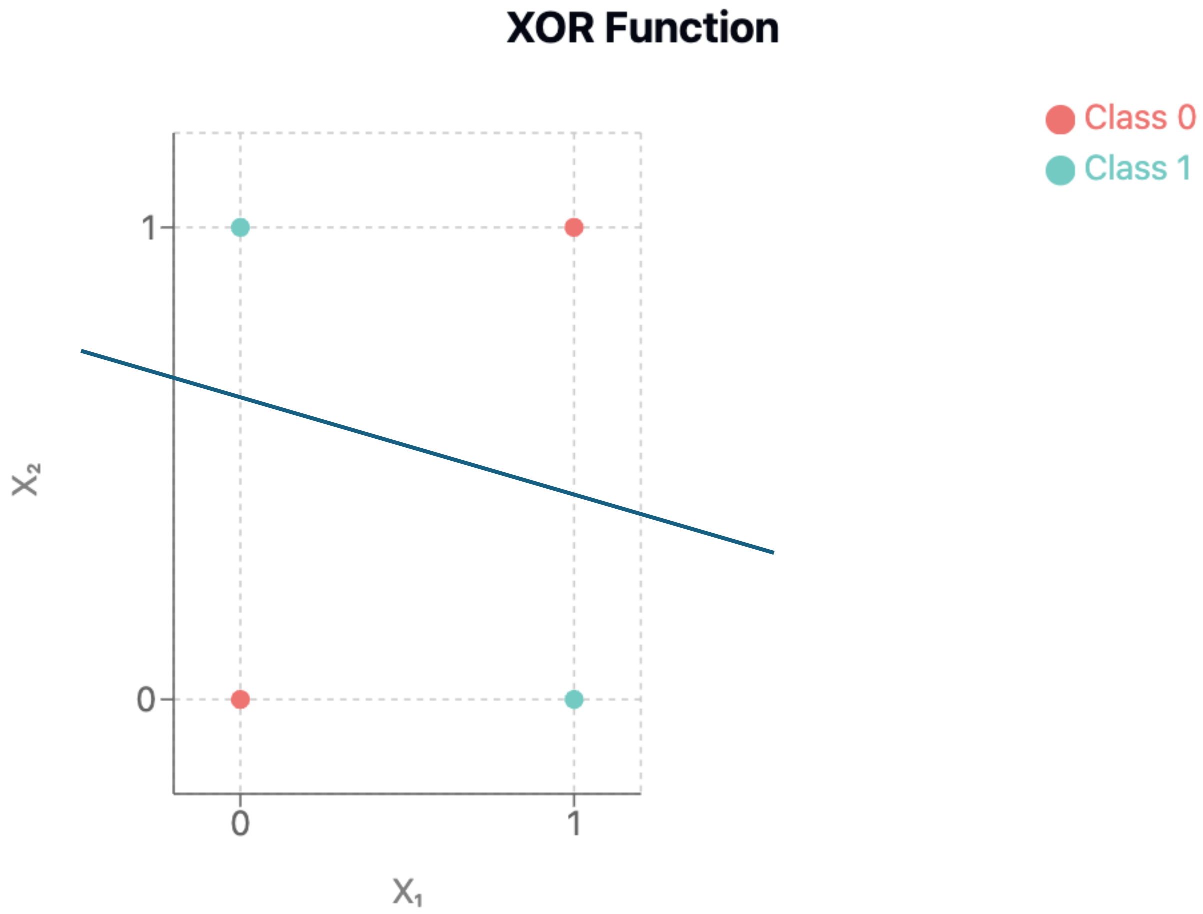
How can you put a linear separator on the plot to separate the two classes?



Perceptrons

Are Perceptrons guaranteed to achieve 100% accuracy?

How can you put a linear separator on the plot to separate the two classes?

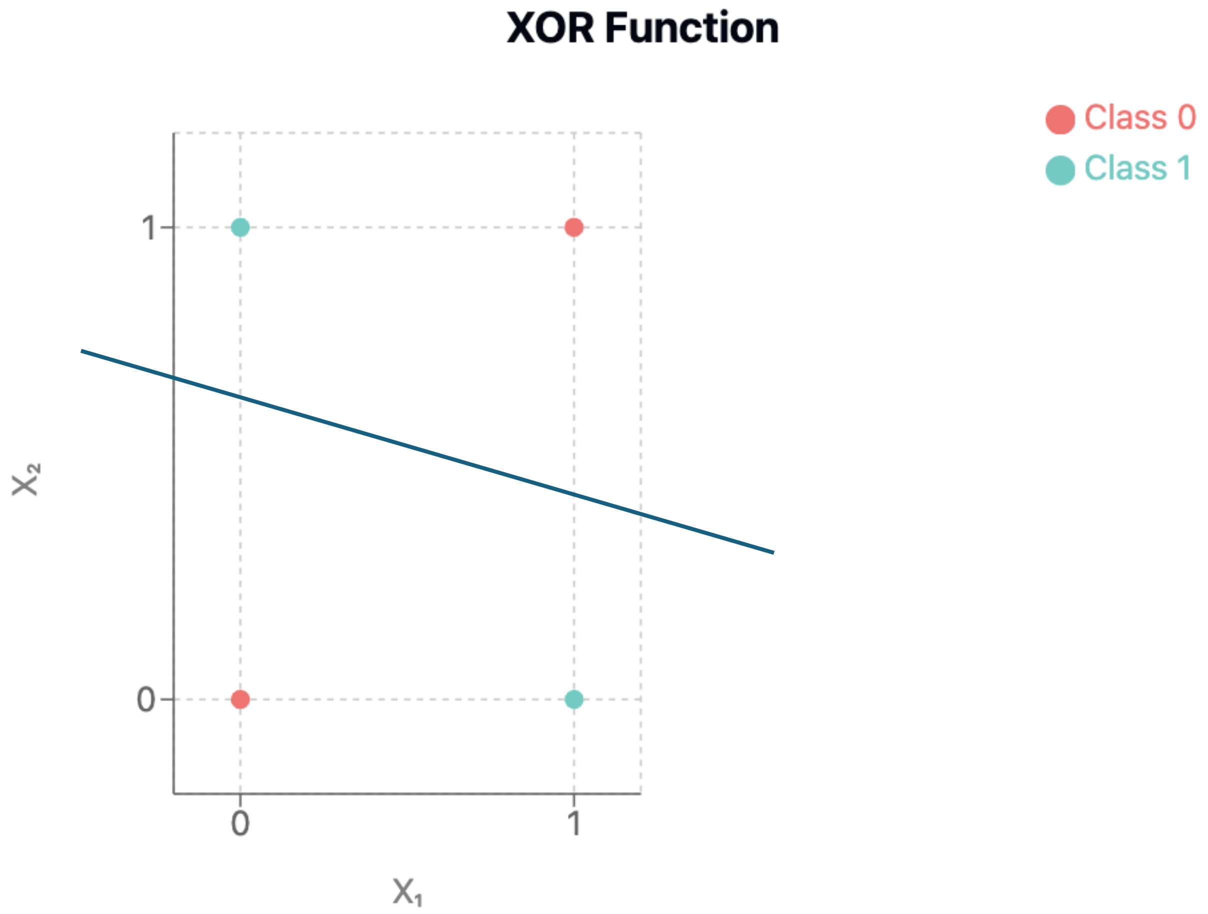


Perceptrons

Are Perceptrons guaranteed to achieve 100% accuracy?

How can you put a linear separator on the plot to separate the two classes?

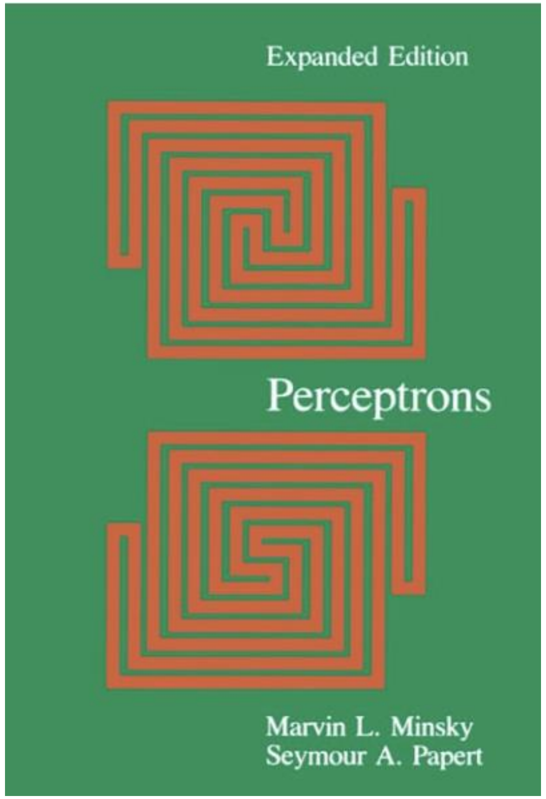
There are simple functions that perceptrons can't learn!



Perceptrons: The Book

Perceptrons: An Introduction to Computational Geometry

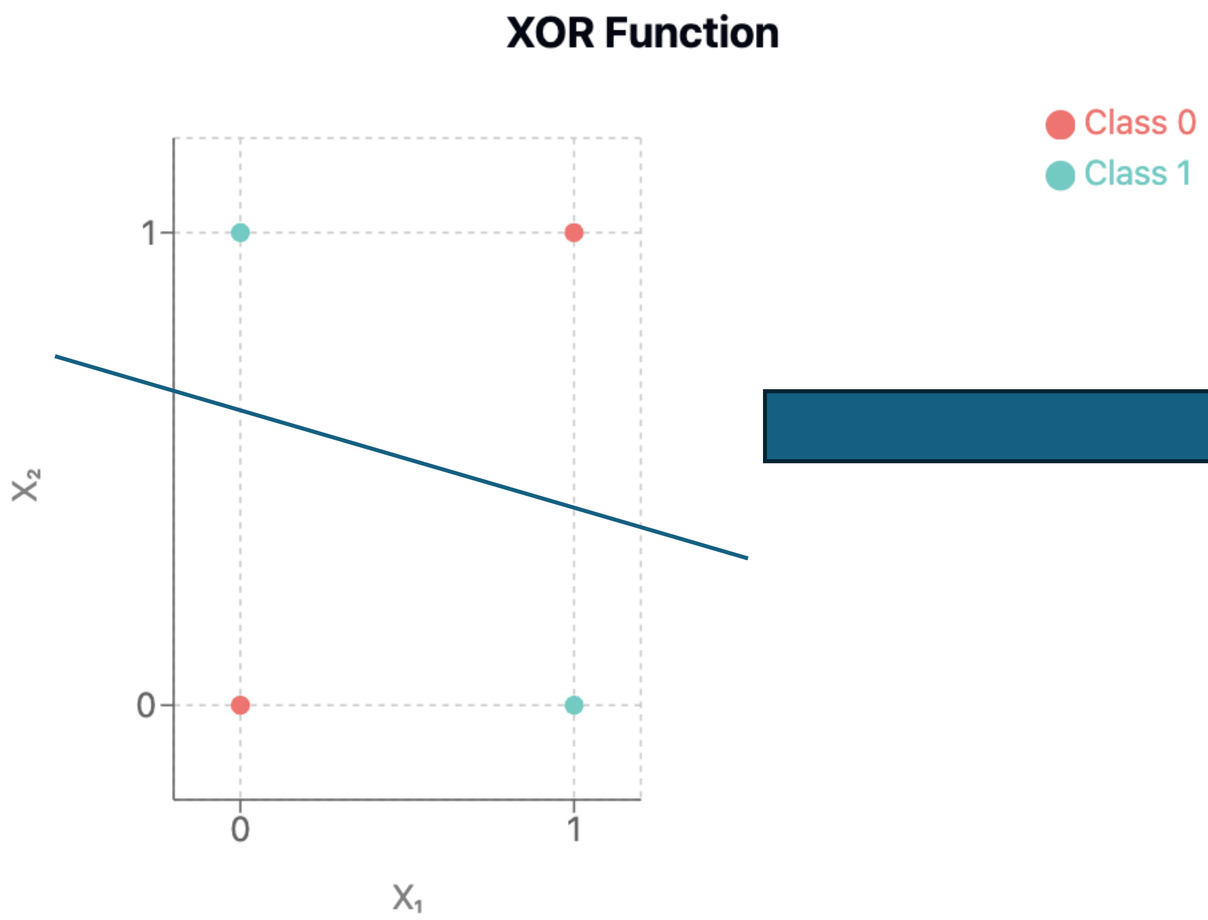
- Published by Marvin Minsky and Seymour Papert in 1969
- Acknowledged some strengths of Perceptrons and identified some fundamental flaws
- (Partially) responsible for shifting AI research away from "connectivism" and towards symbolic AI systems



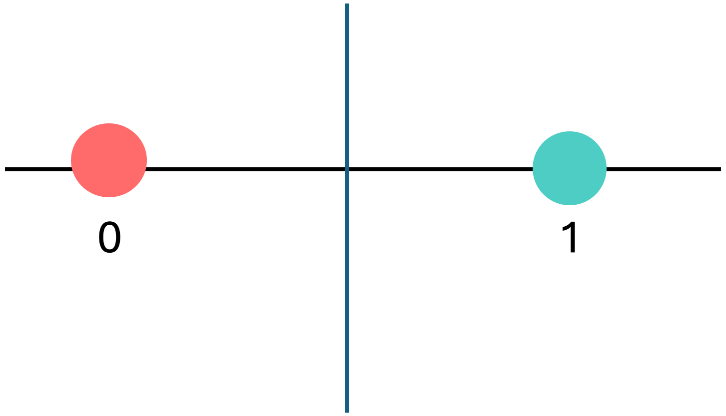
AI Research Timeline

Limited funding for neural networks research in the 1970s
(First AI winter)
1980s – revival of neural networks research
"Invention" of backpropagation, needed for efficient training of neural networks
1987 – collapse of LISP machine market and abandonment of expert systems
(Second AI winter)

The Solution:



Use hand-crafted feature:
$|x_1 - x_2|$

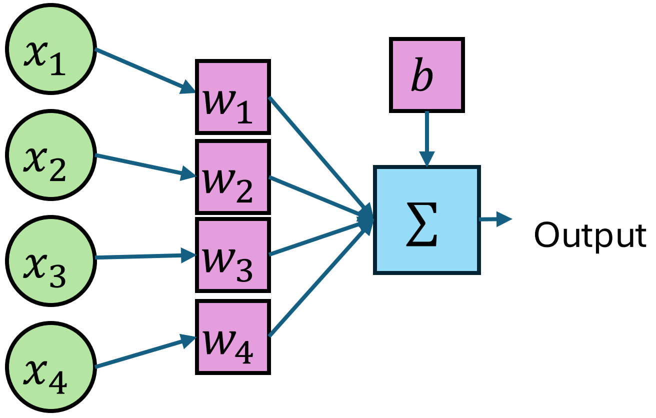


Trivial to find linear separator in new feature space!

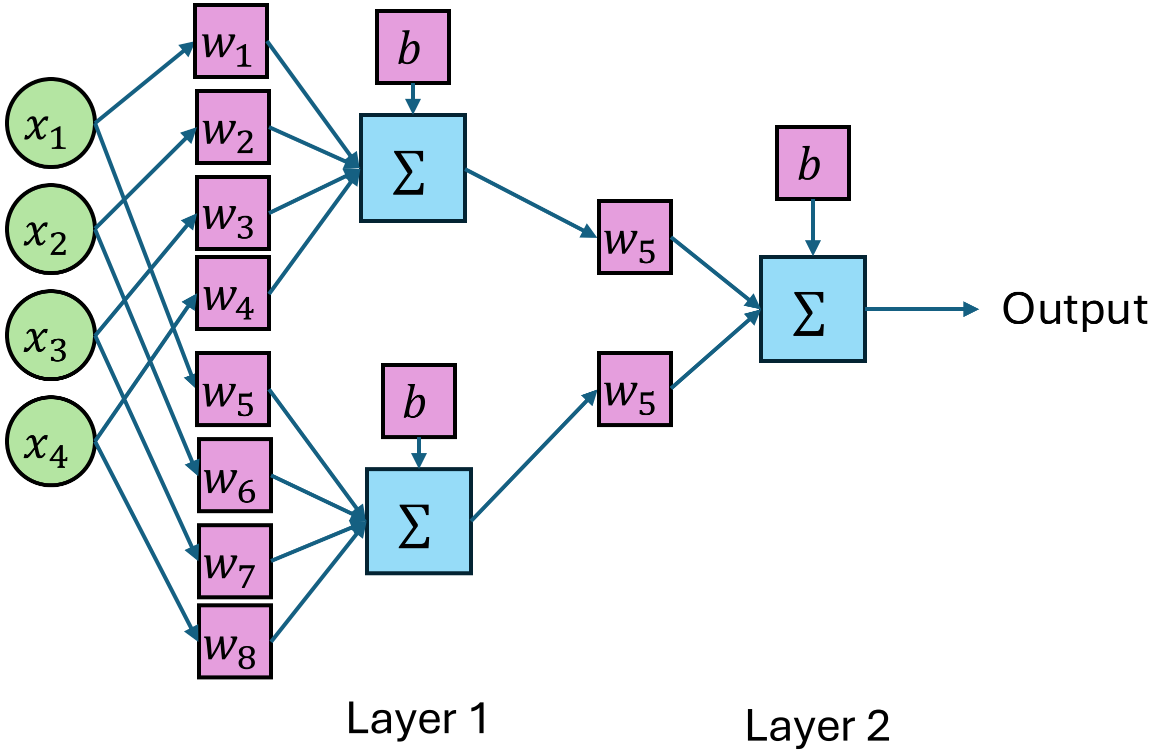
But how do we find appropriate features?

The Solution: learn new features

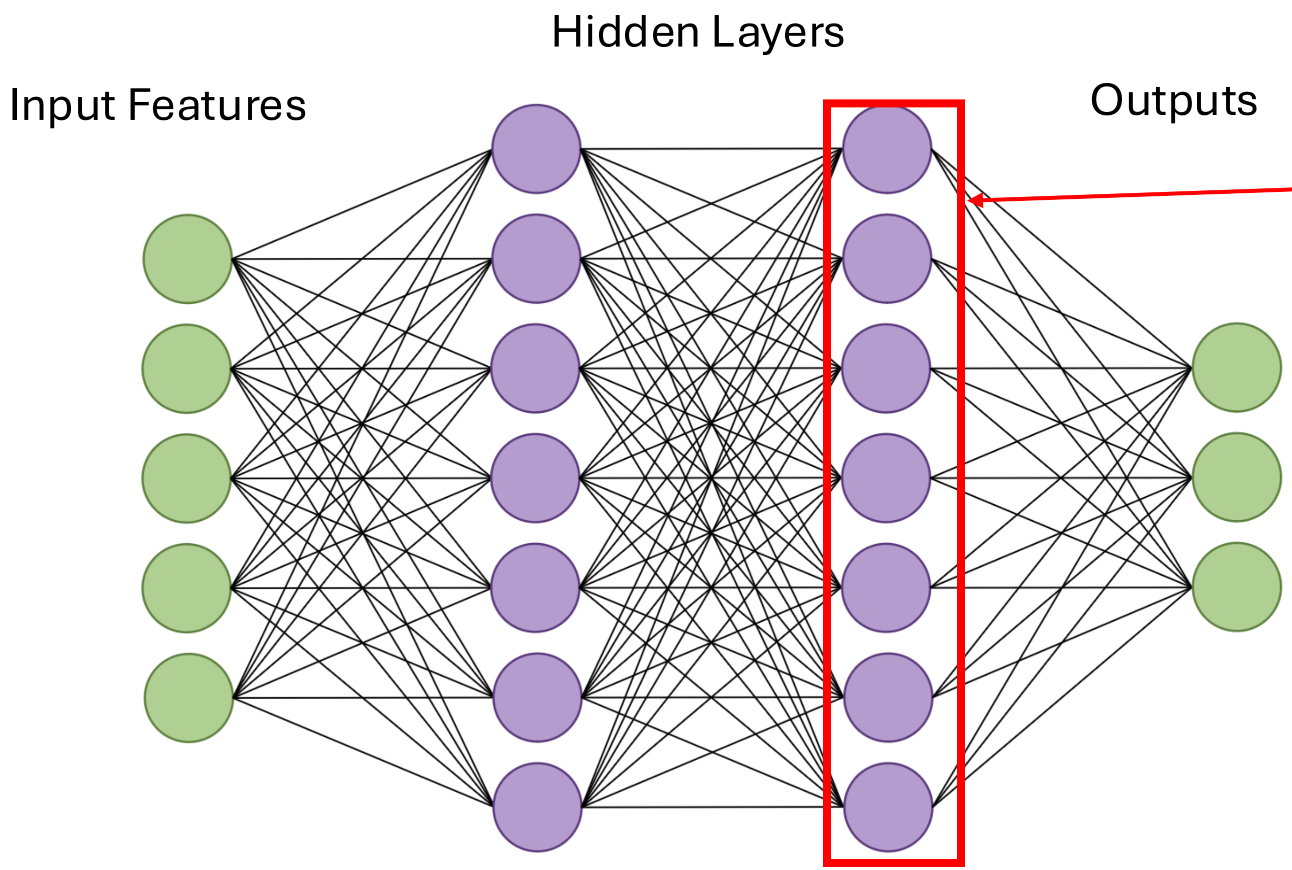
Perceptron



Multi-Layer Perceptron (MLP, Neural Network)



MLPs



A Multi-Layered Neural Net

Learned Representation:
Transformation of original features into new “learned” features

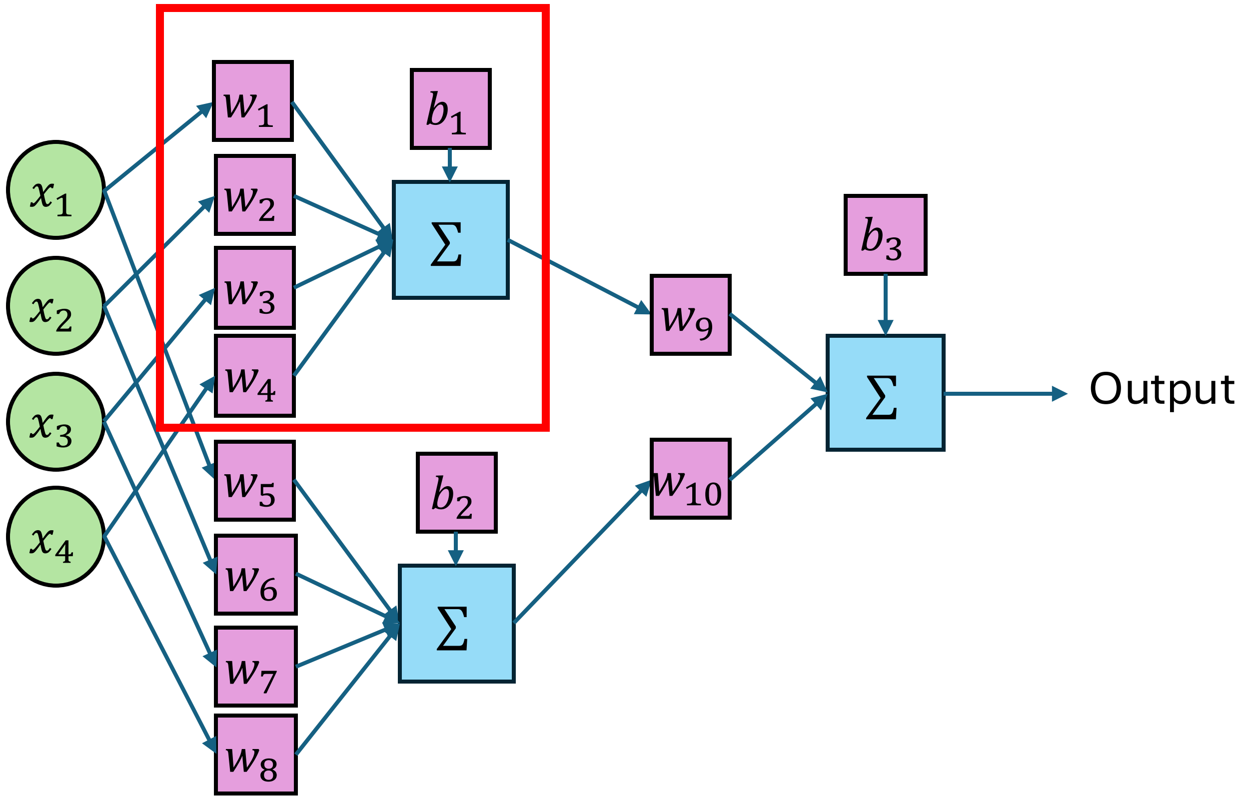
MLPs consist of weights, biases, and activation functions for each neuron.

Goal (for binary-classification): Learn intermediate features, such that the final representation is linearly separable

Multi-Layer Perceptrons

What happens if we remove the threshold activations from a multi-layer perceptron?

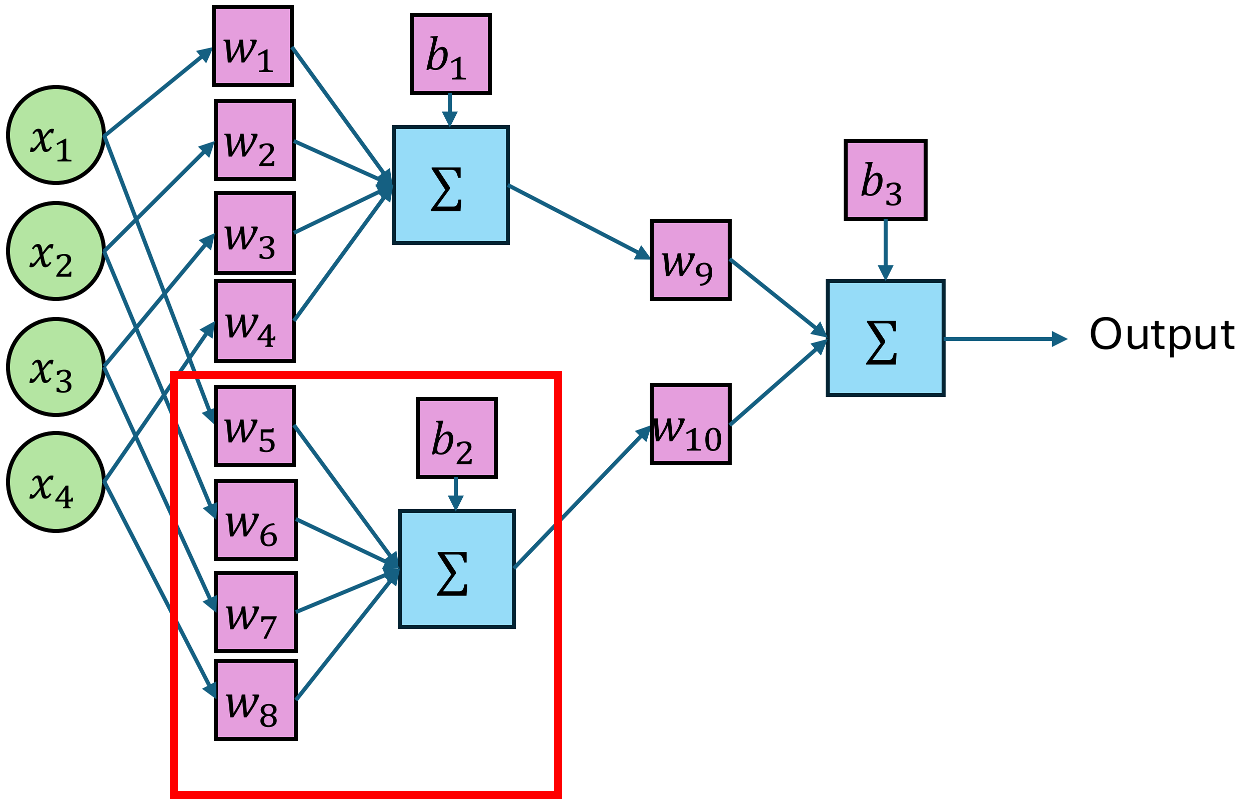
Let $\mathrm{w}^{(1)} = [w_1, w_2, w_3, w_4]$
Perceptron #1: $\mathrm{z}_1 = x^T w^{(1)} + b_1$



Multi-Layer Perceptrons

What happens if we remove the activations from a multi-layer perceptron?

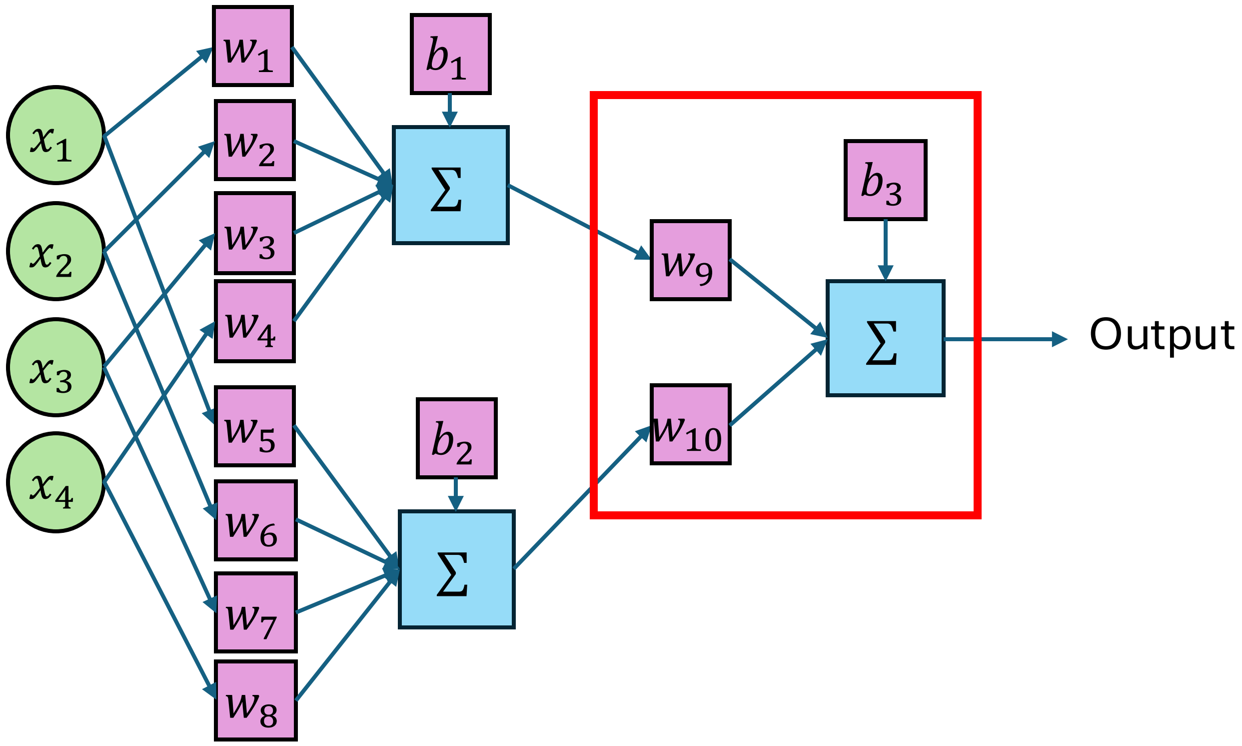
Let $\mathrm{w}^{(2)} = [w_5, w_6, w_7, w_8]$
Perceptron #1: $\mathrm{z}_1 = x^T w^{(1)} + b_1$
Perceptron #2: $z_2 = x^T w^{(2)} + b_2$



Multi-Layer Perceptrons

What happens if we remove the activations from a multi-layer perceptron?

Let $\mathrm{w}^{(2)} = [w_5, w_6, w_7, w_8]$
Perceptron #1: $\mathrm{z}_1 = x^T w^{(1)} + b_1$
Perceptron #2: $z_2 = x^T w^{(2)} + b_2$
Perceptron #3: $z_3 = z_1 w_9 + z_2 w_{10} + b_3$

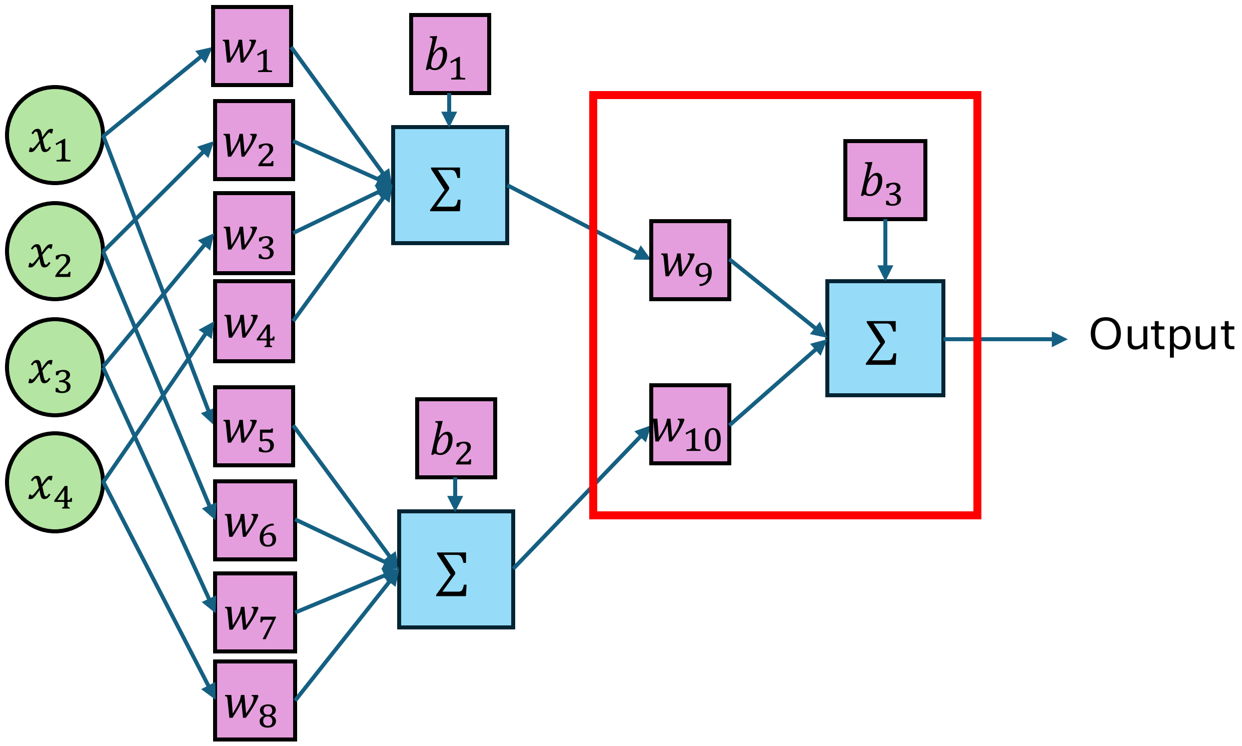


Multi-Layer Perceptrons

What happens if we remove the activations from a multi-layer perceptron?

Let $\mathrm{w}^{(2)} = [w_5, w_6, w_7, w_8]$
Perceptron #1: $z_1 = x^T w^{(1)} + b_1$
Perceptron #2: $z_2 = x^T w^{(2)} + b_2$
Perceptron #3: $z_3 = z_1 w_9 + z_2 w_{10} + b_3$

Entire Network:
$w_9\left(x^T w^{(1)} + b_1\right) + w_{10}\left(x^T w^{(2)} + b_2\right) + b_3$

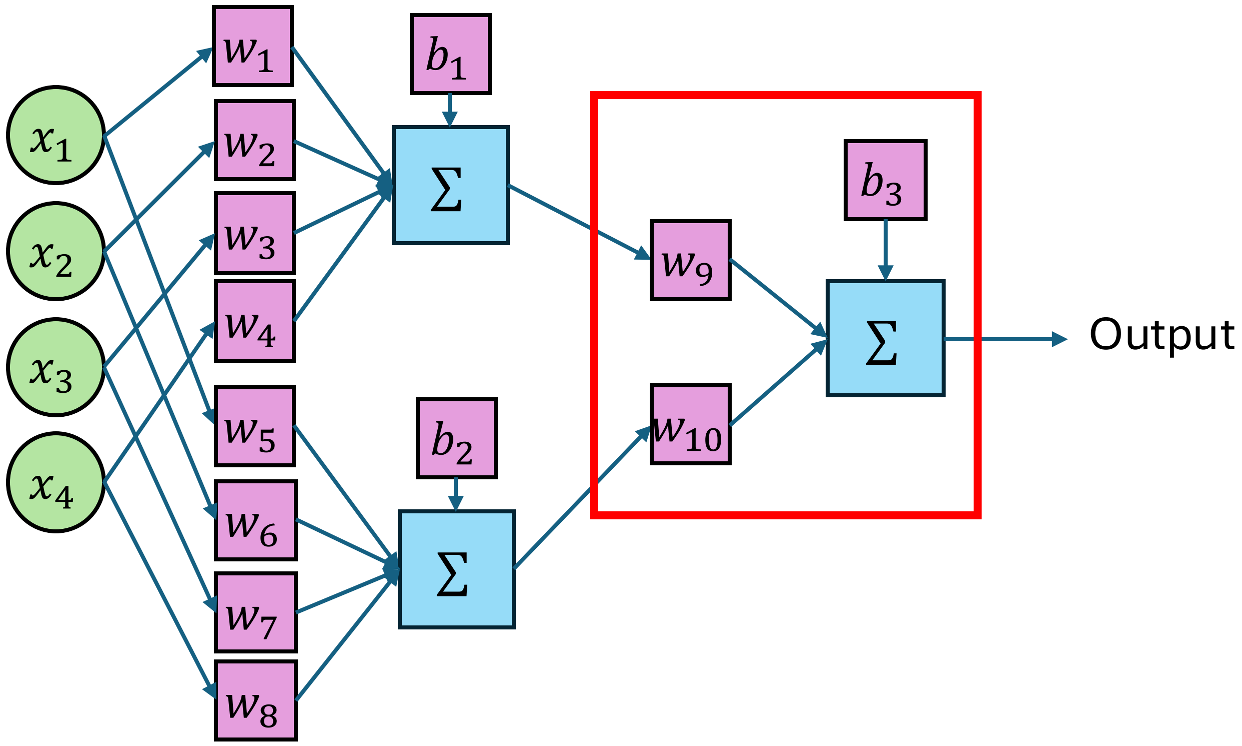


Multi-Layer Perceptrons

What happens if we remove the activations from a multi-layer perceptron?

Let $w^{(2)} = [w_5, w_6, w_7, w_8]$
Perceptron #1: $z_1 = x^T w^{(1)} + b_1$
Perceptron #2: $z_2 = x^T w^{(2)} + b_2$
Perceptron #3: $z_3 = z_1 w_9 + z_2 w_{10} + b_3$

Entire Network:
$z = w_9\left(x^T w^{(1)} + b_1\right) + w_{10}\left(x^T w^{(2)} + b_2\right) + b_3$



With no activation function

MLP’s Expressiveness

With no activation function

MLP's Expressiveness

$$z = w_9\left(x^T w^{(1)} + b_1\right) + w_{10}\left(x^T w^{(2)} + b_2\right) + b_3$$

With no activation function

MLP's Expressiveness

$$z = w_9\left(x^T w^{(1)} + b_1\right) + w_{10}\left(x^T w^{(2)} + b_2\right) + b_3$$
$$z = w_9 x^T w^1 + w_9 \cdot b_1 + w_{10} x^T w^{(2)} + w_{10} b_2 + b_3$$

With no activation function

MLP's Expressiveness

$$z = w_9\left(x^T w^{(1)} + b_1\right) + w_{10}\left(x^T w^{(2)} + b_2\right) + b_3$$
$$z = w_9 x^T w^{(1)} + w_9 \cdot b_1 + w_{10} x^T w^{(2)} + w_{10} b_2 + b_3$$
$$z = x^T (w_9 w^{(1)} + w_{10} w^{(2)}) + (w_9 \cdot b_1 + + w_{10} b_2 + b_3)$$

With no activation function

MLP’s Expressiveness

$$z = w_9\left(x^T w^{(1)} + b_1\right) + w_{10}\left(x^T w^{(2)} + b_2\right) + b_3$$
$$z = w_9 x^T w^1 + w_9 \cdot b_1 + w_{10} x^T w^{(2)} + w_{10} b_2 + b_3$$
$$z = x^T \boxed{(w_9 w^1 + w_{10} w^{(2)})} + (w_9 \cdot b_1 + + w_{10} b_2 + b_3)$$

With no activation function

MLP's Expressiveness

$$z = w_9\left(x^T w^{(1)} + b_1\right) + w_{10}\left(x^T w^{(2)} + b_2\right) + b_3$$
$$z = w_9 x^T w^1 + w_9 \cdot b_1 + w_{10} x^T w^{(2)} + w_{10} b_2 + b_3$$
$$z = x^T \boxed{(w_9 w^1 + w_{10} w^{(2)})} + (w_9 \cdot b_1 + + w_{10} b_2 + b_3)$$

Just a vector...

With no activation function

MLP's Expressiveness

$$z = w_9\left(x^T w^{(1)} + b_1\right) + w_{10}\left(x^T w^{(2)} + b_2\right) + b_3$$

$$z = w_9 x^T w^1 + w_9 \cdot b_1 + w_{10} x^T w^{(2)} + w_{10} b_2 + b_3$$

$$z = x^T \boxed{(w_9 w^1 + w_{10} w^{(2)})} + \boxed{(w_9 \cdot b_1 + + w_{10} b_2 + b_3)}$$

Just a vector...

With no activation function

MLP’s Expressiveness

$$z = w_9\left(x^T w^{(1)} + b_1\right) + w_{10}\left(x^T w^{(2)} + b_2\right) + b_3$$

$$z = w_9 x^T w^1 + w_9 \cdot b_1 + w_{10} x^T w^{(2)} + w_{10} b_2 + b_3$$

$$z = x^T \underbrace{\boxed{(w_9 w^1 + w_{10} w^{(2)})}}_{\text{Just a vector...}} + \underbrace{\boxed{(w_9 \cdot b_1 + + w_{10} b_2 + b_3)}}_{\text{Just a scalar...}}$$

With no activation function

MLP’s Expressiveness

$$z = w_9\left(x^T w^{(1)} + b_1\right) + w_{10}\left(x^T w^{(2)} + b_2\right) + b_3$$

$$z = w_9 x^T w^1 + w_9 \cdot b_1 + w_{10} x^T w^{(2)} + w_{10} b_2 + b_3$$

$$z = x^T \underbrace{\boxed{(w_9 w^1 + w_{10} w^{(2)})}}_{\text{Just a vector...}} + \underbrace{\boxed{(w_9 \cdot b_1 + + w_{10} b_2 + b_3)}}_{\text{Just a scalar...}}$$

$$\boxed{z = x^T \vec{w} + b}$$

With no activation function

MLP's Expressiveness

$$z = w_9\left(x^T w^{(1)} + b_1\right) + w_{10}\left(x^T w^{(2)} + b_2\right) + b_3$$

$$z = w_9 x^T w^1 + w_9 \cdot b_1 + w_{10} x^T w^{(2)} + w_{10} b_2 + b_3$$

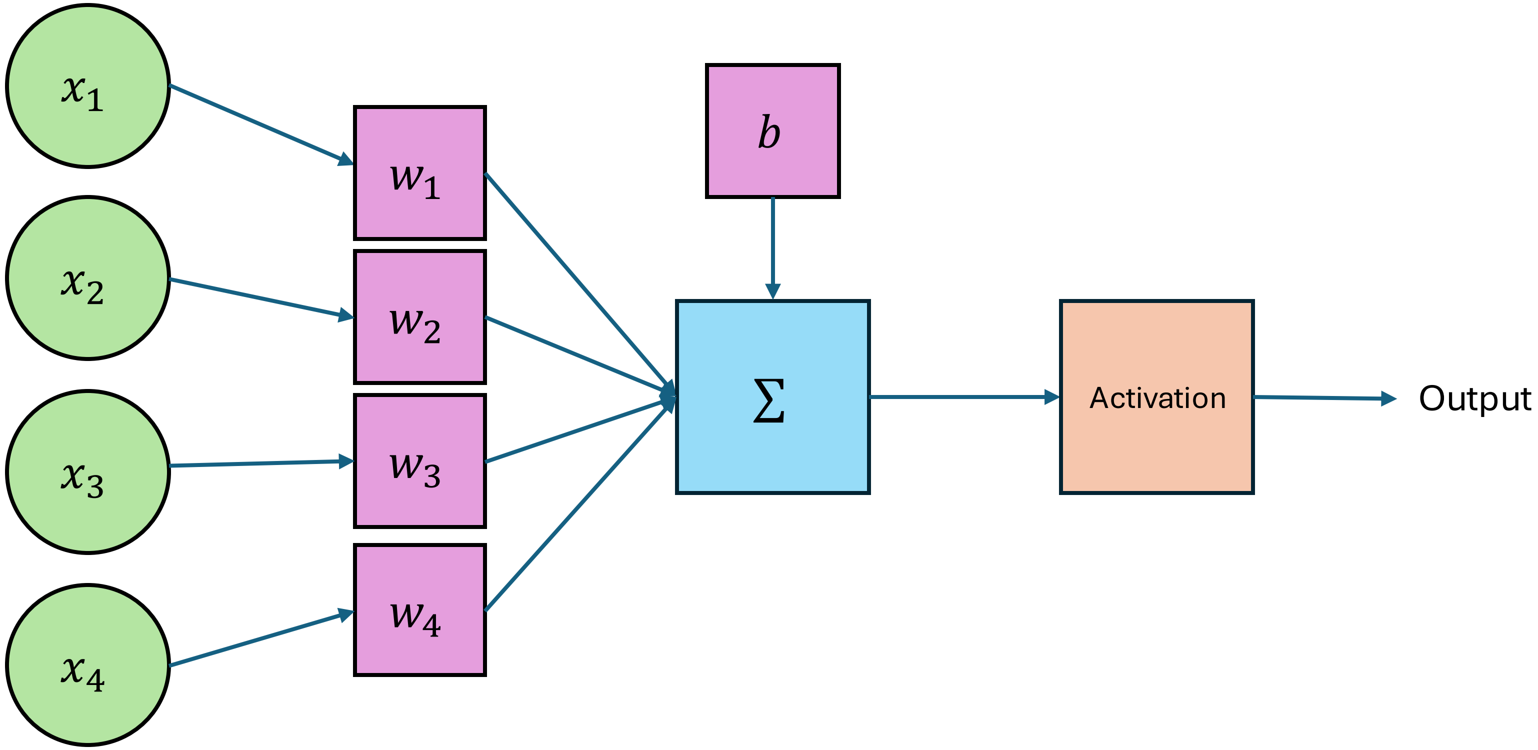
$$z = x^T \underbrace{\boxed{(w_9 w^1 + w_{10} w^{(2)})}}_{\text{Just a vector...}} + \underbrace{\boxed{(w_9 \cdot b_1 + + w_{10} b_2 + b_3)}}_{\text{Just a scalar...}}$$

$$\boxed{z = x^T \vec{w} + b}$$

Multi-Layer Perceptrons without non-linear activation functions are linear functions

Activation Functions

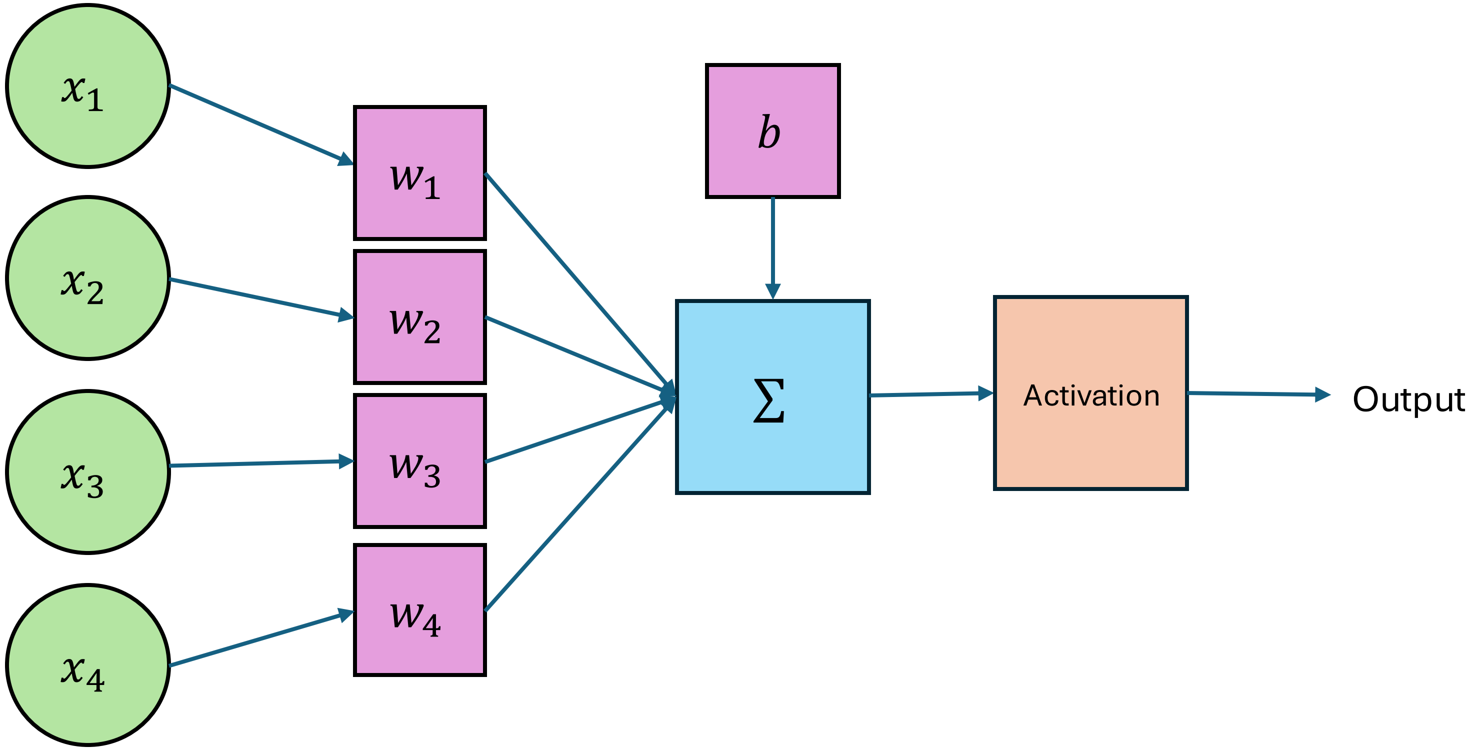
Non-linear functions applied to output of neuron



Activation Functions

Non-linear functions applied to output of neuron

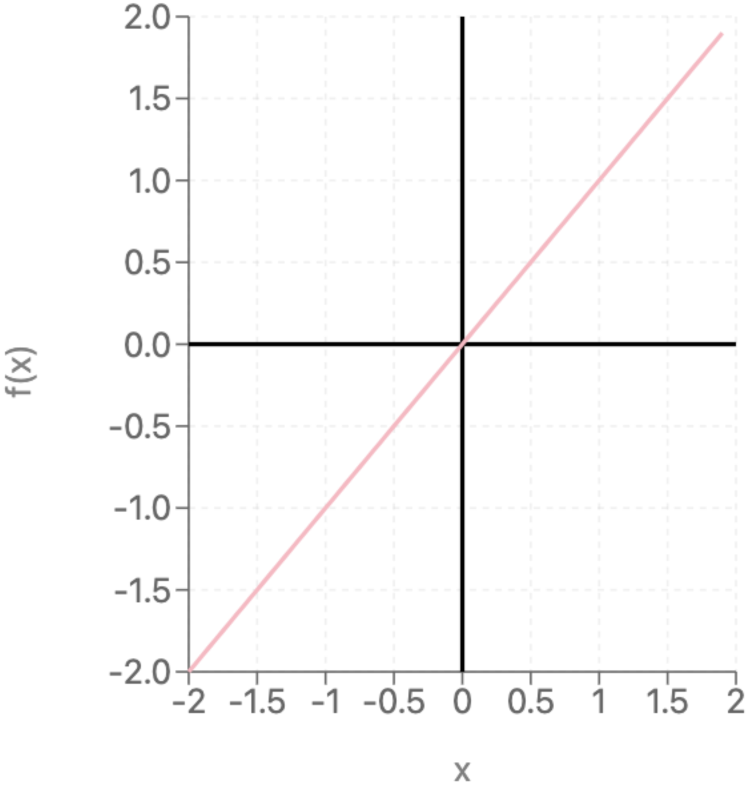
In the perceptron case, the activation function is the threshold



Common Activation Functions

Linear (No Activation)

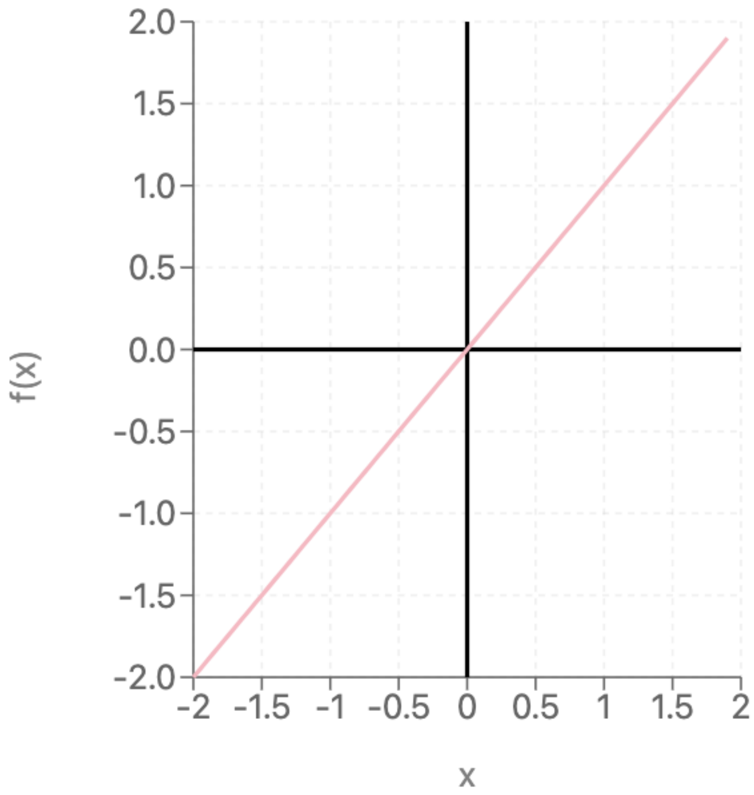
f(x) = x



Common Activation Functions

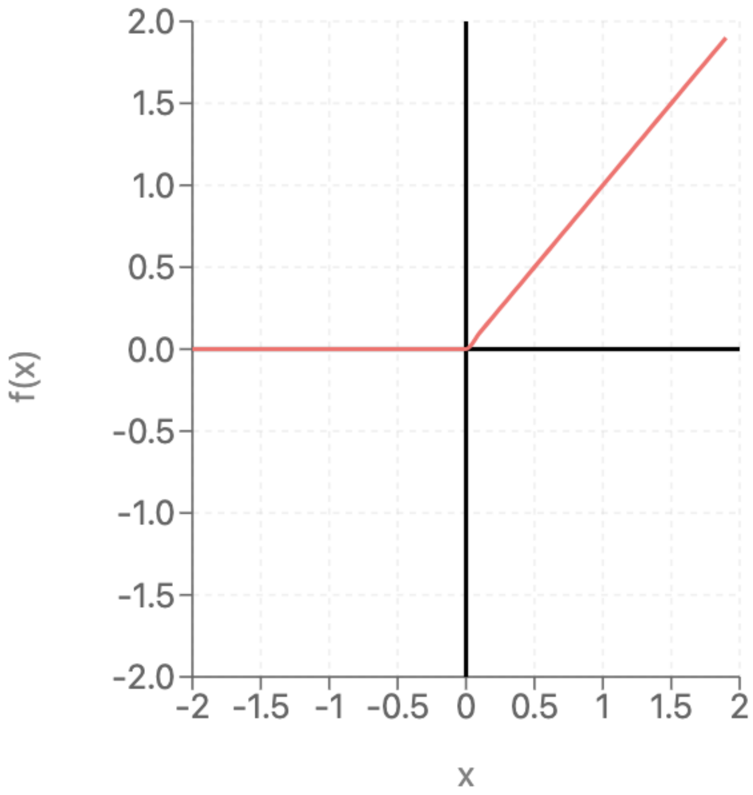
Linear (No Activation)

f(x) = x



ReLU

f(x) = max(0, x)

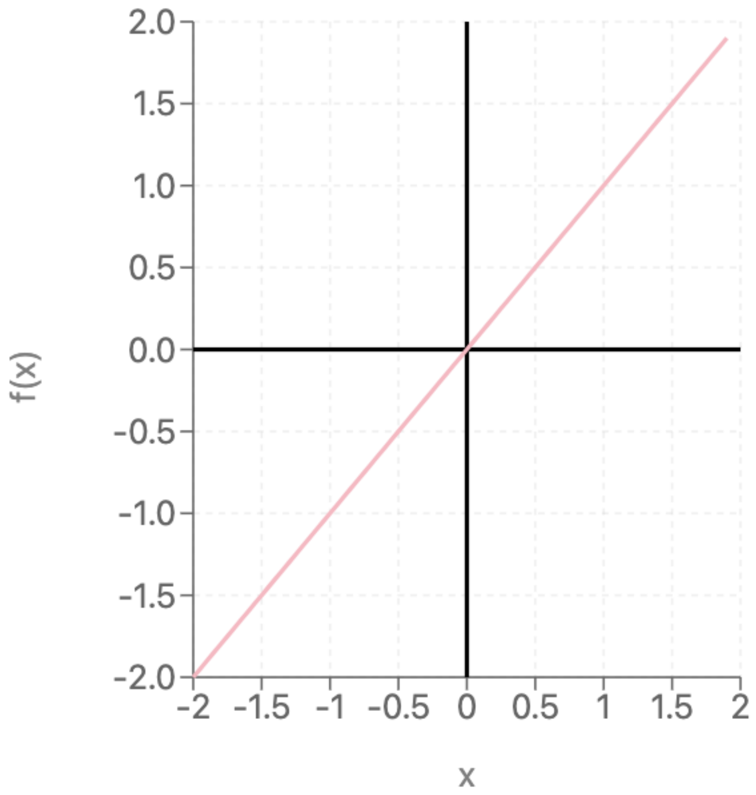


Common Activation Functions

Rectified Linear Unit (ReLU):
One of the most common Activation Functions
Advantages: Simple, easy to compute gradients

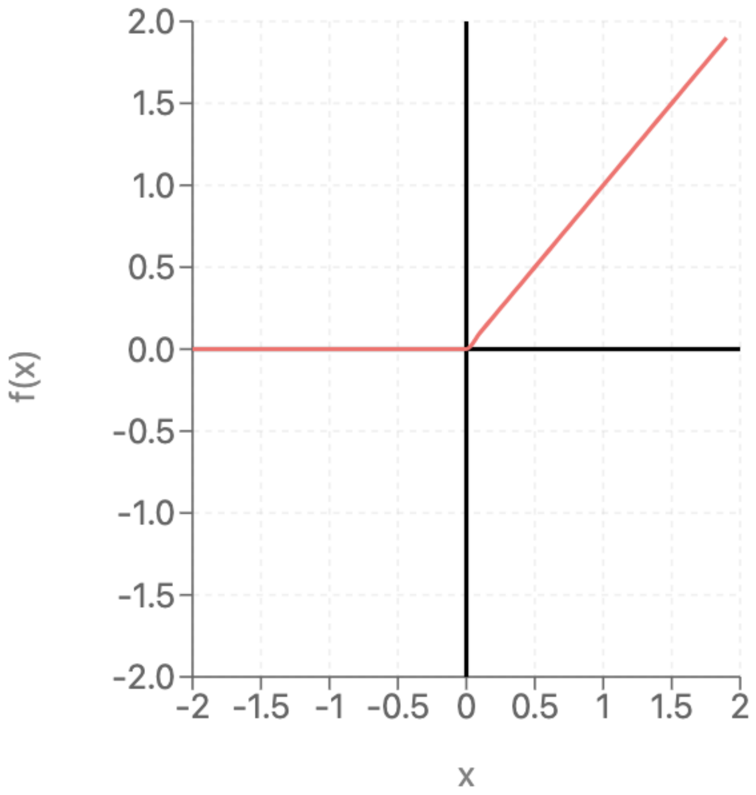
Linear (No Activation)

f(x) = x



ReLU

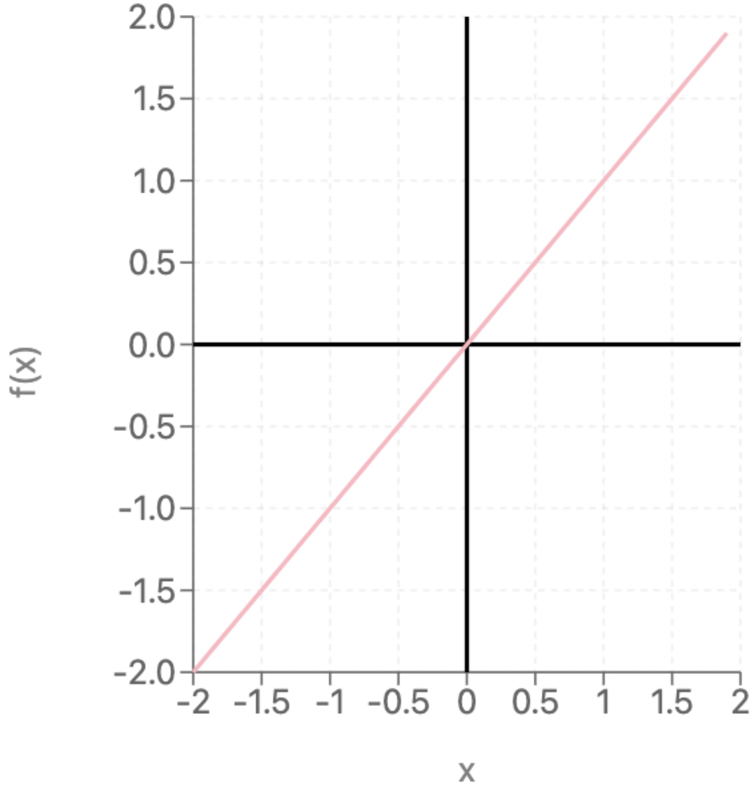
f(x) = max(0, x)



Common Activation Functions

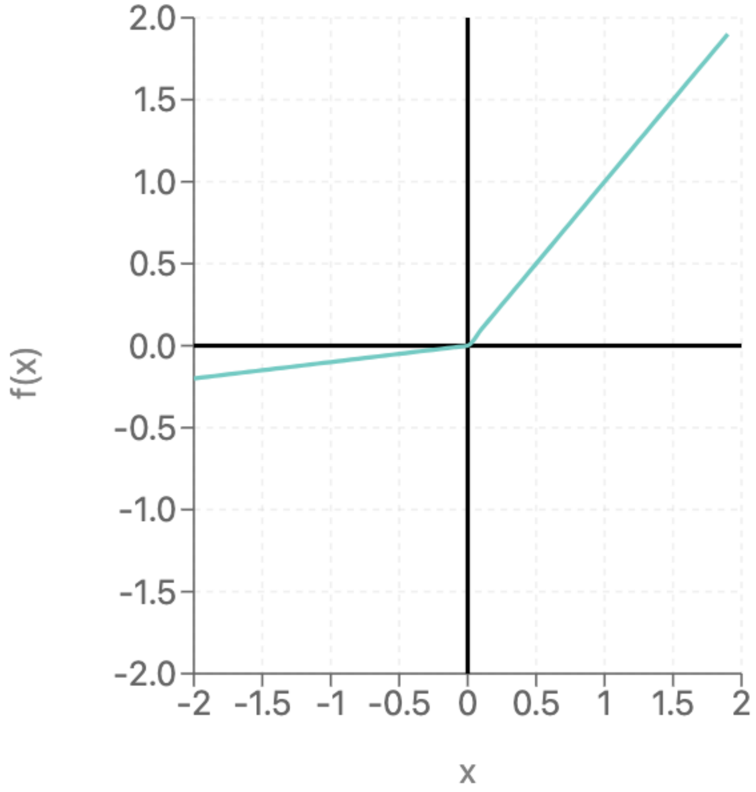
Linear (No Activation)

f(x) = x



Leaky ReLU

f(x) = x if x > 0 else 0.1x

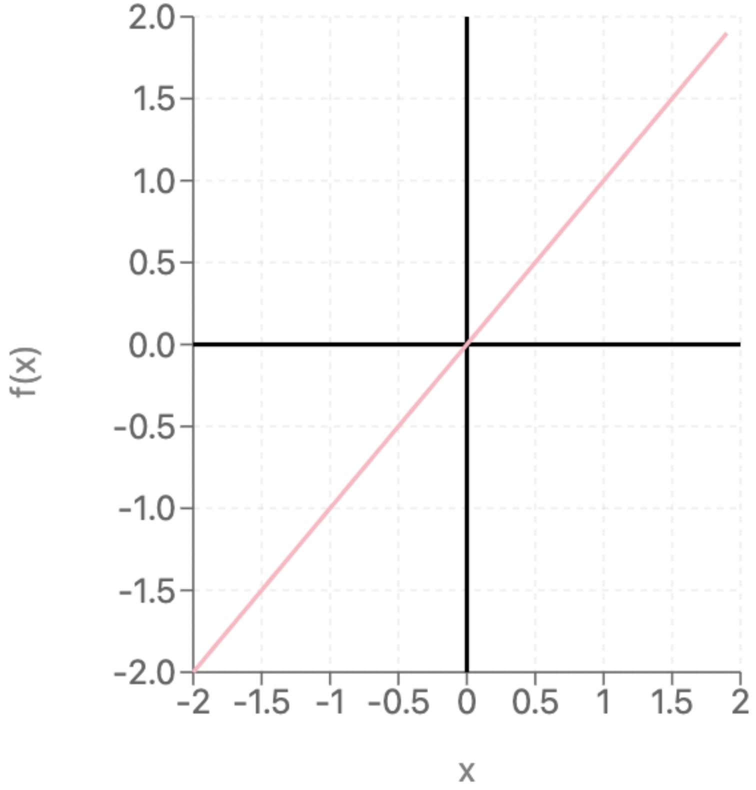


Common Activation Functions

Leaky ReLU:
Common substitute for ReLU, often has better performance
Advantages: Fixes "dying neurons" issue with ReLU.

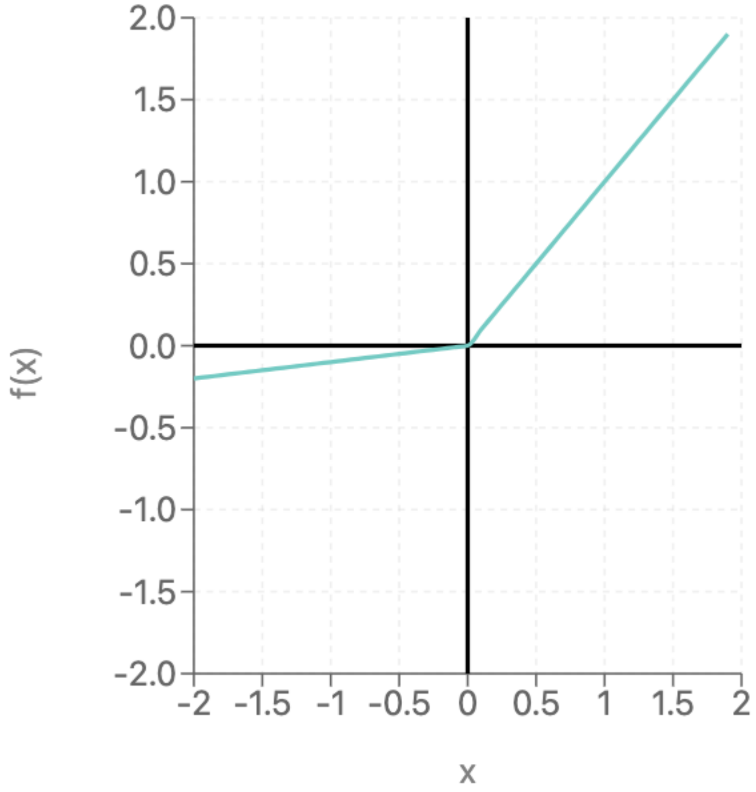
Linear (No Activation)

f(x) = x

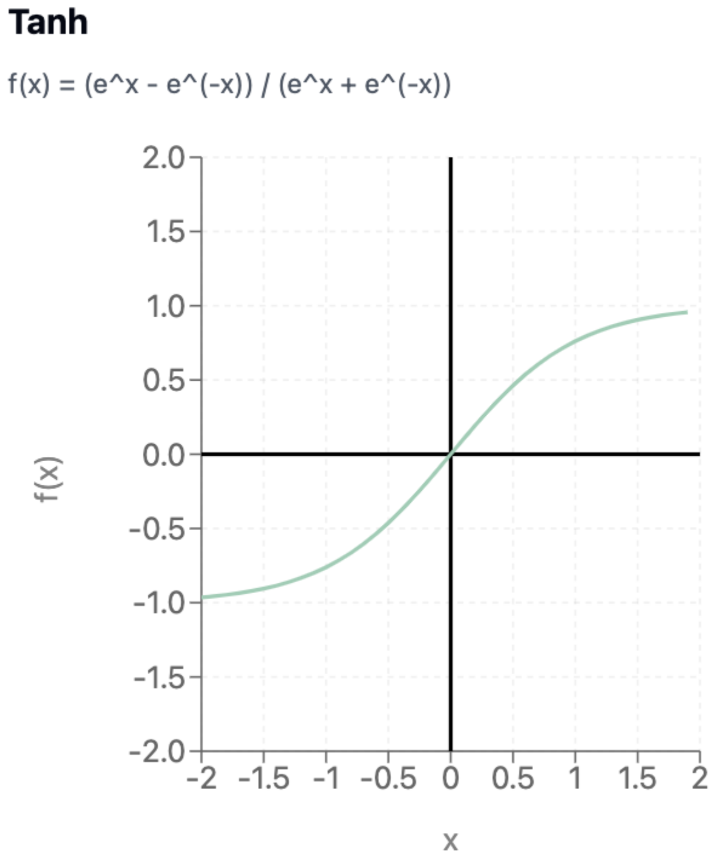


Leaky ReLU

f(x) = x if x > 0 else 0.1x



Tanh



Tanh

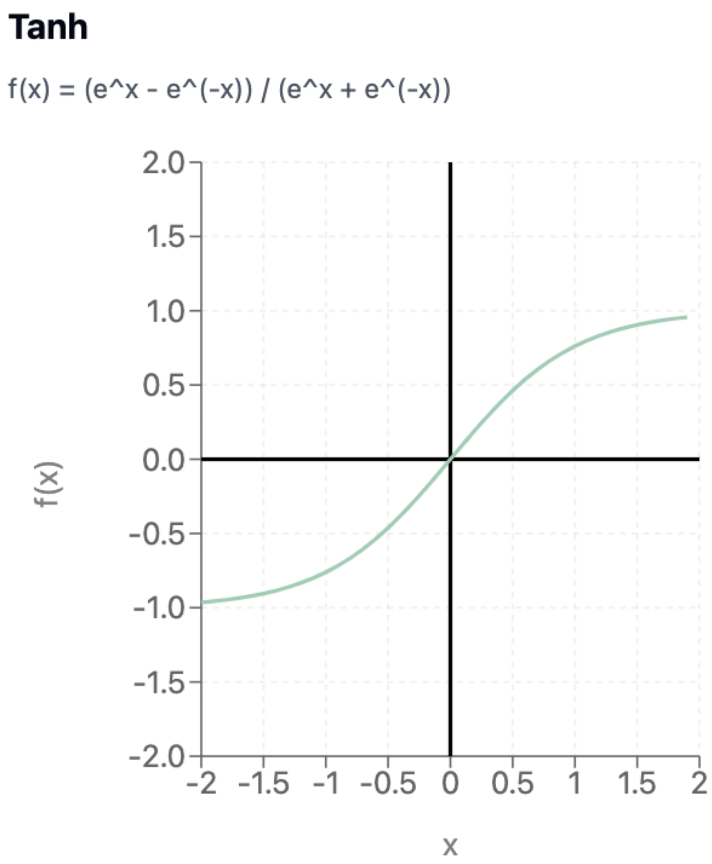
Tanh:

Advantages:

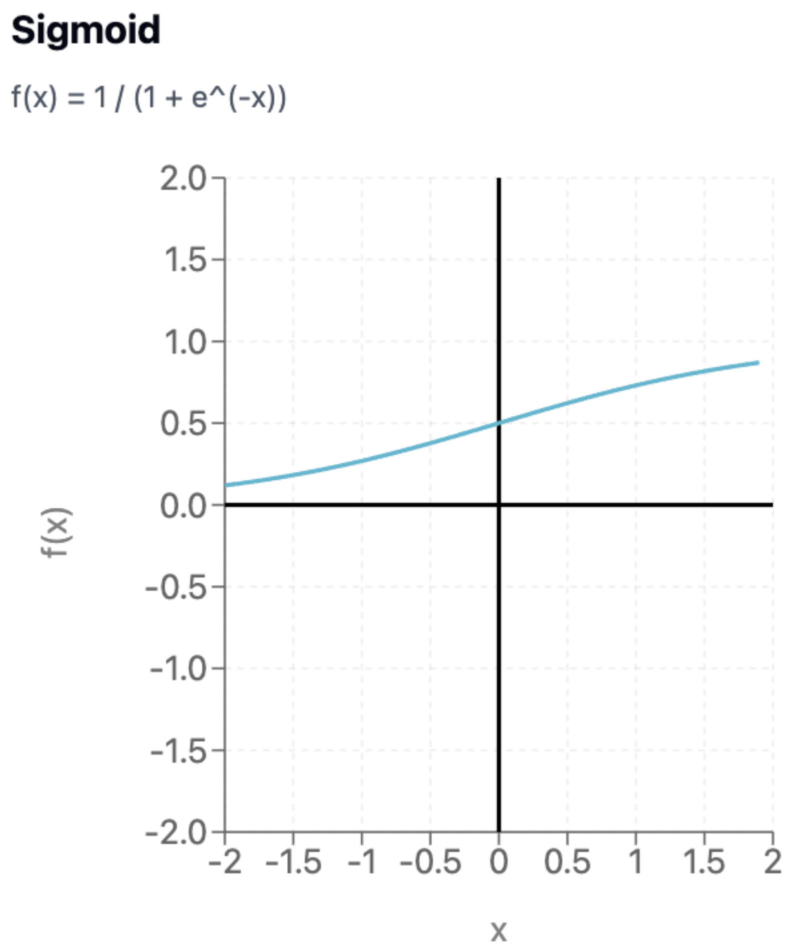
- Always maps output between -1 and 1 (learning is easier when input is normalized and this holds for intermediate layers as well)
- Continuously differentiable

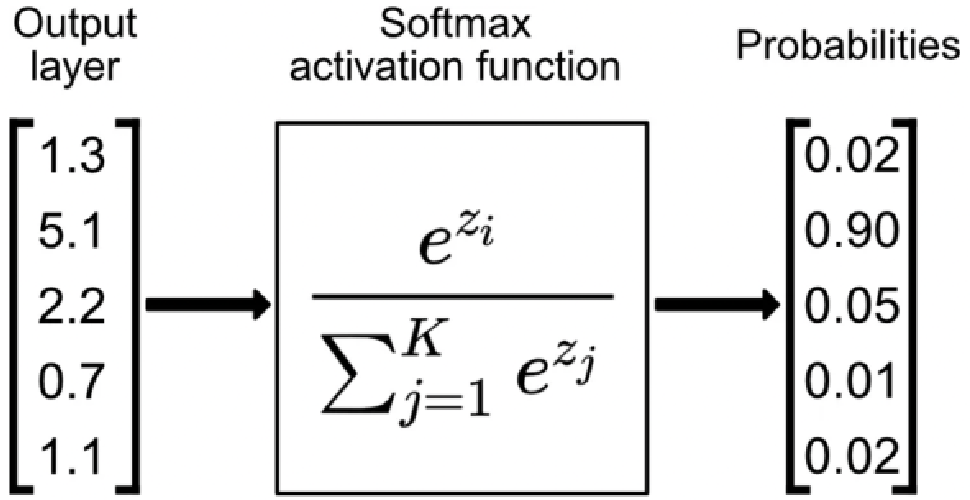
Disadvantages:

- Slower to compute
- Extreme differences in input to activation can get squashed (i.e., z=100 will be very close to z=10000)



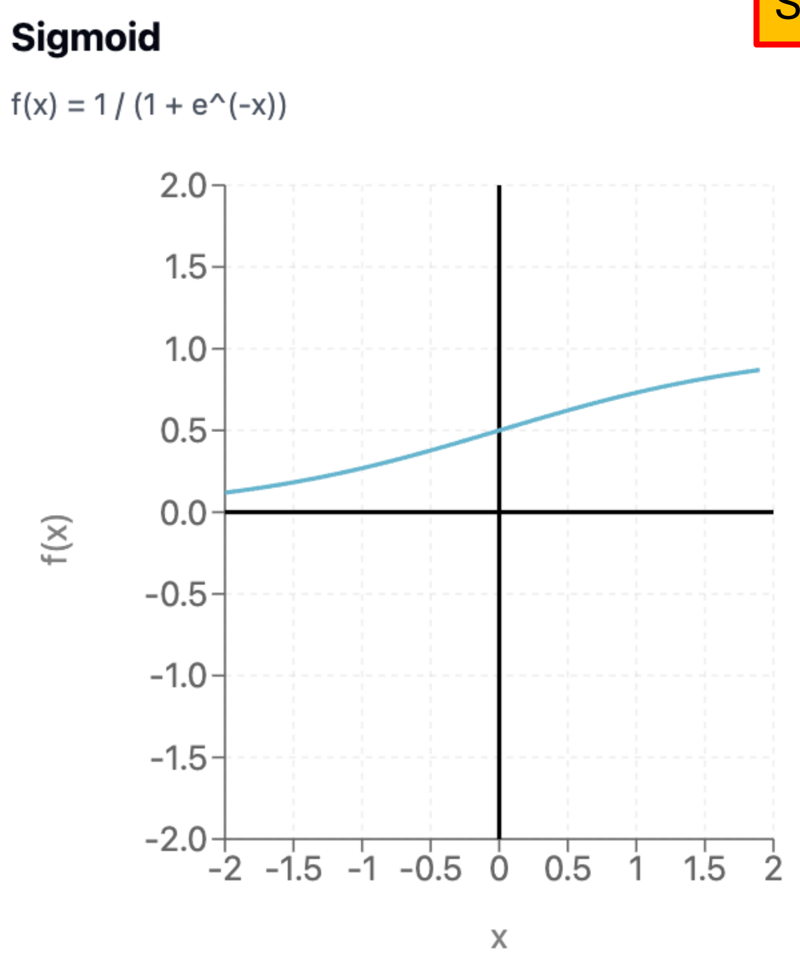
Special Activation Functions for Final Output

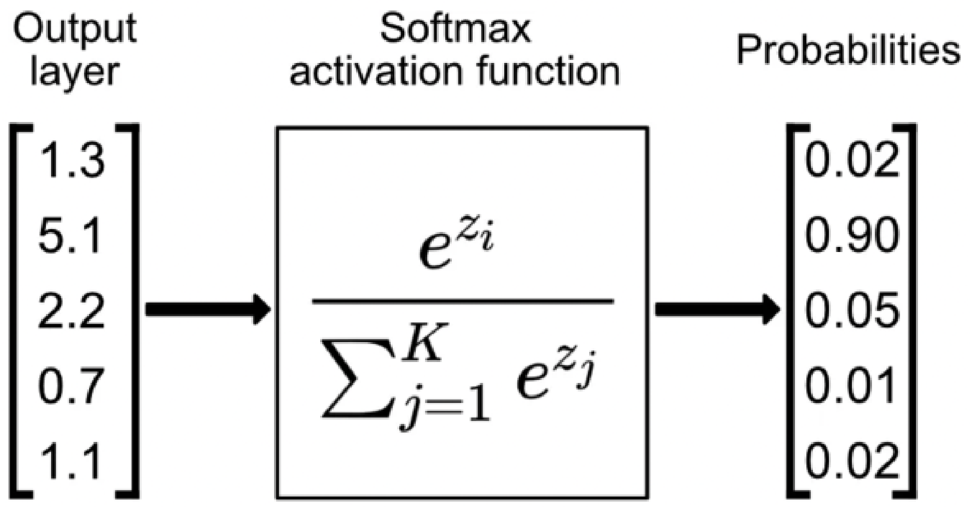




Special Activation Functions for Output

Sigmoid maps input to [0, 1]
Softmax maps vector of inputs to probabilities (outputs sum to 1)

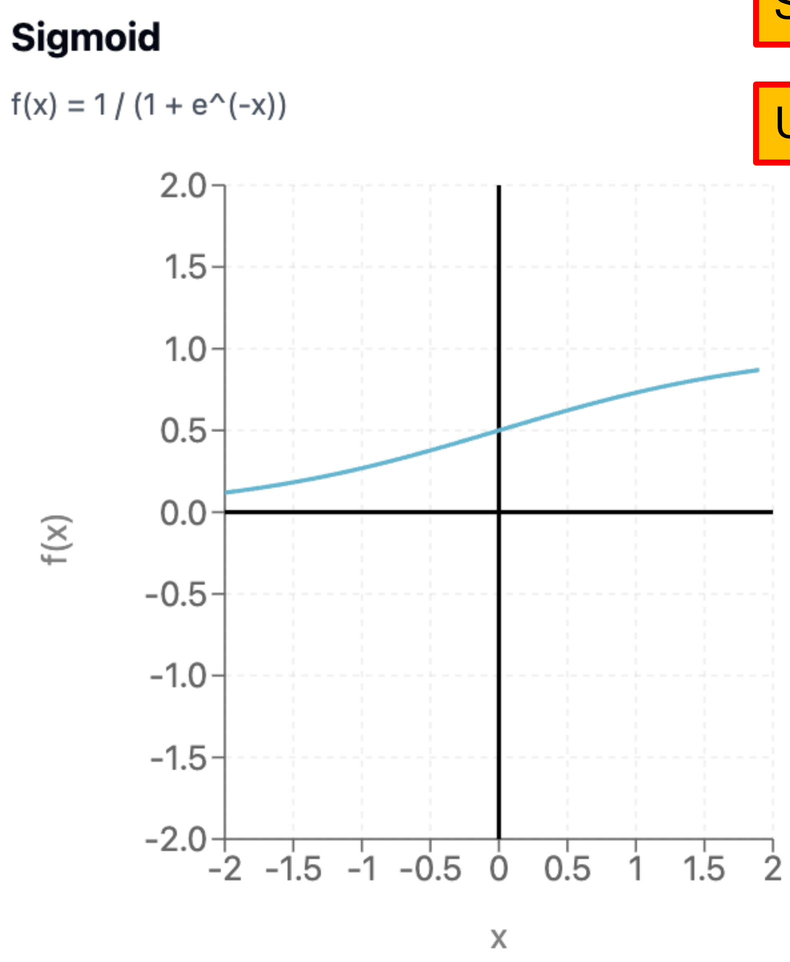


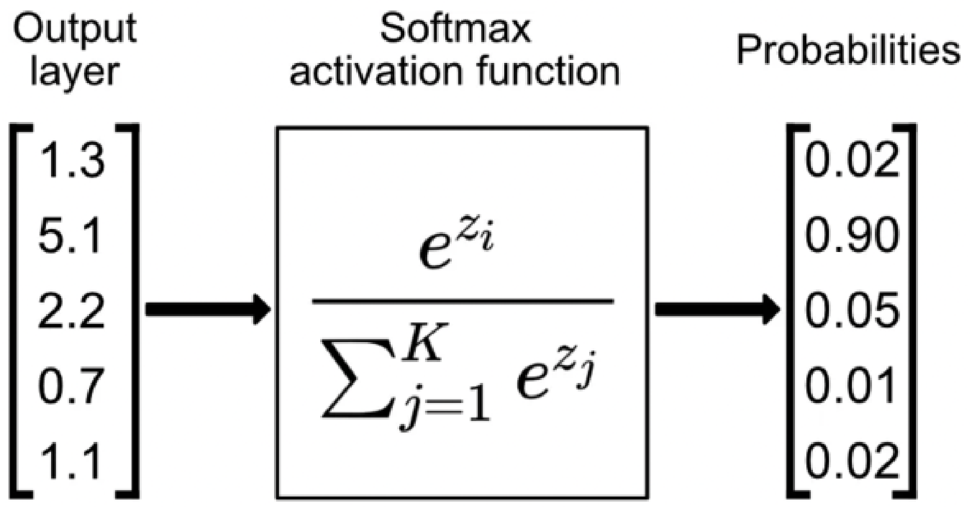


Special Activation Functions for Output

Sigmoid maps input to [0, 1]
Softmax maps vector of inputs to probabilities (outputs sum to 1)

Used for classification tasks

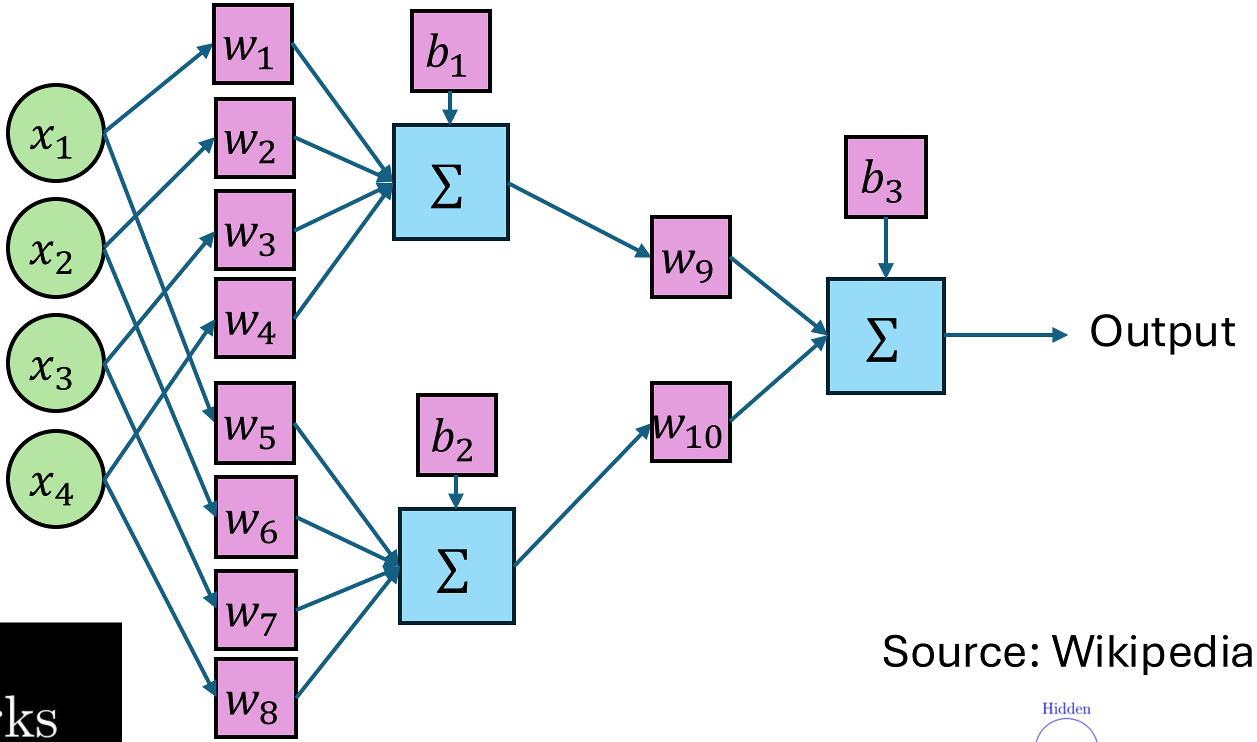




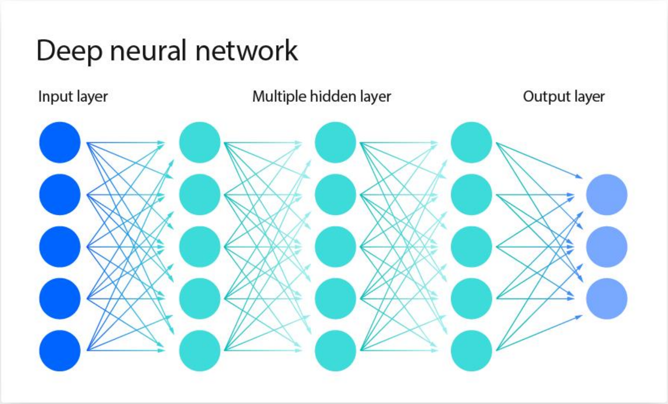
MLPs With Activation Functions

We almost never draw activation functions in our neural network diagrams, but they must always be there!

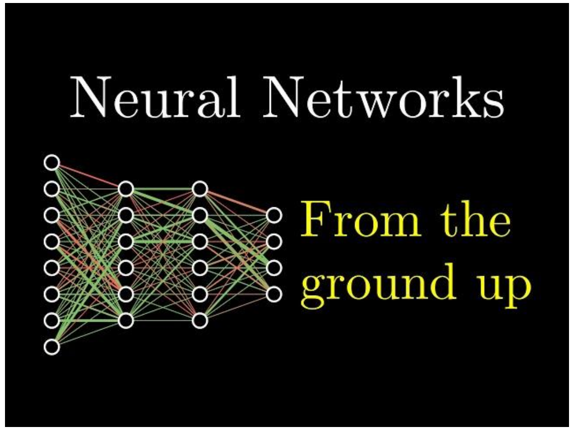
Source: Me



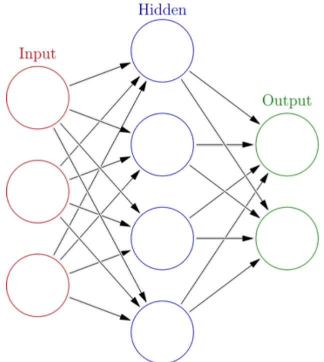
Source: IBM



Source: 3B1B



Source: Wikipedia



Neural Networks

Neural Networks

- Without non-linear activation functions, a neural network is just a Linear Regression.

Neural Networks

- Without non-linear activation functions, a neural network is just a Linear Regression.
- With non-linear activation functions, a neural network is a **universal function approximator**.

Neural Networks

- Without non-linear activation functions, a neural network is just a Linear Regression.
- With non-linear activation functions, a neural network is a **universal function approximator**.
  - For any function, there exists a neural network of fixed depth that can approximate within some ϵ of error.

Neural Networks

- Without non-linear activation functions, a neural network is just a Linear Regression.
- With non-linear activation functions, a neural network is a **universal function approximator**.
  - For any function, there exists a neural network of fixed depth that can approximate within some ϵ of error.
  - If $\epsilon = 0$, i.e., we want a perfect approximation, we may need an infinitely wide network.

Neural Networks

- Without non-linear activation functions, a neural network is just a Linear Regression.
- With non-linear activation functions, a neural network is a **universal function approximator**.
  - For any function, there exists a neural network of fixed depth that can approximate within some ϵ of error.
  - If $\epsilon = 0$, i.e., we want a perfect approximation, we may need an infinitely wide network.
  - This is an **existence** theorem, meaning it tells you that a neural network exists with these properties. It does not tell you how to find the weights of this network.

Intuition

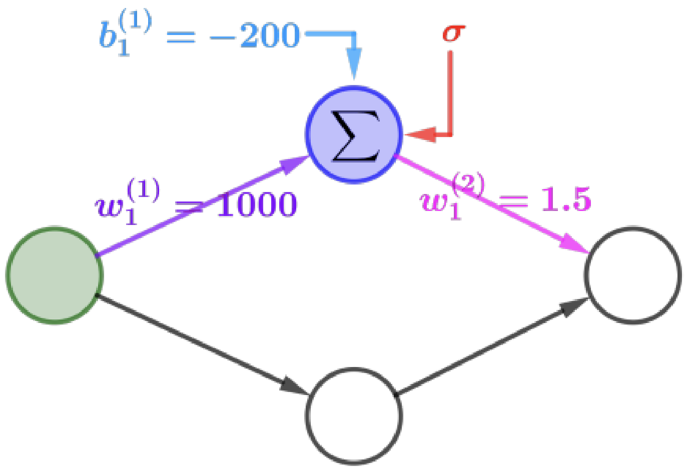


Figure 13: Addition of w_1^2

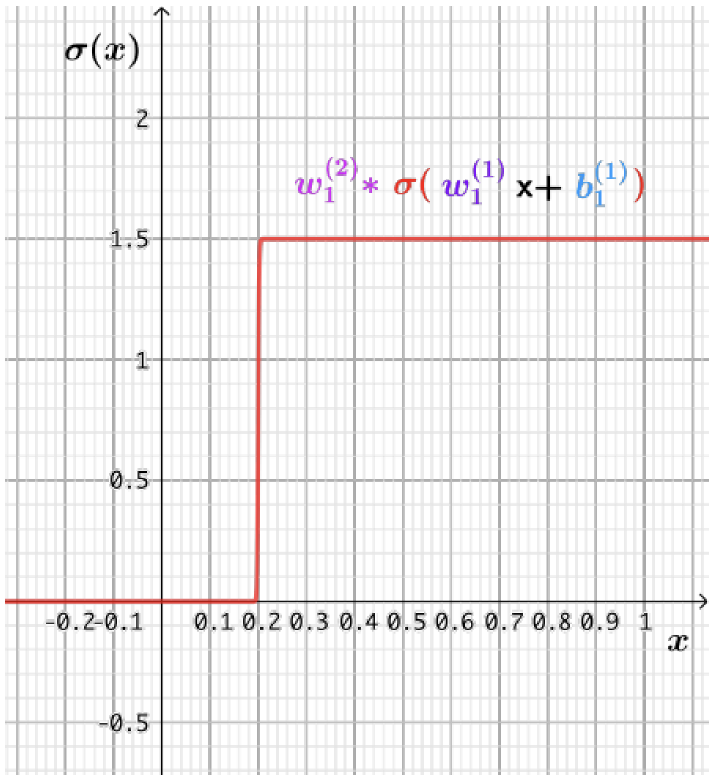


Figure 14: Scaled step-function

With large weights, sigmoid activation functions look like step functions

Approximating functions with step functions

Figure 18: Approximation (N=4)

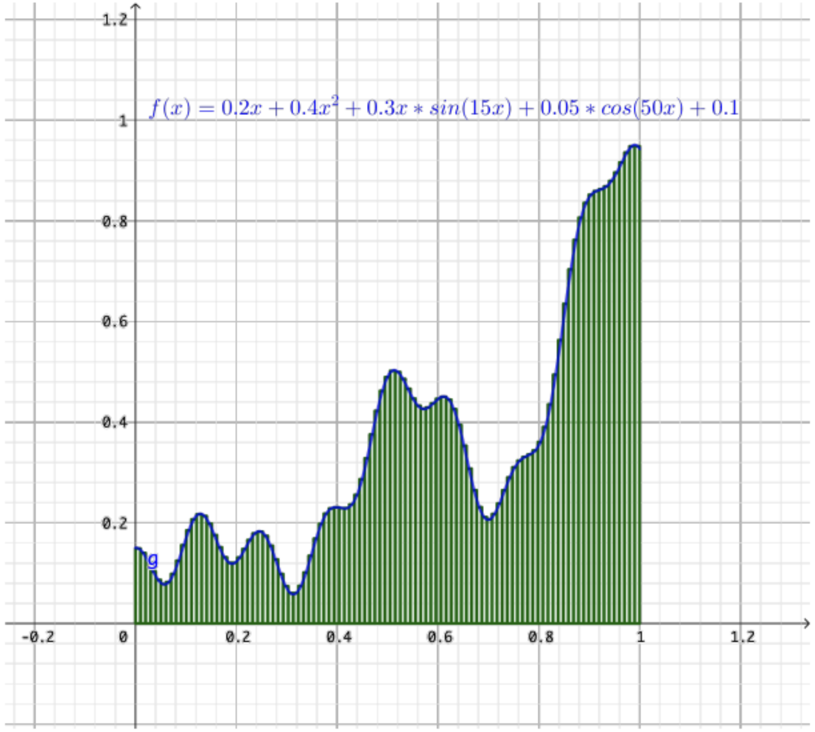


Figure 19: Approximation (N=100)

We can use step functions to approximate arbitrary functions

Neural Networks

- So that’s it, right? That’s why deep learning is so successful. Because neural networks can approximate any function?

Neural Networks

- So that's it, right? That's why deep learning is so successful. Because neural networks can approximate any function?

Neural Networks are not the only Universal Function Approximator

Neural Networks

- So that's it, right? That's why deep learning is so successful. Because neural networks can approximate any function?

Neural Networks are not the only Universal Function Approximator

Decision Trees of infinite depth can fit any function with 100% accuracy

Neural Networks

- So that’s it, right? That’s why deep learning is so successful. Because neural networks can approximate any function?

Neural Networks are not the only Universal Function Approximator

Decision Trees of infinite depth can fit any function with 100% accuracy

Piecewise polynomials are universal function approximators (think Taylor expansions)

Neural Networks

- So that's it, right? That's why deep learning is so successful. Because neural networks can approximate any function?

Neural Networks are not the only Universal Function Approximator

Decision Trees of infinite depth can fit any function with 100% accuracy

Piecewise polynomials are universal function approximators (think Taylor expansions)

Wavelets (i.e., small pieces of sine and cosine) are universal function approximators

Neural Networks

- So that's it, right? That's why deep learning is so successful. Because neural networks can approximate any function?

Neural Networks are not the only Universal Function Approximator

Decision Trees of infinite depth can fit any function with 100% accuracy

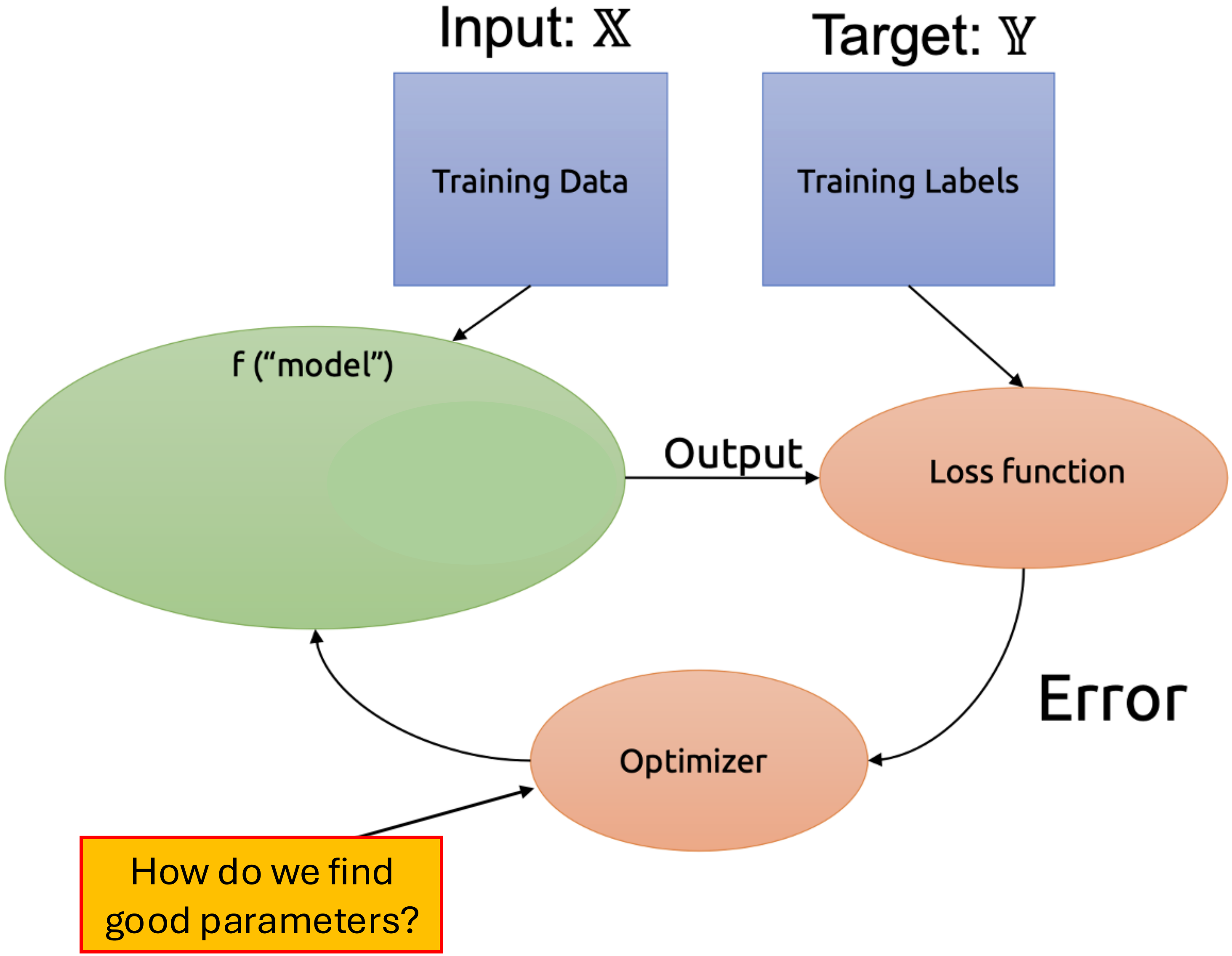
Piecewise polynomials are universal function approximators (think Taylor expansions)

Wavelets (i.e., small pieces of sine and cosine) are universal function approximators

This theorem explains why neural networks are good at fitting the **training** dataset, not why they perform well on the **test** dataset.

Optimization

Learning Network Parameters



Option 1: Closed Form Solution

Goal: Minimize *Loss* function

Process:

- Find derivative (or gradient) of loss function
- Set derivative to 0
- Solve for parameters

Option 1: Closed Form Solution

Goal: Minimize *Loss* function

Process:

- Find derivative (or gradient) of loss function
- Set derivative to 0
- Solve for parameters

Worked for Linear Regressions!

Option 1: Closed Form Solution

Goal: Minimize *Loss* function

Process:

- Find derivative (or gradient) of loss function
- Set derivative to 0
- Solve for parameters

Worked for Linear Regressions!

Only had one point where $\nabla f_{\theta} = \vec{0}$ and that point was a global optimum.

Option 1: Closed Form Solution

Goal: Minimize *Loss* function

Process:

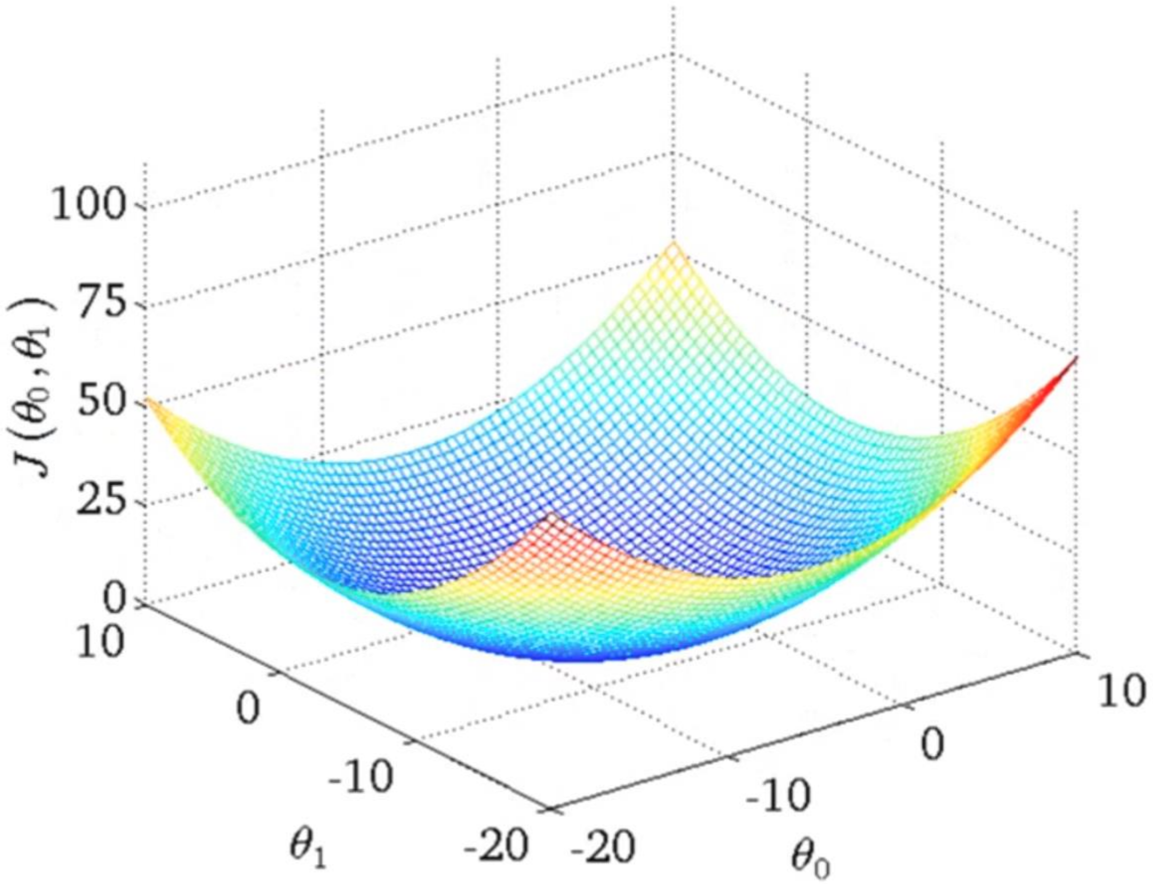
- Find derivative (or gradient) of loss function
- Set derivative to 0
- Solve for parameters

Worked for Linear Regressions!

Only had one point where $\nabla f_\theta = \vec{0}$ and that point was a global optimum.

MSE is *convex* with respect to the parameters of the linear Regression

Convexity



Formally:

- For any two points x_1, x_2 and $\lambda \in [0, 1]$
- $\lambda f(x_1) + (1-\lambda) f(x_2) \leq \lambda x_1 + (1-\lambda) x_2$

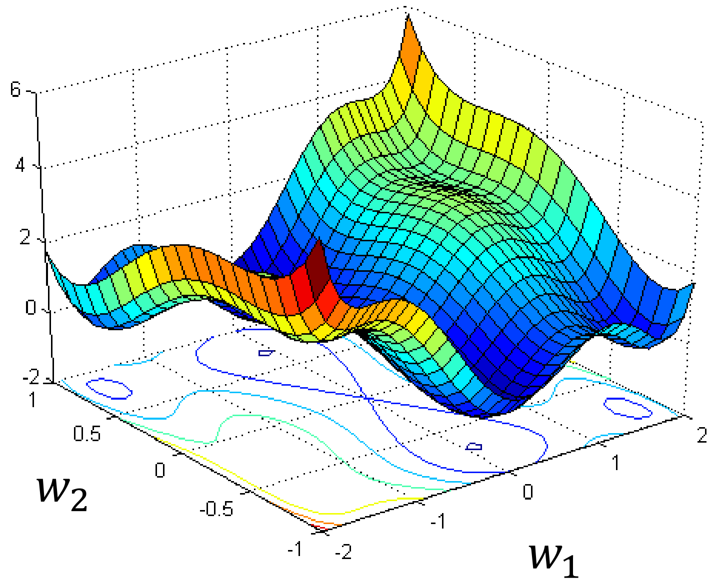
The line connecting any two points on the graph will always be above the function.

For convex functions, finding a point with $\nabla f = 0$ is **sufficient** for knowing the point is a **global** minimum

Picture Source: Andrew Ng

Non-Convex Functions

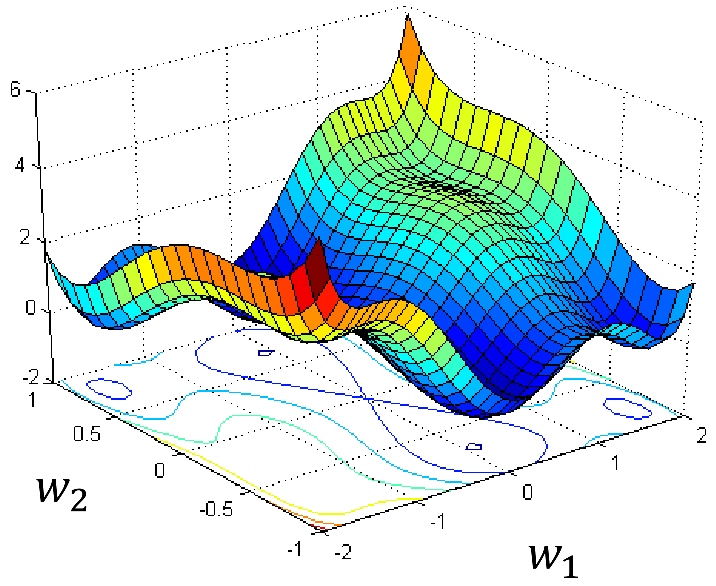
MSE is **not** convex with respect to network parameters when non-linear activations are involved.



Non-Convex Functions

MSE is **not** convex with respect to network parameters when non-linear activations are involved.

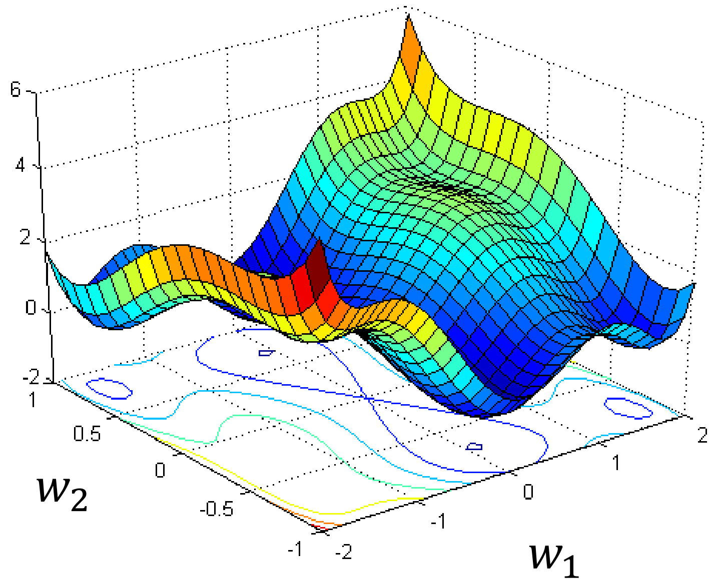
Multiple local minima



Non-Convex Functions

MSE is **not** convex with respect to network parameters when non-linear activations are involved.

Multiple local minima

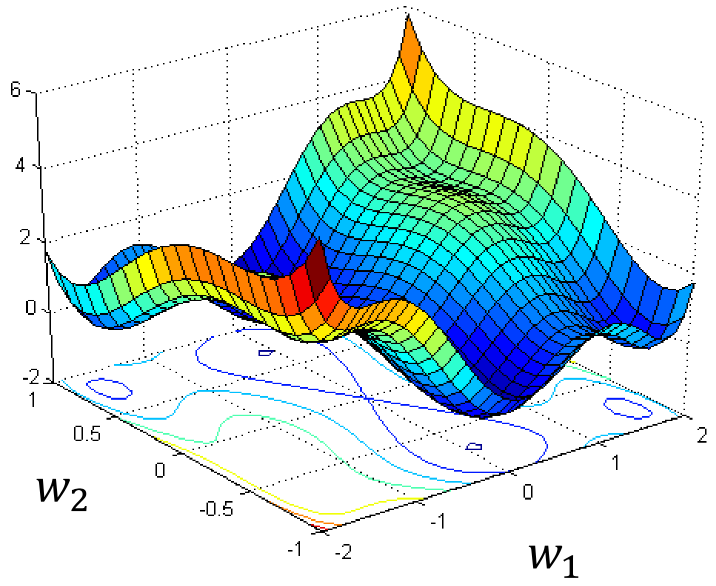


Local maxima

Non-Convex Functions

MSE is **not** convex with respect to network parameters when non-linear activations are involved.

Multiple local minima



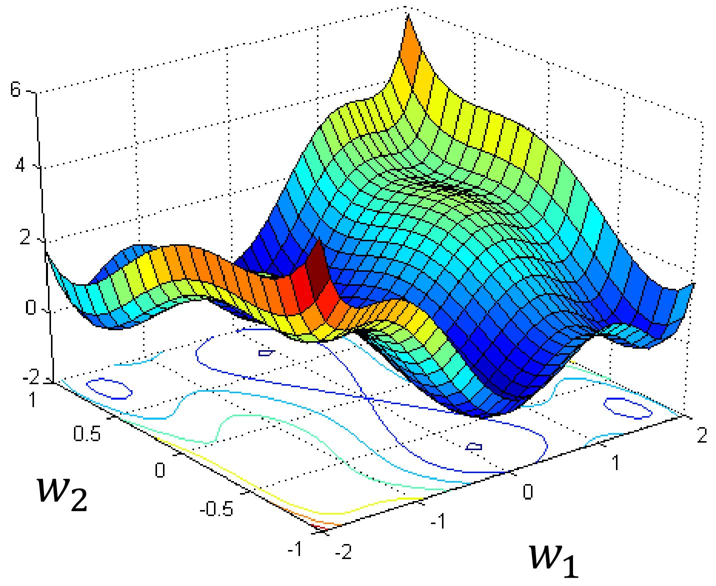
Saddle points

Local maxima

Non-Convex Functions

MSE is **not** convex with respect to network parameters when non-linear activations are involved.

Multiple local minima



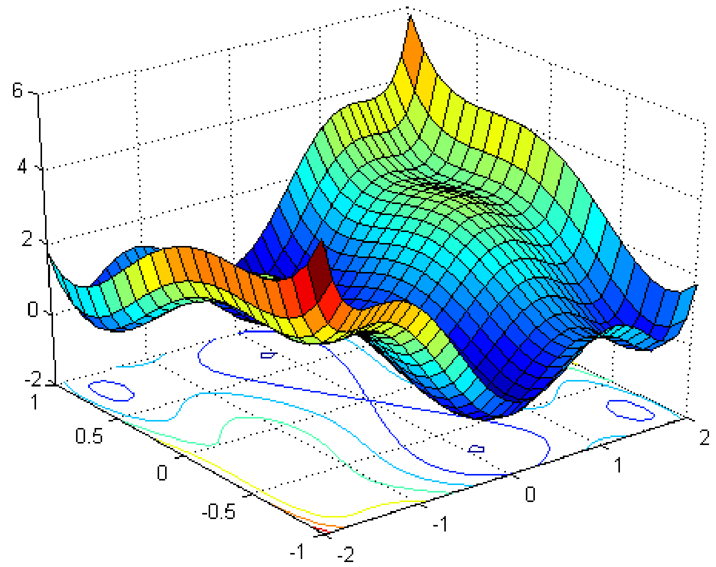
Saddle points

If ReLU or other piecewise activation function is used, may need 2^n piecewise functions to write out ∇f_θ...

Local maxima

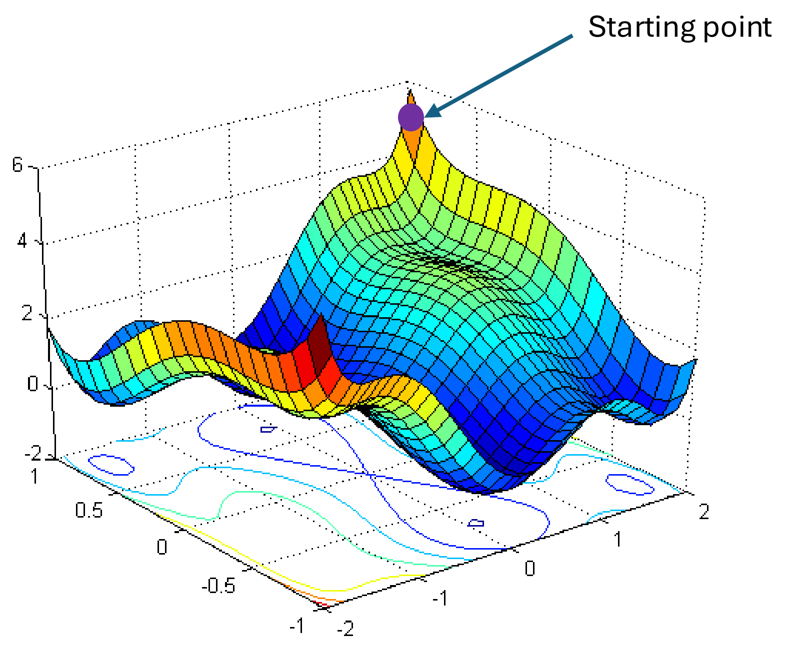
Option 2: Gradient Descent

1. Start with some initial set of parameters
2. Take small step in the direction of the negative gradient
3. Repeat 2 until convergence



Option 2: Gradient Descent

1. Start with some initial set of parameters
2. Take small step in the direction of the negative gradient
3. Repeat 2 until convergence

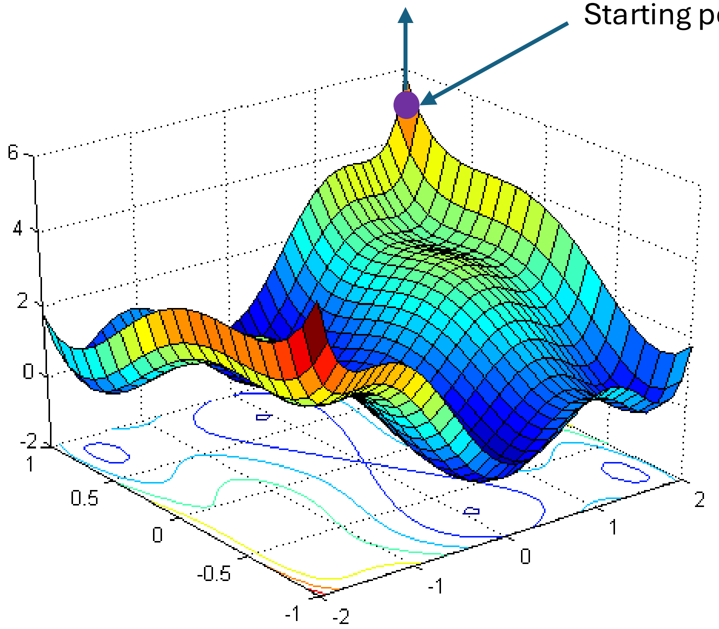


Option 2: Gradient Descent

1. Start with some initial set of parameters
2. Take small step in the direction of the negative gradient
3. Repeat 2 until convergence

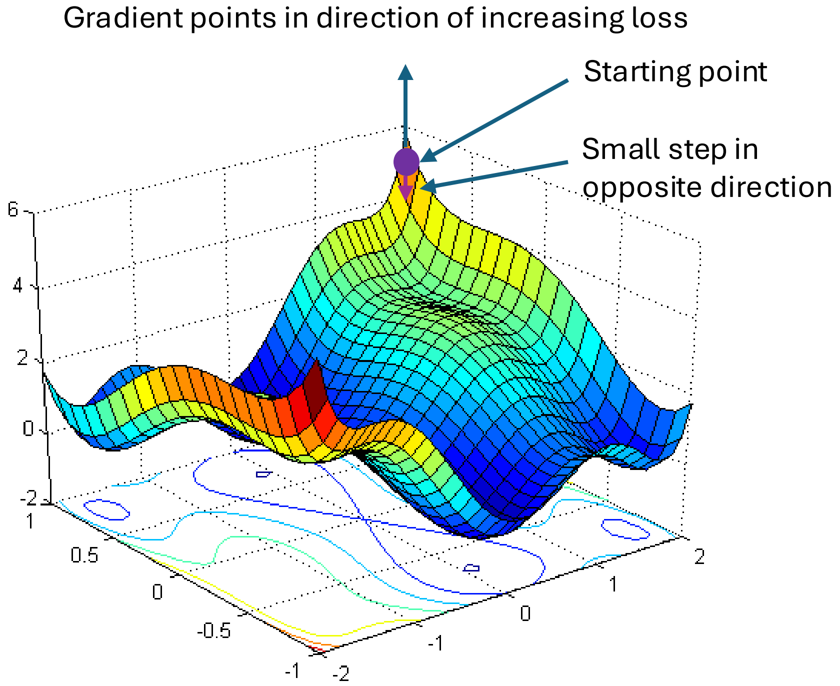
Gradient points in direction of increasing loss

Starting point



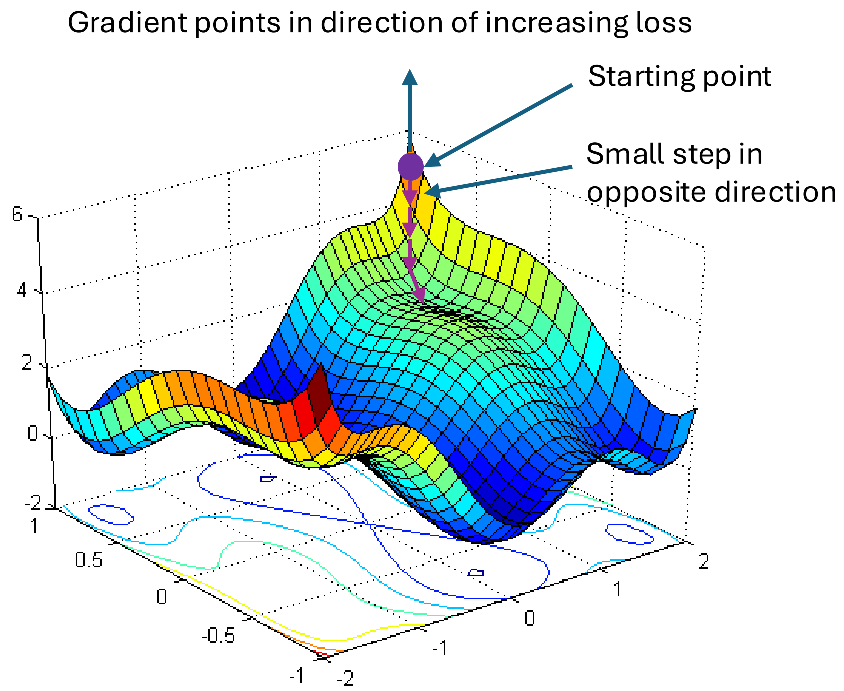
Option 2: Gradient Descent

1. Start with some initial set of parameters
2. Take small step in the direction of the negative gradient
3. Repeat 2 until convergence



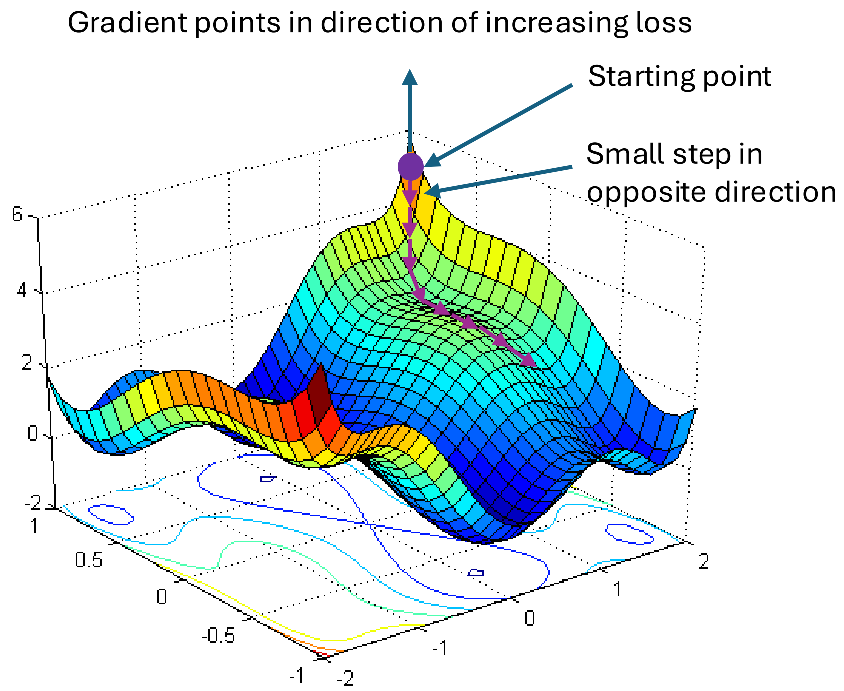
Option 2: Gradient Descent

1. Start with some initial set of parameters
2. Take small step in the direction of the negative gradient
3. Repeat 2 until convergence



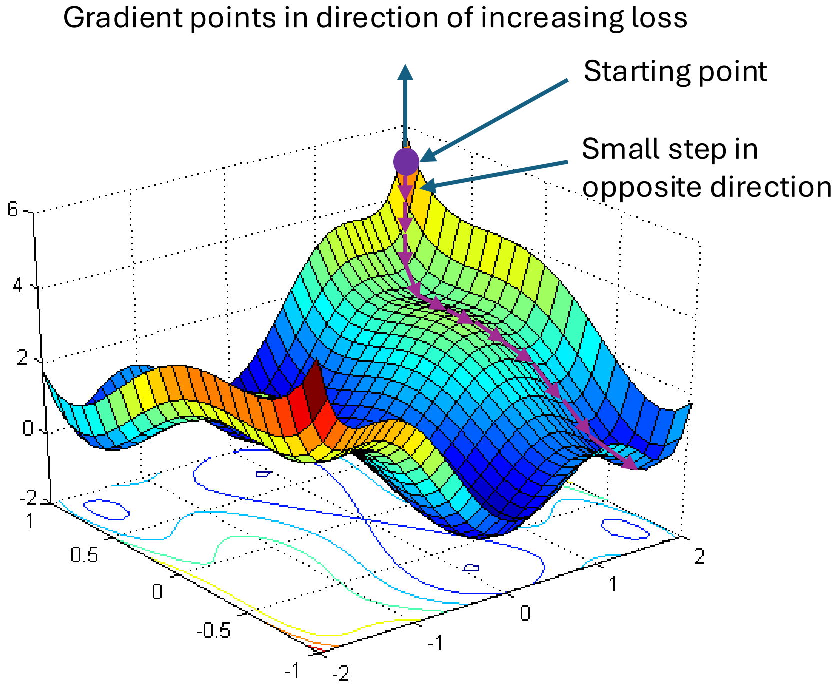
Option 2: Gradient Descent

1. Start with some initial set of parameters
2. Take small step in the direction of the negative gradient
3. Repeat 2 until convergence



Option 2: Gradient Descent

1. Start with some initial set of parameters
2. Take small step in the direction of the negative gradient
3. Repeat 2 until convergence



Vector Calculus

Vector Calculus

- Partial Derivative: the derivative of a multi-variable function with respect to one of its inputs

Vector Calculus

- Partial Derivative: the derivative of a multi-variable function with respect to one of its inputs
- Example: $f(x, w, b) = wx + b$

Vector Calculus

- Partial Derivative: the derivative of a multi-variable function with respect to one of its inputs
- Example: $f(x, w, b) = wx + b$
- The partial derivative with respect to w is $\frac{\partial f}{\partial w}$

Vector Calculus

- Partial Derivative: the derivative of a multi-variable function with respect to one of its inputs
- Example: $f(x, w, b) = wx + b$
- The partial derivative with respect to w is $\frac{\partial f}{\partial w}$
- How to compute: Treat all other variables as constants and differentiate with respect to that variable

$$\frac{\partial f}{\partial w} = \frac{\partial}{\partial w}(wx + b) = \frac{\partial}{\partial w}(wx) + \frac{\partial}{\partial w}(b) = x$$

Gradients

Gradient: the vector of partial derivatives

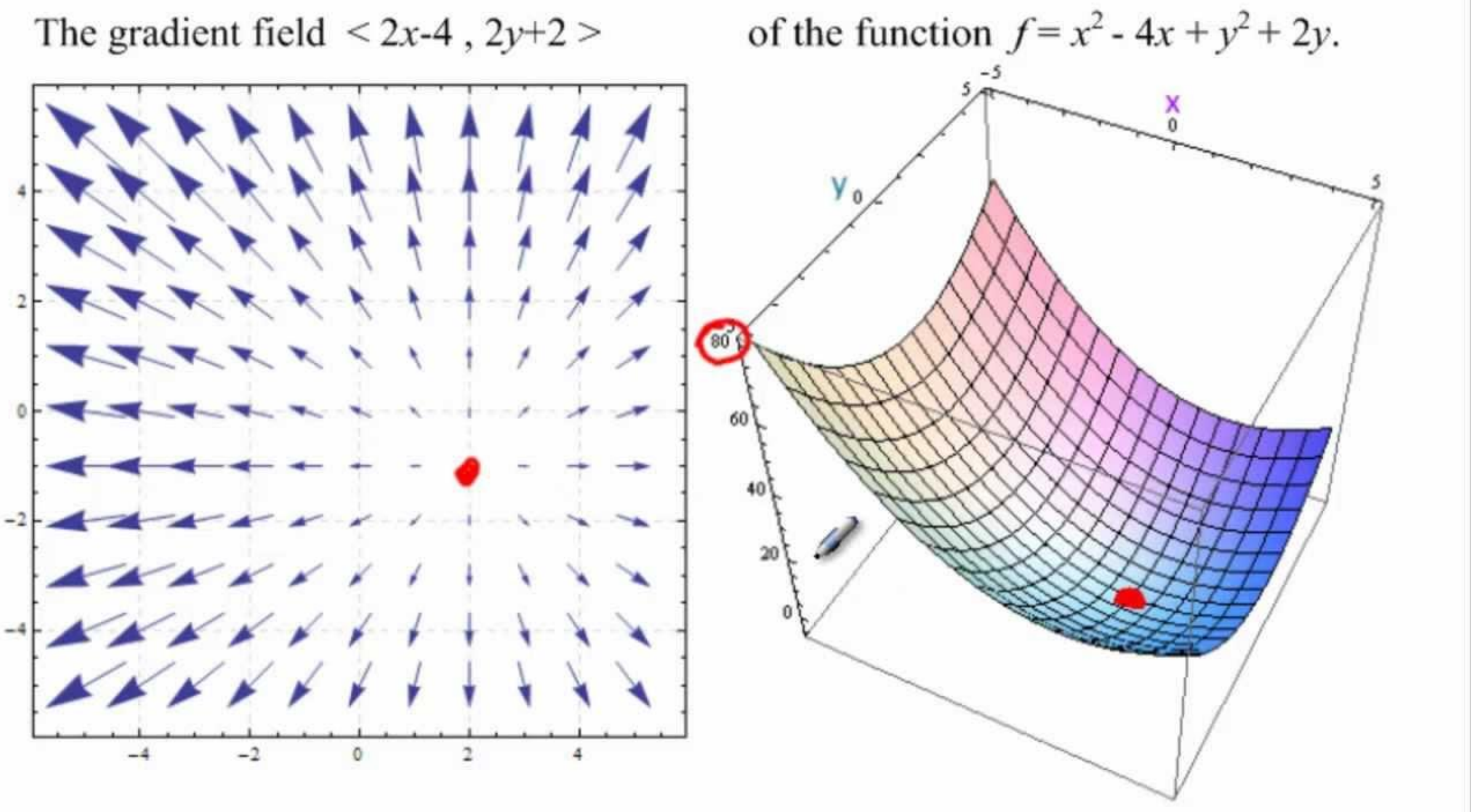
Vector "points" in direction of increasing f values.

$$\nabla f = [\frac{\partial f}{\partial w}, \frac{\partial f}{\partial b}, \dots]$$

$$f(x, w, b) = wx + b$$

$$\nabla f_\theta = [\frac{\partial f}{\partial w}, \frac{\partial f}{\partial b}, \frac{\partial f}{\partial x}]$$

Gradients

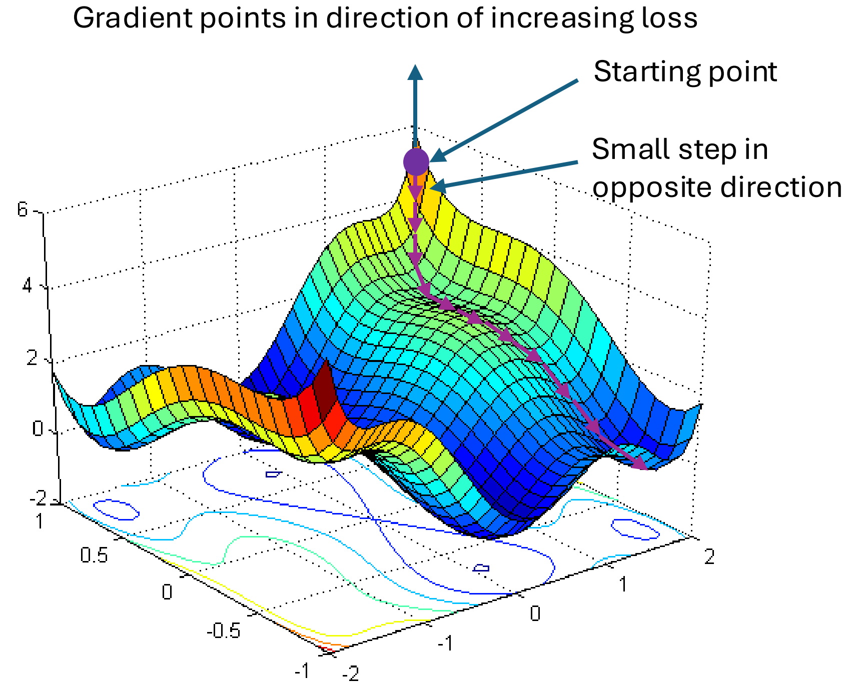

The gradient field < 2x-4 , 2y+2 > of the function f = x² - 4x + y² + 2y.
x
y
80
60
40
20
0
-4
-2
0
2
4
5
-5

Option 2: Gradient Descent

1. Start with some initial set of parameters
2. Take small step in the direction of the negative gradient
3. Repeat 2 until convergence

For N iterations or until $\Delta\theta < \epsilon$:

$$\vec{\theta} \leftarrow \theta - \alpha \nabla f_\theta$$



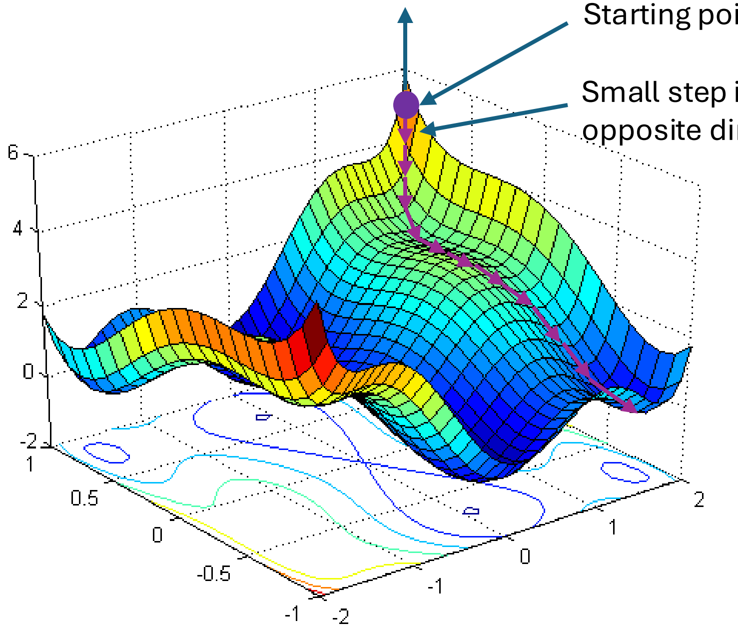
Option 2: Gradient Descent

1. Start with some initial set of parameters
2. Take small step in the direction of the negative gradient
3. Repeat 2 until convergence

For N iterations or until $\Delta\theta < \epsilon$:

$$\vec{\theta} \leftarrow \theta - \alpha \nabla f_{\theta}$$

Gradient of what?



Option 2: Gradient Descent

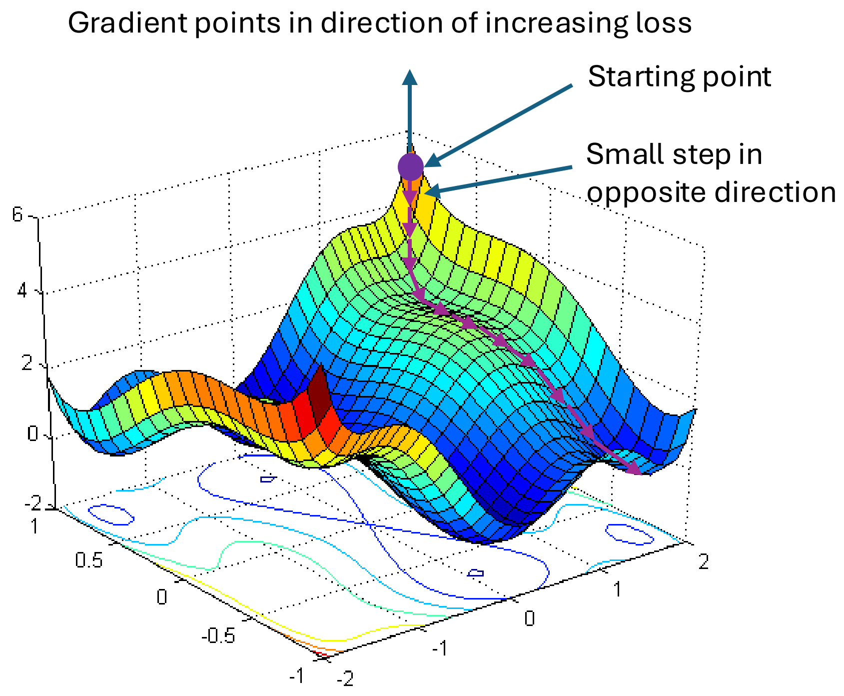
1. Start with some initial set of parameters
2. Take small step in the direction of the negative gradient
3. Repeat 2 until convergence

For N iterations or until $\Delta\theta < \epsilon$:

$$\vec{\theta} \leftarrow \theta - \alpha \nabla f_{\theta}$$

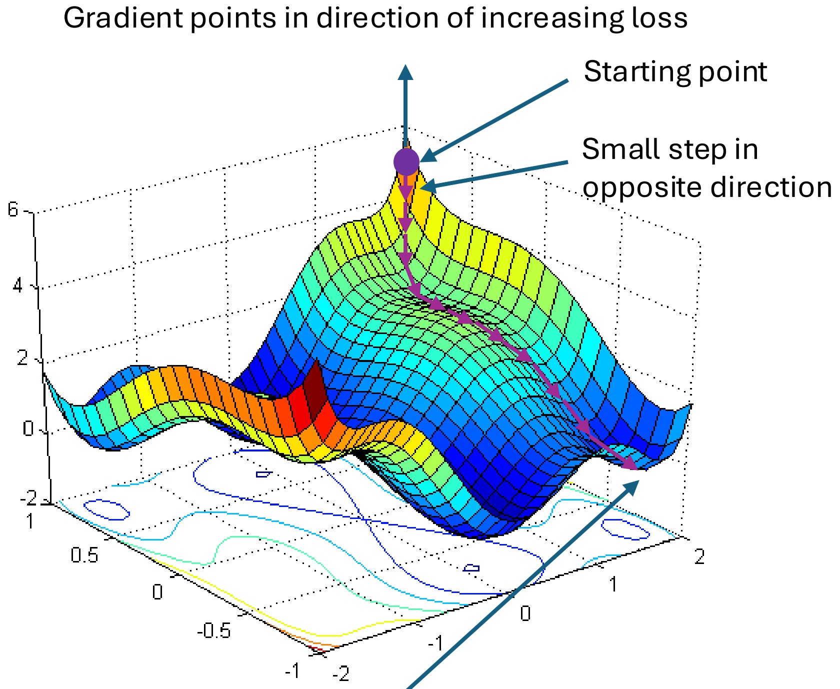
Gradient of what?

Why is this negative?



Option 2: Gradient Descent

1. Start with some initial set of parameters
2. Take small step in the direction of the negative gradient
3. Repeat 2 until convergence



For N iterations or until $\Delta\theta < \epsilon$:

$$\vec{\theta} \leftarrow \theta - \alpha \nabla f_{\theta}$$

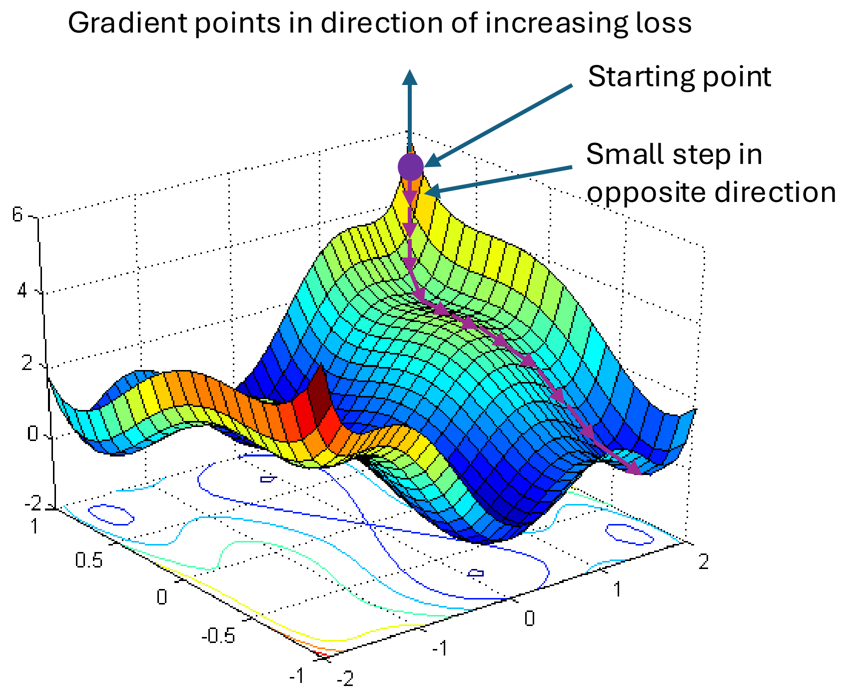
Gradient of what?

Why is this negative?

Wait, this isn't even the best θ

Option 2: Gradient Descent

1. Start with some initial set of parameters
2. Take small step in the direction of the negative gradient
3. Repeat 2 until convergence



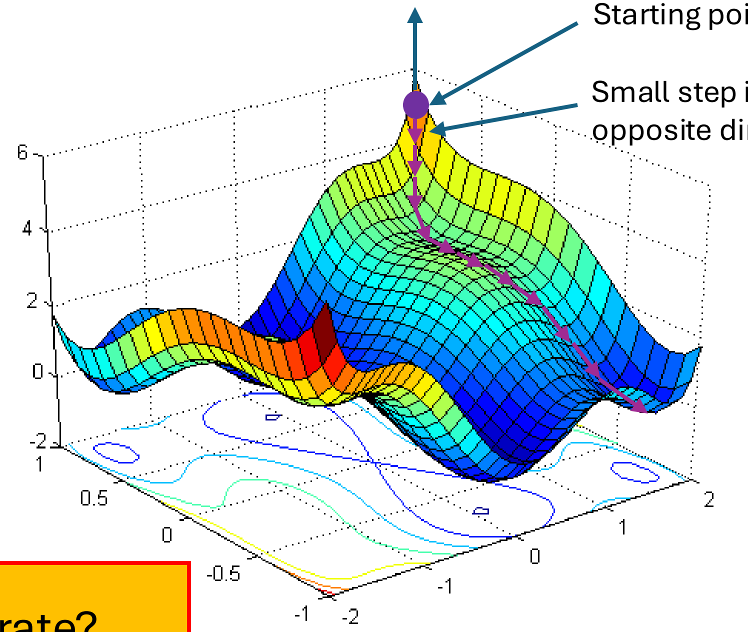
For N iterations or until $\Delta\theta < \epsilon$:

$$\vec{\theta} \leftarrow \theta - \alpha \nabla f_{\theta}$$

Learning Rate $\alpha \in [0,1]$

Option 2: Gradient Descent

1. Start with some initial set of parameters
2. Take small step in the direction of the negative gradient
3. Repeat 2 until convergence



For N iterations or until $\Delta\theta < \epsilon$:

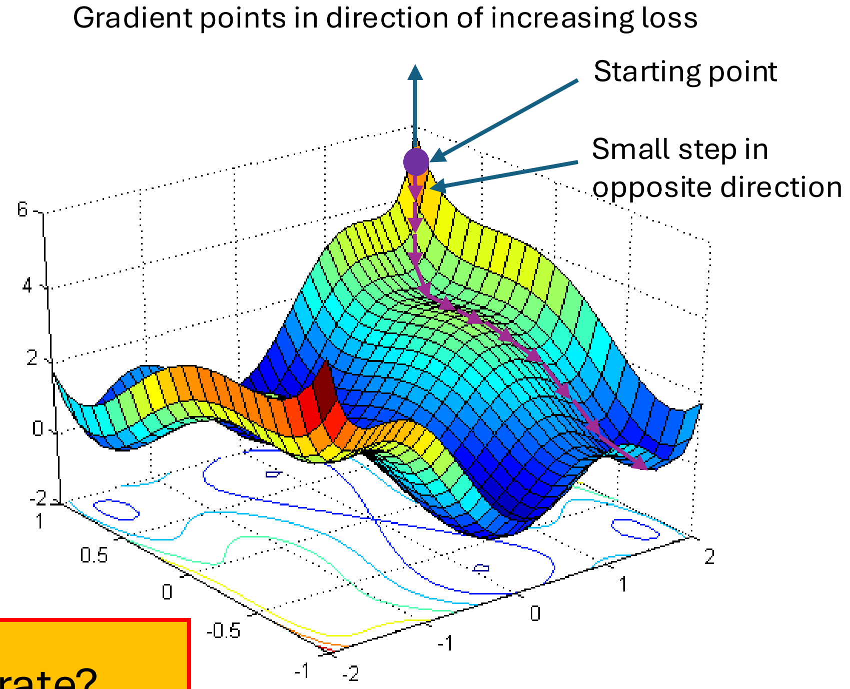
$$\vec{\theta} \leftarrow \theta - \alpha \nabla f_{\theta}$$

Learning Rate $\alpha \in [0,1]$

Why do we need a learning rate?

Option 2: Gradient Descent

1. Start with some initial set of parameters
2. Take small step in the direction of the negative gradient
3. Repeat 2 until convergence



For N iterations or until $\Delta\theta < \epsilon$:

$$\vec{\theta} \leftarrow \theta - \alpha \nabla f_{\theta}$$

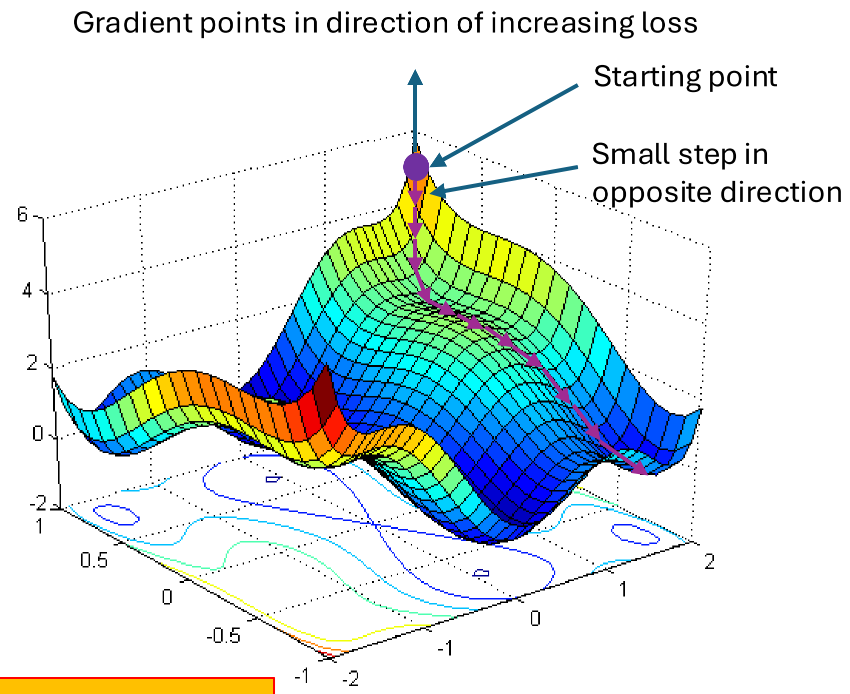
Learning Rate $\alpha \in [0,1]$

Why do we need a learning rate?

Derivatives/Gradients only hold locally

Option 2: Gradient Descent

1. Start with some initial set of parameters
2. Take small step in the direction of the negative gradient
3. Repeat 2 until convergence



For N iterations or until $\Delta\theta < \epsilon$:

$$\vec{\theta} \leftarrow \theta - \alpha \nabla f_\theta$$

Gradient Descent does not converge to the global minimum.
It can (and pretty much always does) get stuck in local minima.

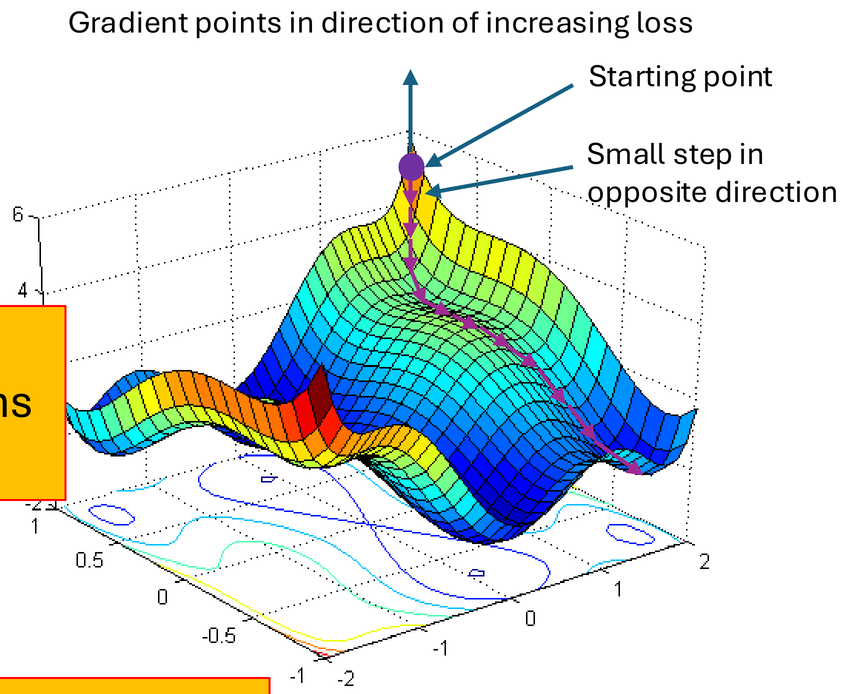
Option 2: Gradient Descent

1. Start with some initial set of parameters
2. Take small step in the direction of the negative gradient
3. Repeat 2 until convergence

Understanding gradient descent is the single most important concept in all of Deep Learning. Most decisions in DL are made for reasons related to gradients.

For N iterations or until $\Delta\theta < \epsilon$:

$$\vec{\theta} \leftarrow \theta - \alpha \nabla f_{\theta}$$



Gradient Descent does not converge to the global minimum. It can (and pretty much always does) get stuck in local minima.

Review: Mean Squared Error

Used previously for linear regression:

Model with parameters θ

$$MSE = \frac{\sum_i^n (y_i - f_\theta(\vec{x}))\hat{}2}{n}$$

Used for regression tasks (prediction of continuous variable)

Gradients

Gradients

Gradient descent needs gradients, how do we actually calculate them?

Gradients

Gradient descent needs gradients, how do we actually calculate them?

$$L = \frac{\sum_i^n (y_i - f_\theta(\vec{x}))\hat{}2}{n}$$

Gradients

Gradient descent needs gradients, how do we actually calculate them?

$$L = \frac{\sum_i^n (y_i - f_\theta(\vec{x}))\hat{}2}{n}$$

$$L = \frac{\sum_i^n [y_i^2 - 2f_\theta(\vec{x}) + f_\theta(\vec{x})^2]}{n}$$

Gradients

Gradient descent needs gradients, how do we actually calculate them?

$$L = \frac{\sum_i^n (y_i - f_\theta(\vec{x}))\verb|^|2}{n}$$

$$L = \frac{\sum_i^n [y_i^2 - 2f_\theta(\vec{x}) + f_\theta(\vec{x})^2]}{n}$$

$$\nabla_\theta L = \frac{\sum_i^n 2 \cdot \nabla f_\theta(\vec{x}) + 2 \cdot \nabla f_\theta(\vec{x}) \cdot f_\theta(\vec{x})]}{n}$$

Gradients

Gradient descent needs gradients, how do we actually calculate them?

$$L = \frac{\sum_i^n (y_i - f_\theta(\vec{x}))\hat{}2}{n}$$

$$L = \frac{\sum_i^n [y_i^2 - 2 f_\theta(\vec{x}) + f_\theta(\vec{x})^2]}{n}$$

$$\nabla_\theta L = \frac{\sum_i^n 2 \cdot \nabla f_\theta(\vec{x}) + 2 \cdot \nabla f_\theta(\vec{x}) \cdot f_\theta(\vec{x})]}{n}$$

But what is this?

Gradients

Gradient descent needs gradients, how do we actually calculate them?

$$L = \frac{\sum_i^n (y_i - f_\theta(\vec{x}))^\wedge 2}{n}$$

$$L = \frac{\sum_i^n [y_i^2 - 2f_\theta(\vec{x}) + f_\theta(\vec{x})^2]}{n}$$

$$\nabla_\theta L = \frac{\sum_i^n 2 \cdot \nabla f_\theta(\vec{x}) + 2 \cdot \nabla f_\theta(\vec{x}) \cdot f_\theta(\vec{x})]}{n}$$

But what is this?

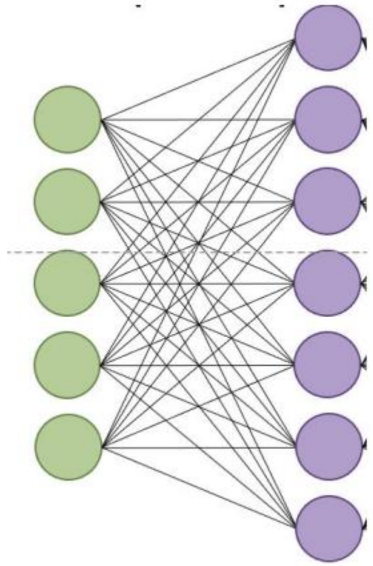
$f_\theta = wx + b$
For a single output

Weight Matrix for a Layer of Neurons

- We have an input of size n and we want an output vector of size m.
- We will represent our weights as a matrix.
  - What should the dimensions of our matrix be?

n inputs

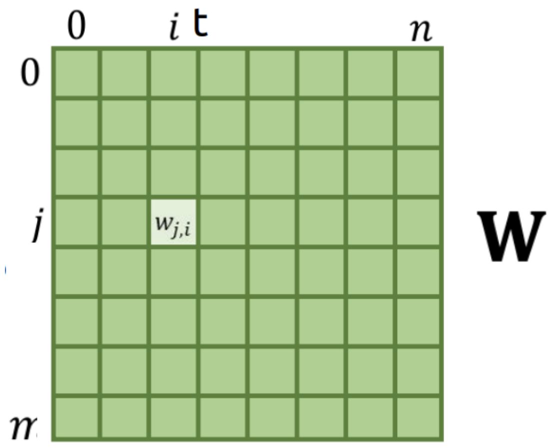
m inputs



Weight Matrix

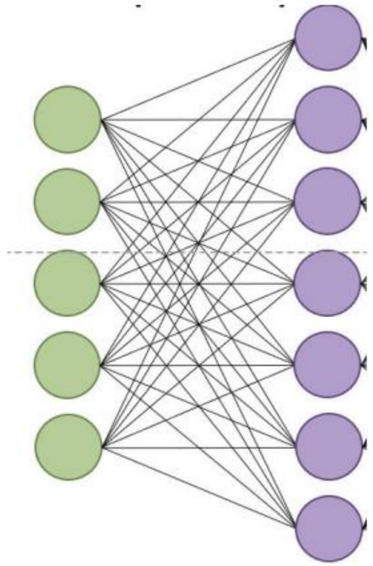
- We have an input of size *n* and we want an output vector of size *m*.
- We will represent our weights as a matrix.
  - What should the dimensions of our matrix be?

$w_{j,i}$ is the j^{th} row and the i^{th} column of our matrix, or the weight multiplied by the i^{th} index of the input which is used to create the j^{th} index in the output



n inputs

m inputs

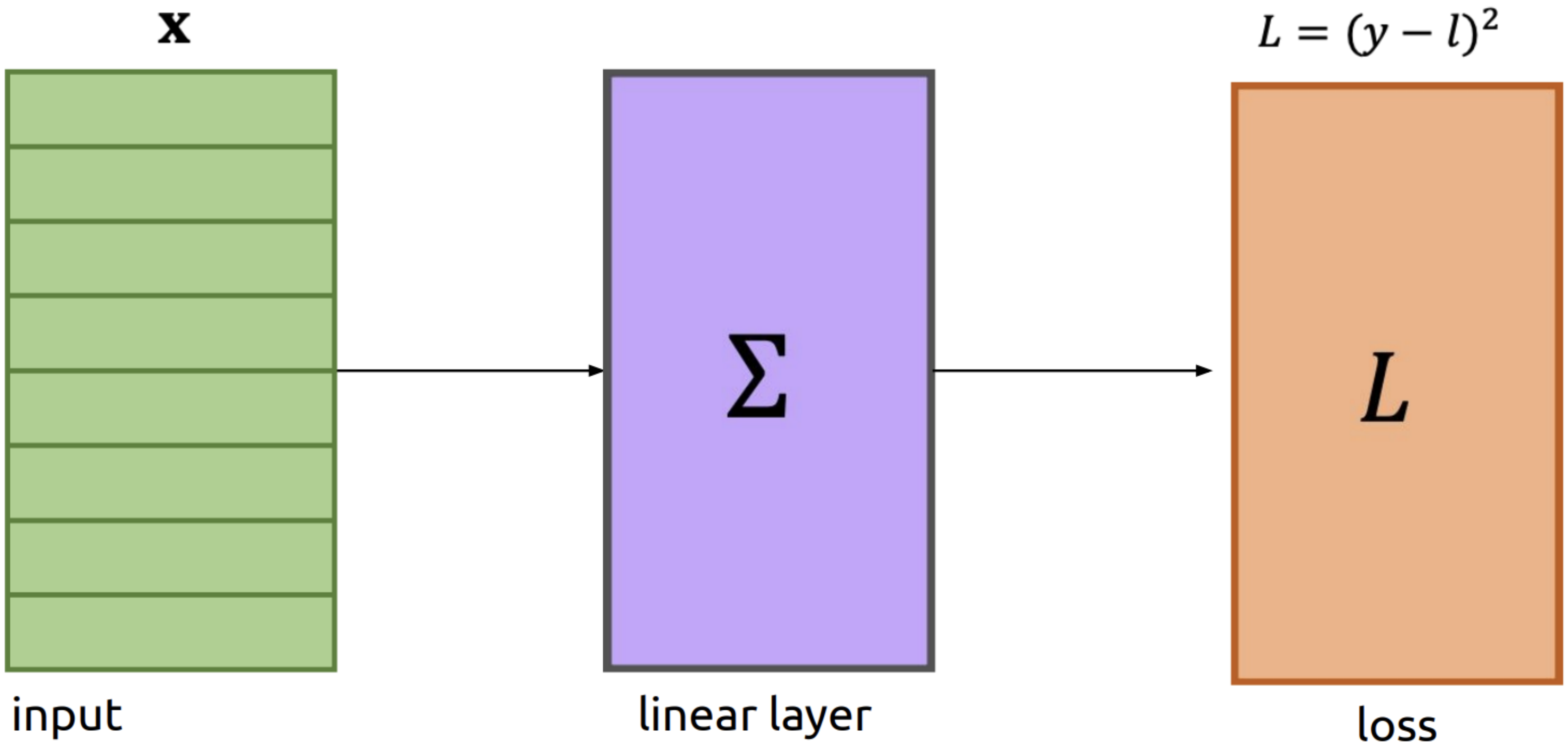


j=1

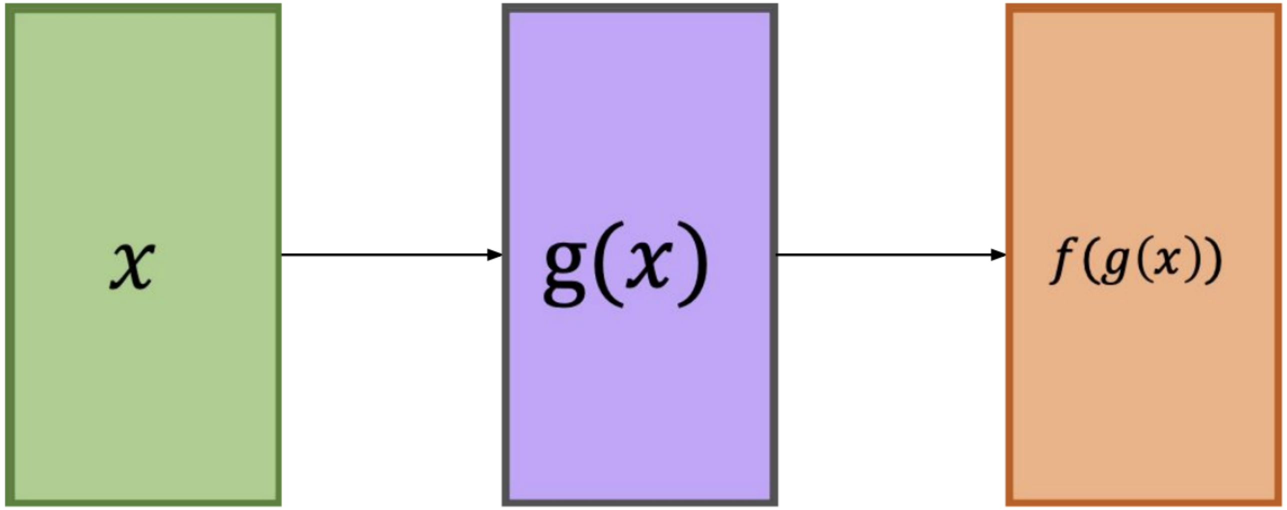
$$l_j = \sum_k w_{j,k}\, x_k + b_j$$

$$l = \boldsymbol{w}.\boldsymbol{x} + \boldsymbol{b}$$

$$L = (y - l)^2$$



Looking at composite function!



Chain rule

If f and g are both differentiable and $F(x)$ is the composite function defined by $F(x) = f(g(x))$ then F is differentiable and F' is given by the product

$$F'(x) = f'(g(x))\, g'(x)$$

Differentiate outer function

Differentiate inner function

Courtesy: https://www.onlinemathlearning.com/chain-rule.html

Applying Chain rule [Example]

$$f(x) = x^2 \qquad \mathrm{g}(x) = (2x^2 + 1)$$

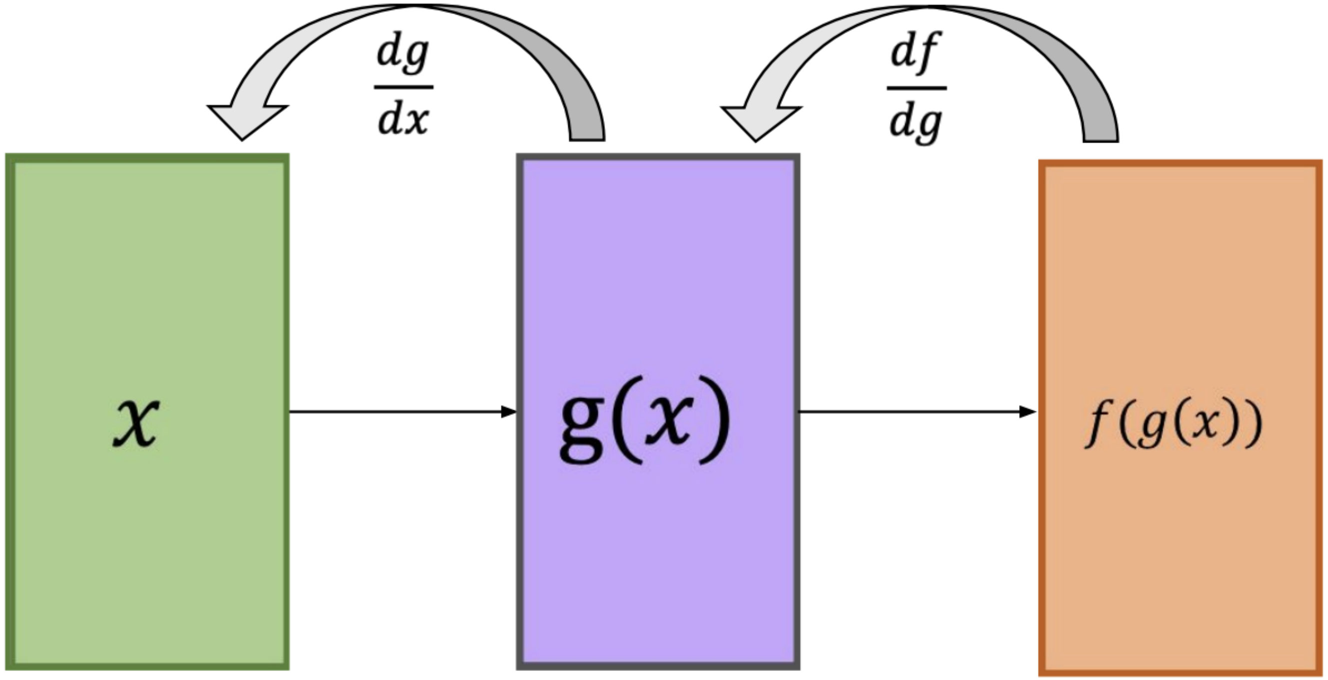
$$F(x) = f(g(x))$$

$$F(x) = (2x^2 + 1)^2$$

The Chain Rule (for Differentiation)

- Given arbitrary function: $f\big(g(x)\big) \Rightarrow \frac{df}{dx} = \frac{df}{dg} \cdot \frac{dg}{dx}$

Each layer computes the gradients with respect to it's variables and passes the result backwards



Backpropagation

(or backward pass)

Classification

Classification

In general, we'd like to optimize the accuracy of our model (#correct/#total)

Classification

In general, we’d like to optimize the accuracy of our model (#correct/#total)

Need Loss function to be small for best model, not large.

Classification

In general, we'd like to optimize the accuracy of our model (#correct/#total)

Need Loss function to be small for best model, not large.

Proposed Loss Function: $L = 1 - \frac{\#\ Correct}{n}$

Classification

In general, we'd like to optimize the accuracy of our model (#correct/#total)

Need Loss function to be small for best model, not large.

Proposed Loss Function: $L = 1 - \frac{\#\,Correct}{n}$

The Issue: most of the time, the gradient of this loss function is $\nabla \mathrm{L}_{\theta} = 0$

Classification

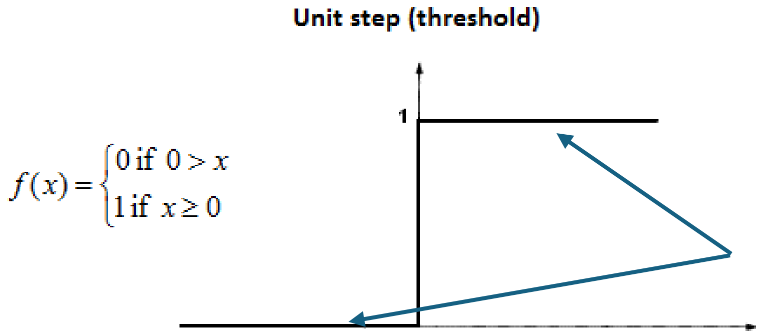
In general, we'd like to optimize the accuracy of our model (#correct/#total)

Need Loss function to be small for best model, not large.

Proposed Loss Function: $L = 1 - \frac{\# \, Correct}{n}$

The Issue: most of the time, the gradient of this loss function is $\nabla \mathrm{L}_{\theta} = 0$

Gradient is only non-zero when changing a θ has an impact on output predictions



0 gradient everywhere except x=0

X=0 is not differentiable, but it does have a sub-gradient

Classification

In general, we'd like to optimize the accuracy of our model (#correct/#total)

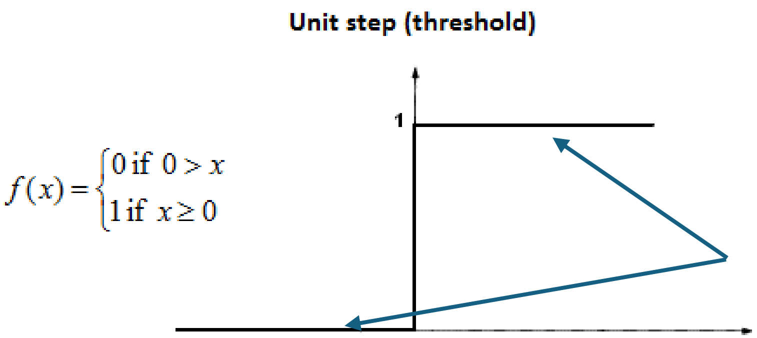
Need Loss function to be small for best model, not large.

Proposed Loss Function: $L = 1 - \frac{\#\ Correct}{n}$

The Issue: most of the time, the gradient of this loss function is $\nabla L_\theta = 0$

Gradient is only non-zero when changing a θ has an impact on output predictions

We cannot use classification as a loss function because it is **incompatible with gradient descent**



0 gradient everywhere except x=0

X=0 is not differentiable, but it does have a sub-gradient

Remaining Questions for next week:

1) What loss function can we use for classification?
2) How do we actually calculate the gradient of a network?
    1) If the loss function is applied to the whole dataset, shouldn't we be concerned about the size of the dataset?
    2) Gradient descent is an iterative approach. If each iteration is slow, the whole algorithm will take too long to finish.
3) Gradient descent can get stuck in local minima.

Can we do better?