Deep Learning

CSCI 1470

Day 5: MLPs and Optimization

Eric Ewing

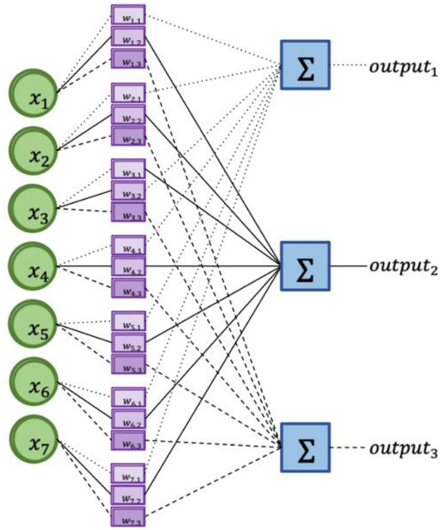
Friday, 1/31/25

Review

Perceptrons used for binary classification.

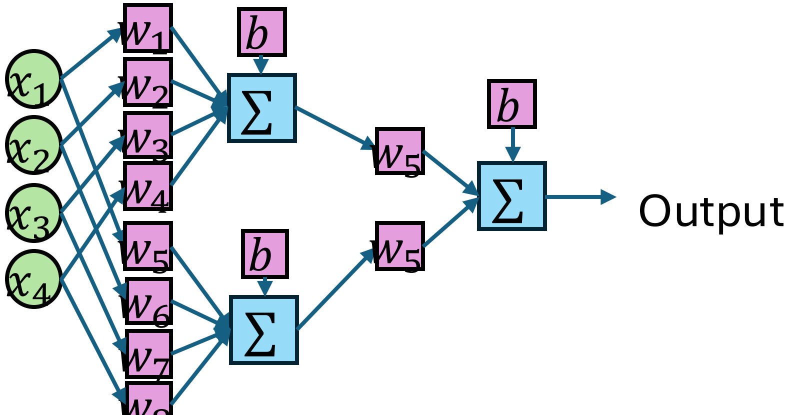
Want to perform multi-class classification?

Multiple Perceptrons sharing inputs



Want to learn more complex functions?

Multiple Layers



Today's Goals

Introduction to the theory of neural networks

(1) Universal Approximation Theory

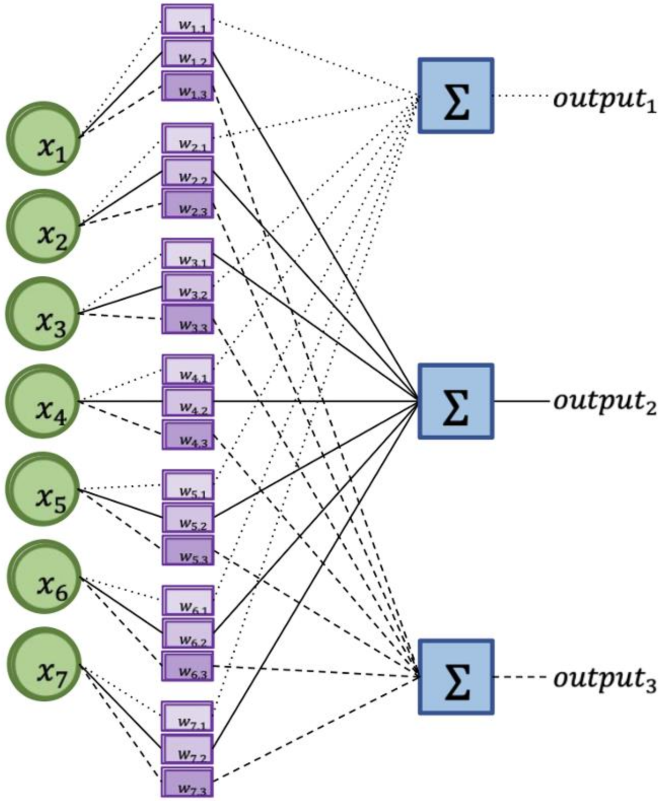
(2) How do we train Neural Networks?

Multi-Class Classification

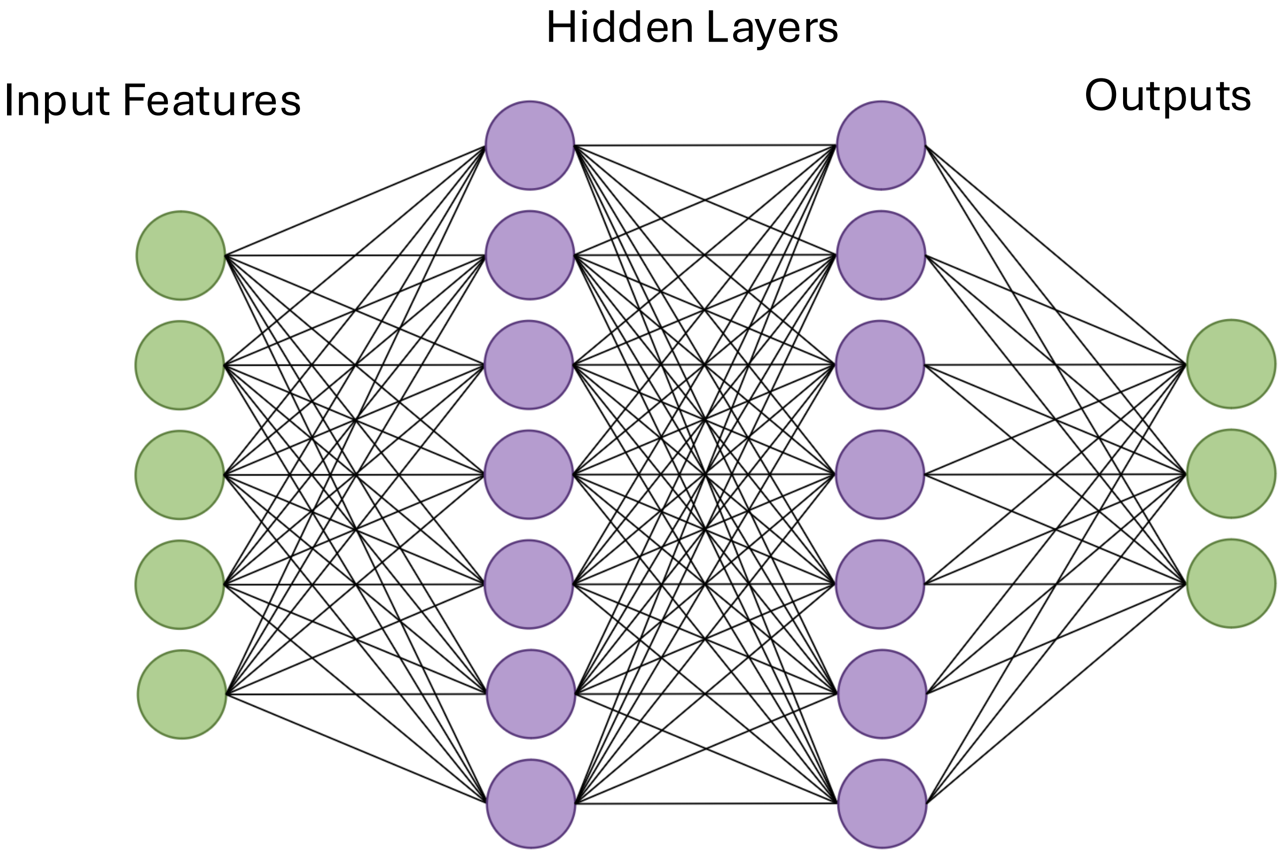
Multi-class classification with "perceptrons"

Need to remove threshold function from outputs

Why?



MLPs

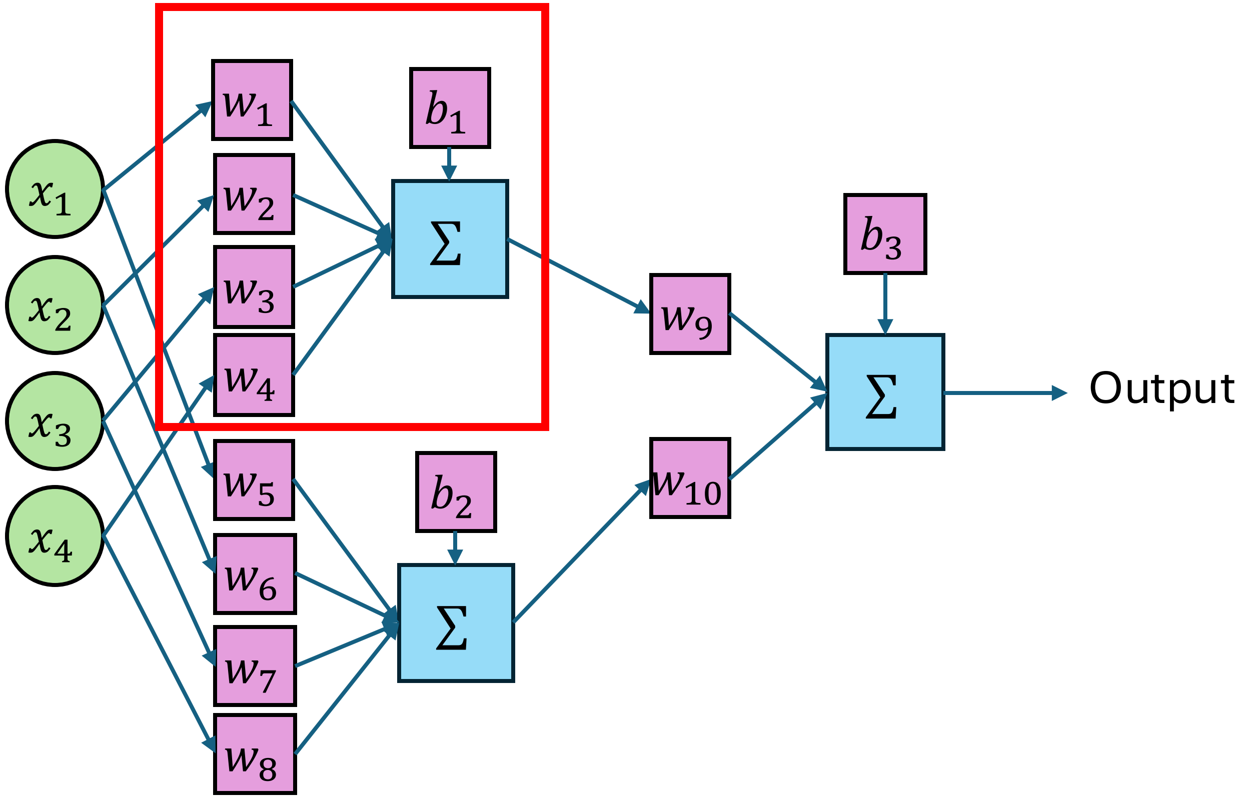


A Multi-Layered Neural Net

Multi-Layer Perceptrons

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

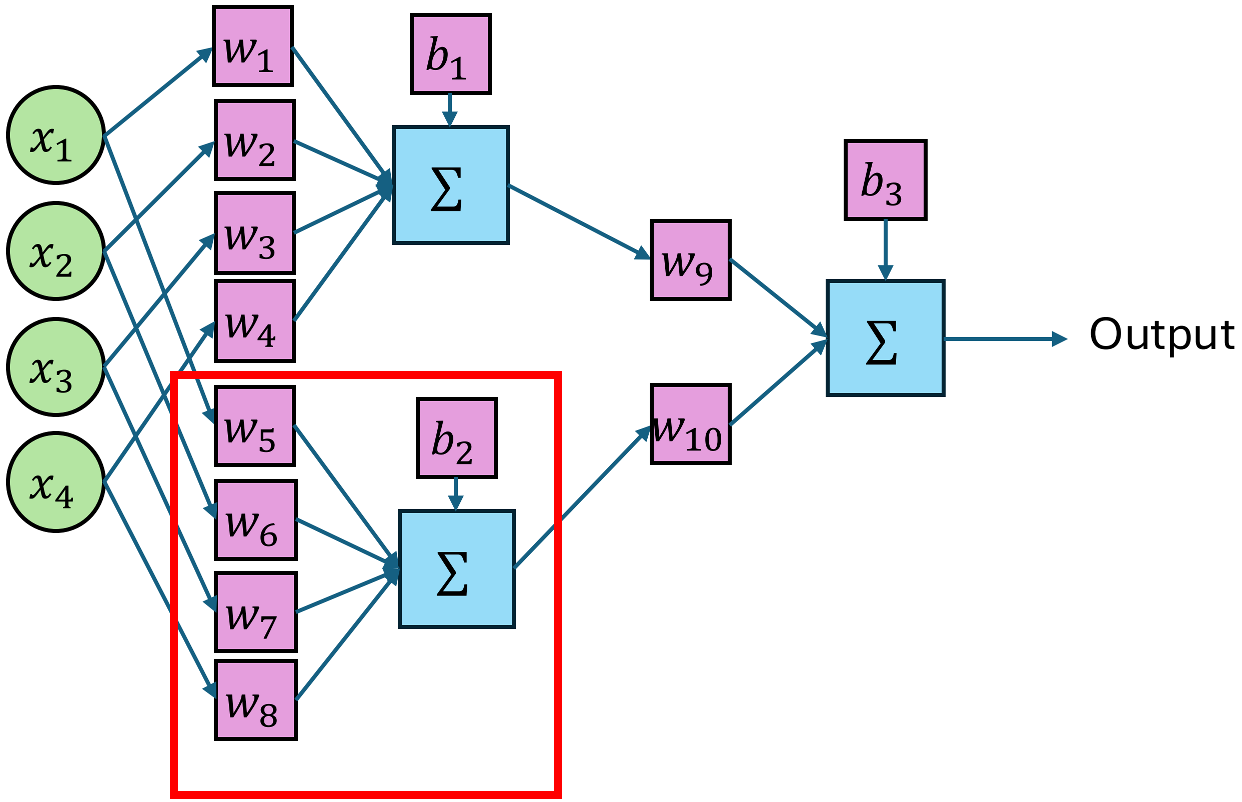
Let $\mathrm{w}^{(1)} = [w_1, w_2, w_3, w_4]$
Perceptron #1: $\mathrm{z}_1 = x^T w^{(1)} + b_1$



Multi-Layer Perceptrons

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

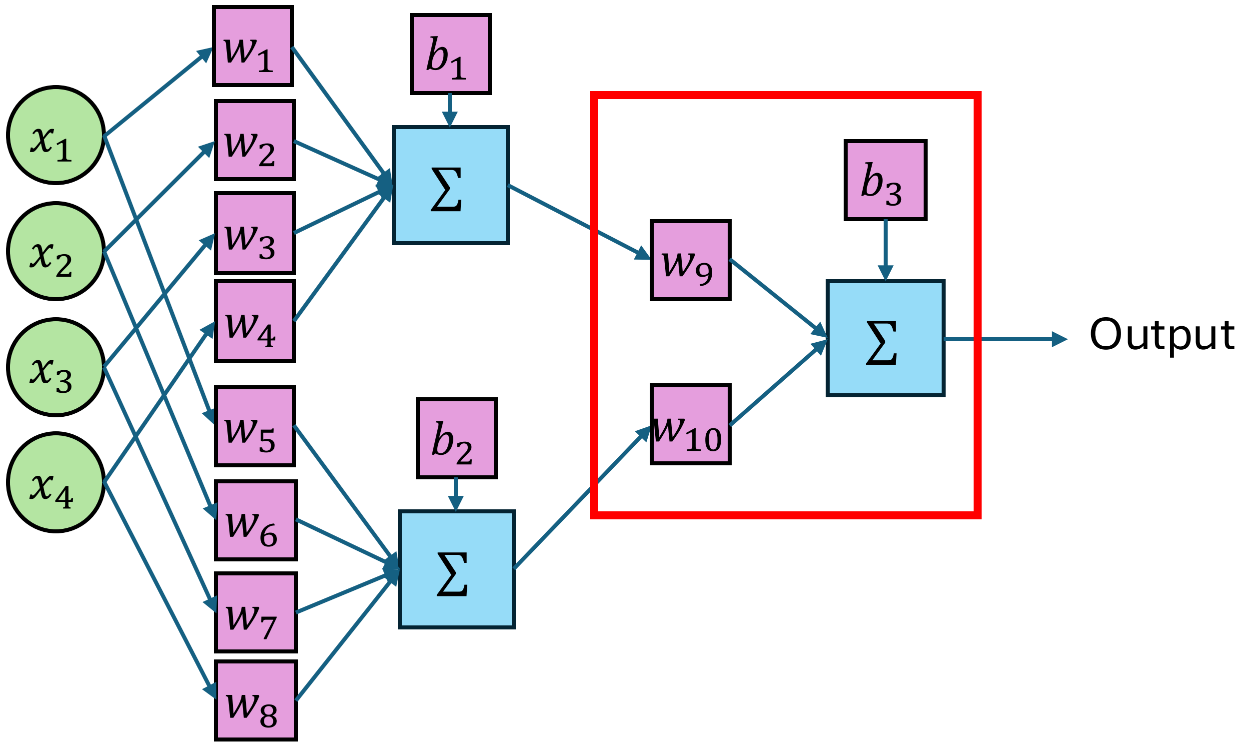
Let $\mathrm{w}^{(2)} = [w_5, w_6, w_7, w_8]$
Perceptron #1: $\mathrm{z}_1 = x^T w^{(1)} + b_1$
Perceptron #2: $z_2 = x^T w^{(2)} + b_2$



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: $z_1 = x^T w^{(1)} + b_1$
Perceptron #2: $z_2 = x^T w^{(2)} + b_2$
Perceptron #3: $z_3 = z_1 w_9 + z_2 w_{10} + b_3$

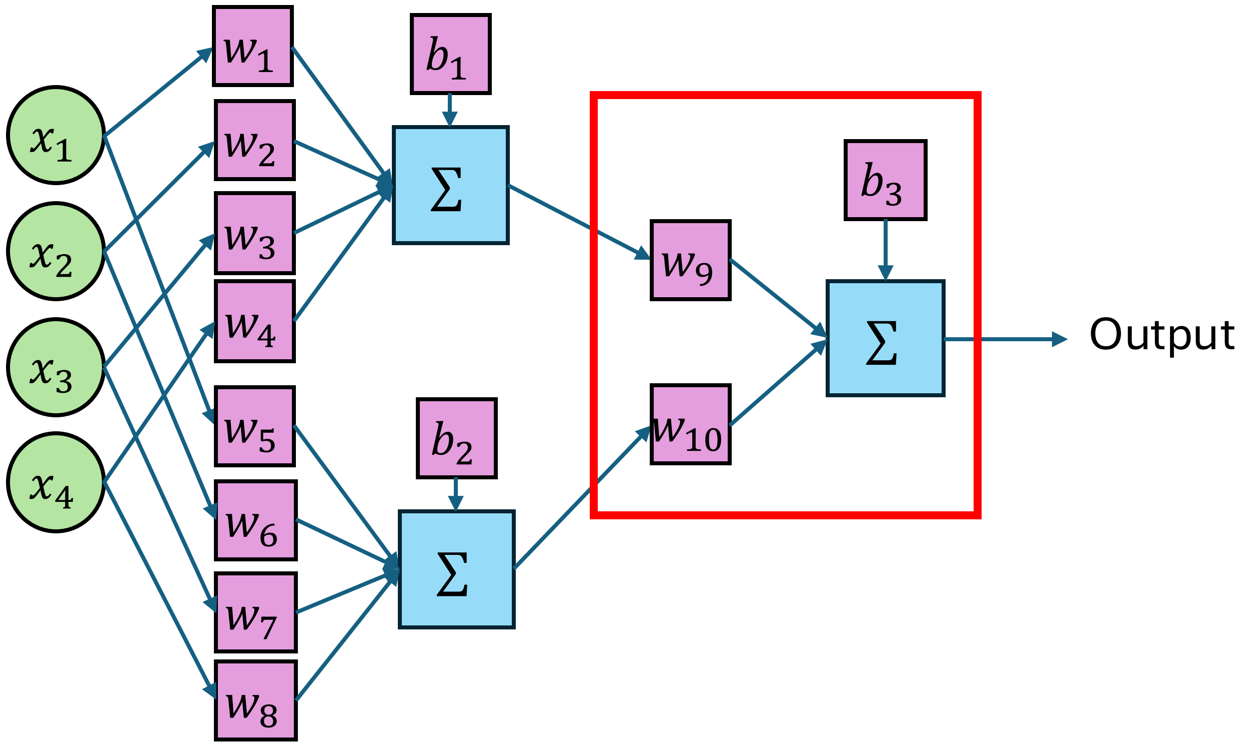


Multi-Layer Perceptrons

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: $\mathrm{z}_1 = x^T w^{(1)} + b_1$
Perceptron #2: $z_2 = x^T w^{(2)} + b_2$
Perceptron #3: $z_3 = z_1 w_9 + z_2 w_{10} + b_3$

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

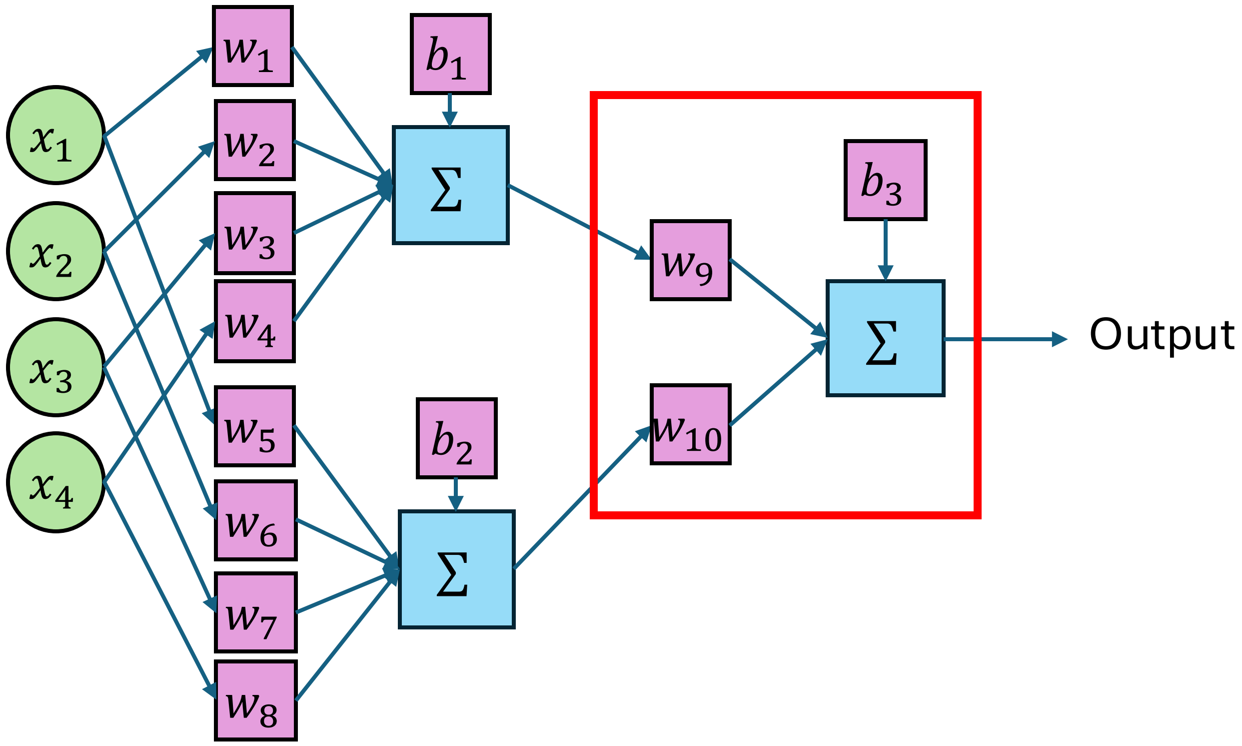


Multi-Layer Perceptrons

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

Let $\mathrm{w}^{(2)} = [w_5, w_6, w_7, w_8]$
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:
$z = w_9\left(x^T w^{(1)} + b_1\right) + w_{10}\left(x^T w^{(2)} + b_2\right) + b_3$



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

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

Just a vector...

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...

Just a vector...

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

With no activation function

MLP's Expressiveness

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

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

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

Just a vector...

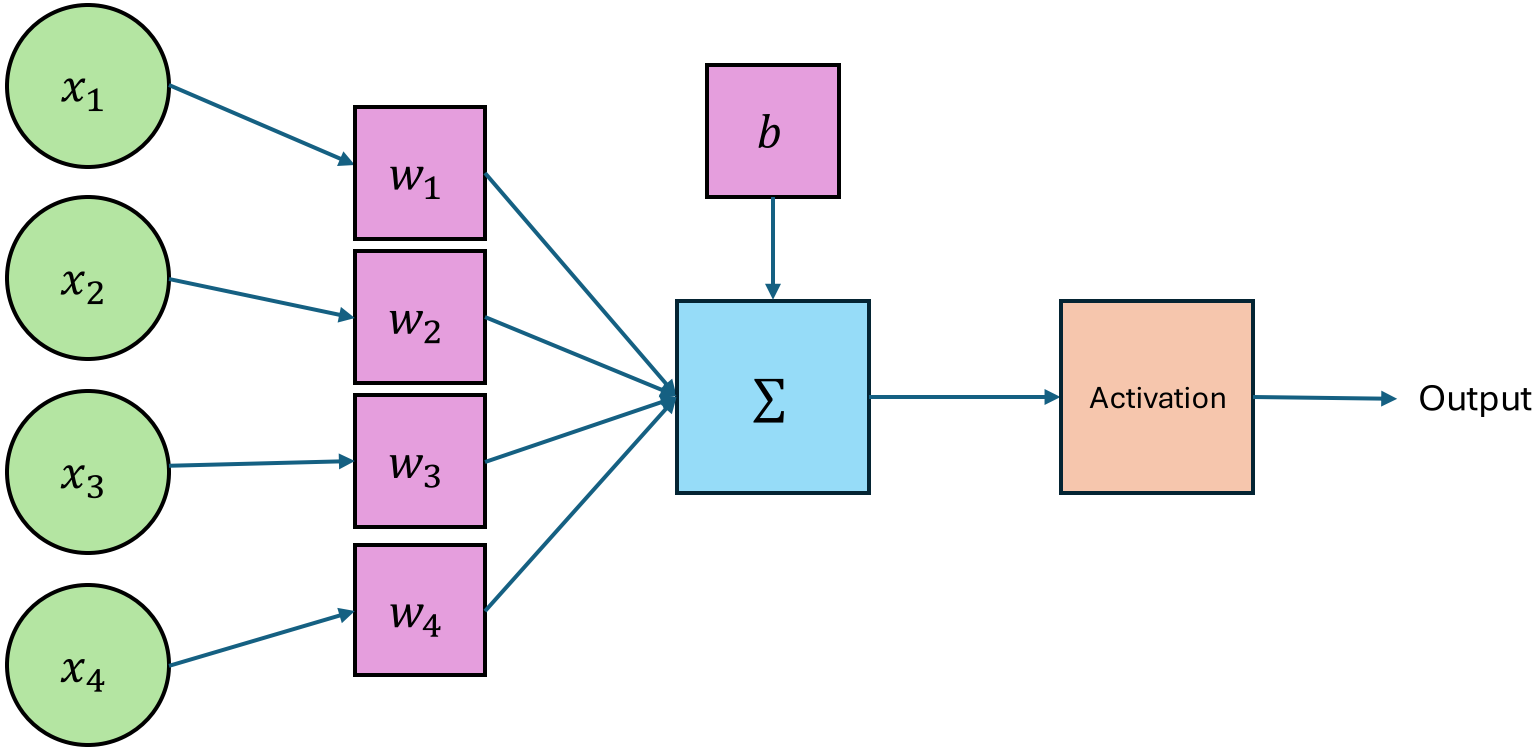
Just a vector...

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

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

Activation Functions

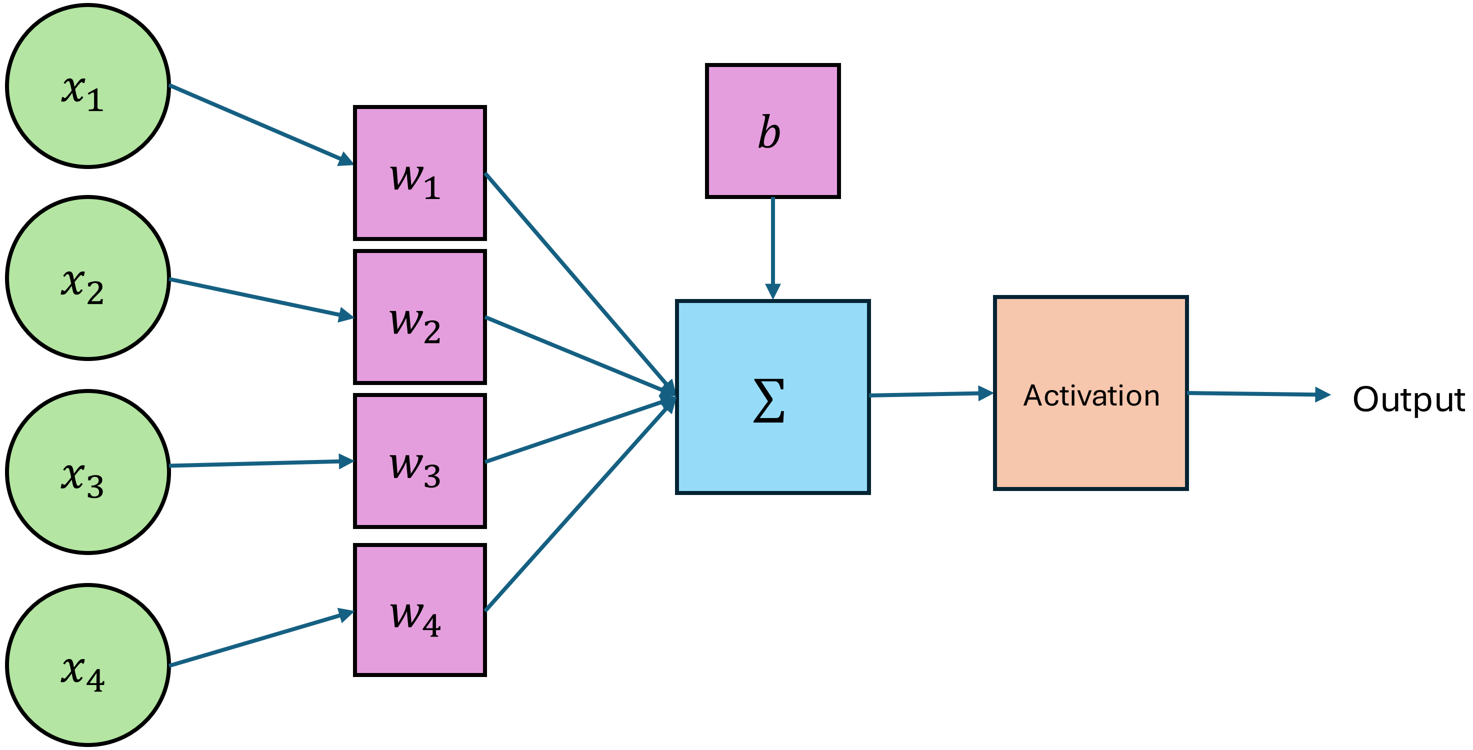
Non-linear functions applied to output of neuron



Activation Functions

Non-linear functions applied to output of neuron

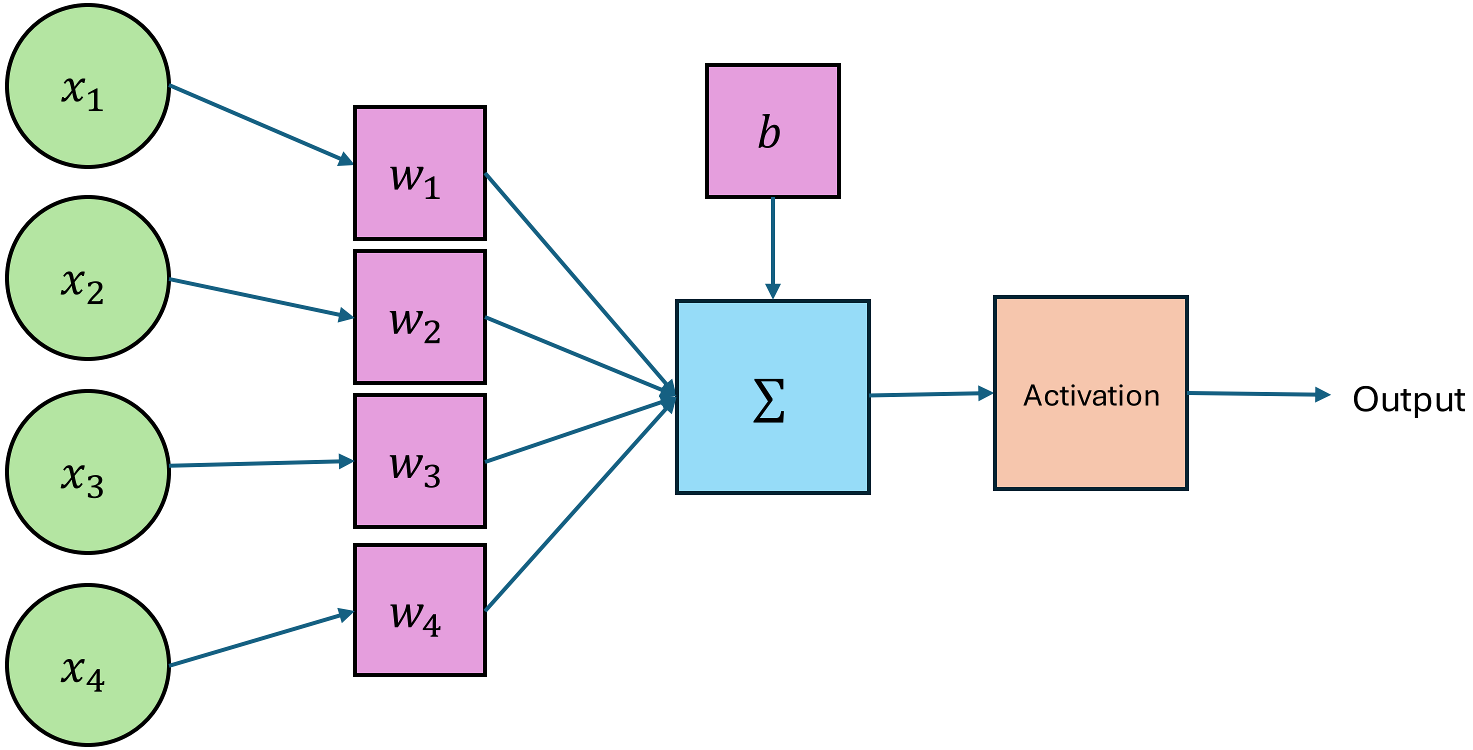
In the perceptron case, the activation function is the threshold



Activation Functions

Non-linear functions applied to output of neuron

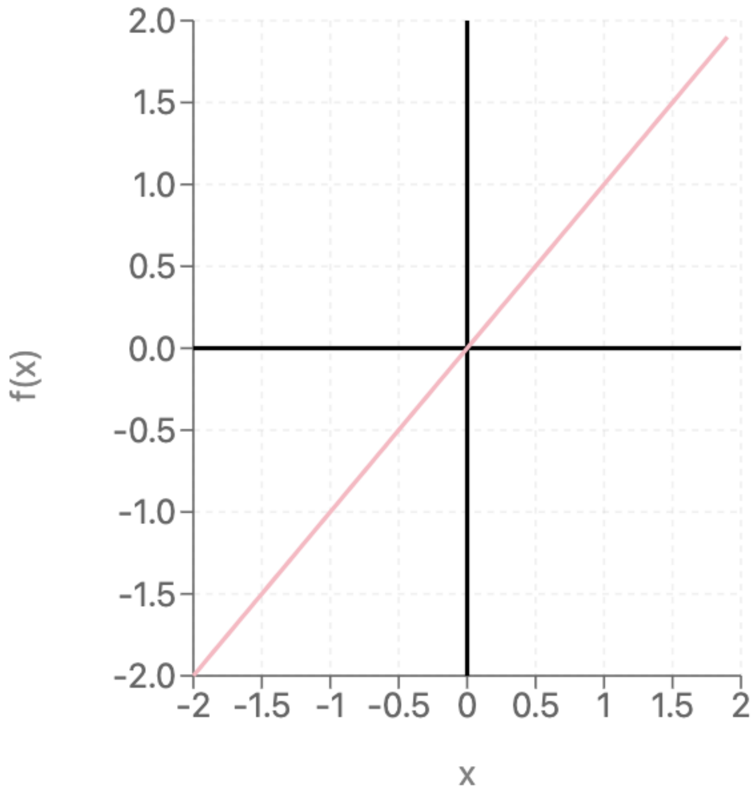
In the perceptron case, the activation function is the threshold



Common Activation Functions

Linear (No Activation)

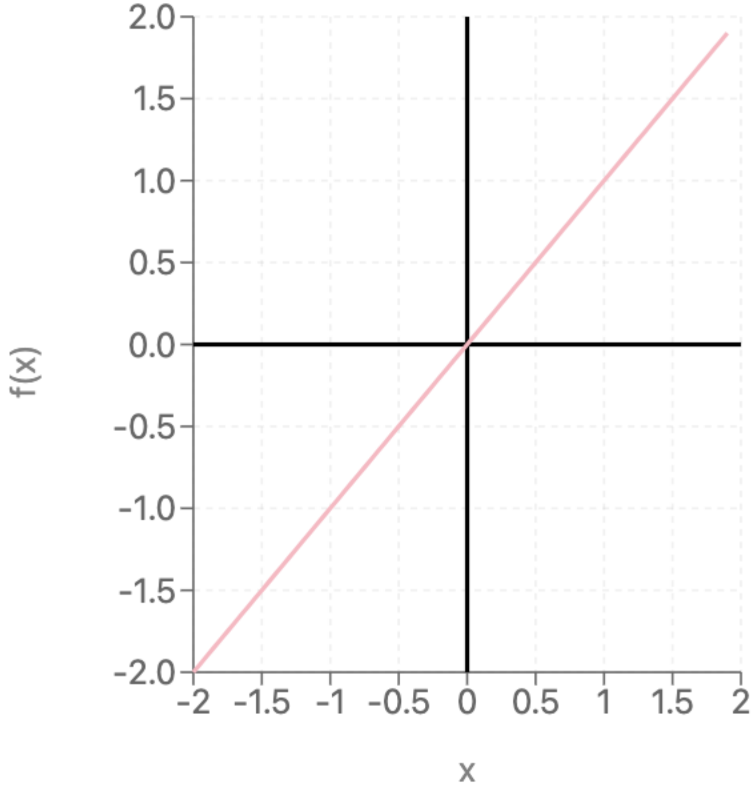
f(x) = x



Common Activation Functions

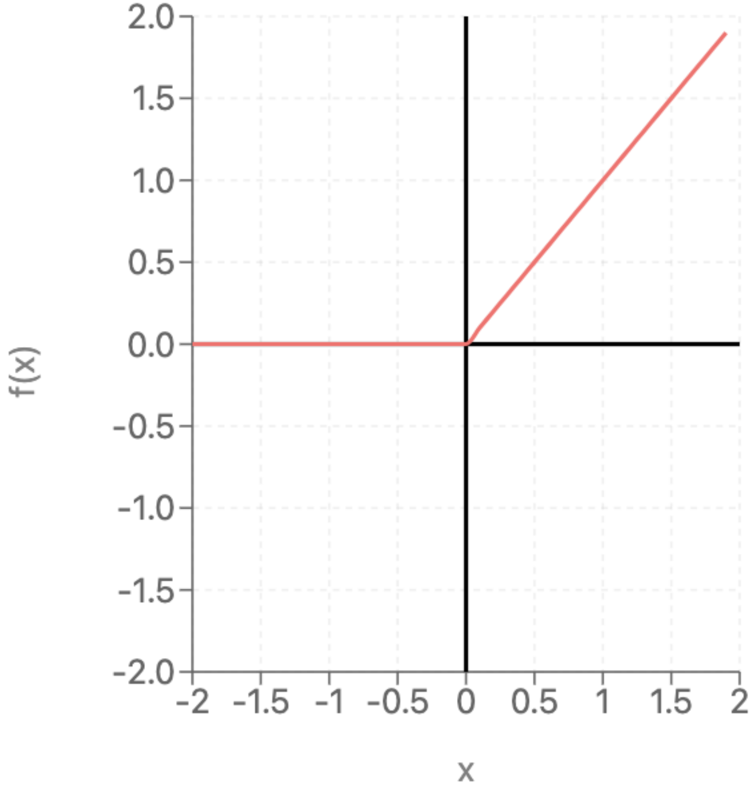
Linear (No Activation)

f(x) = x



ReLU

f(x) = max(0, x)

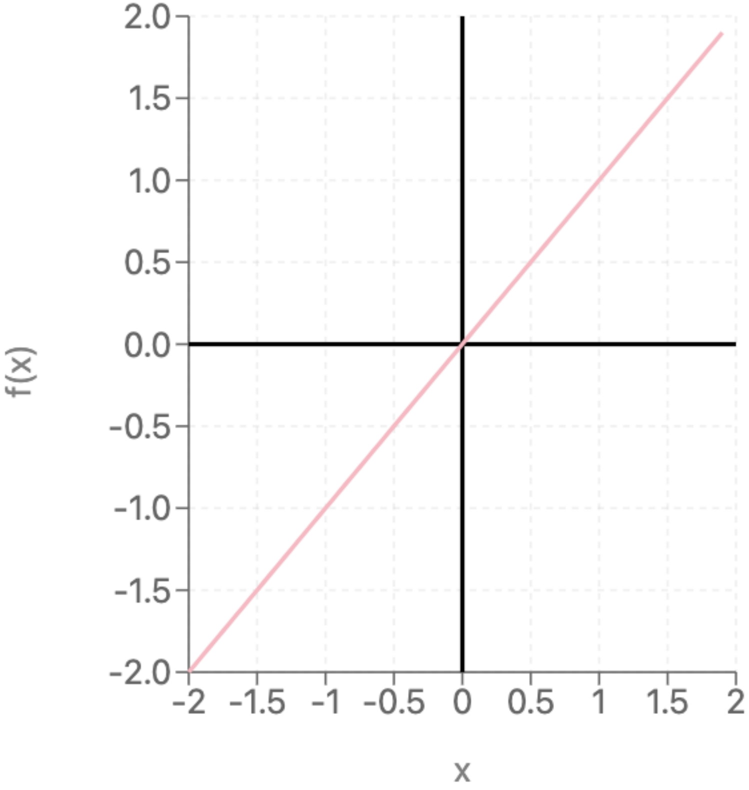


Common Activation Functions

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

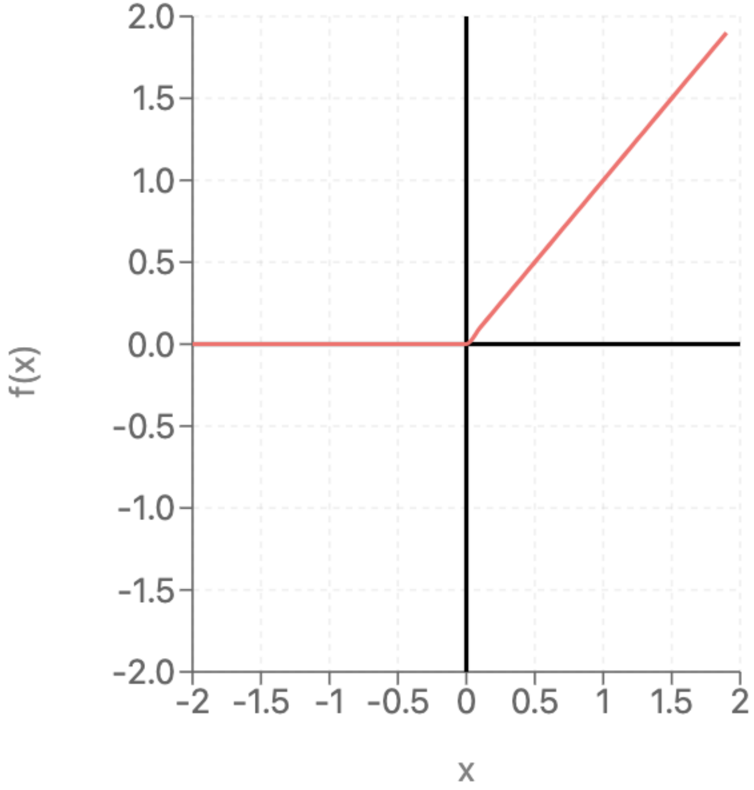
Linear (No Activation)

f(x) = x



ReLU

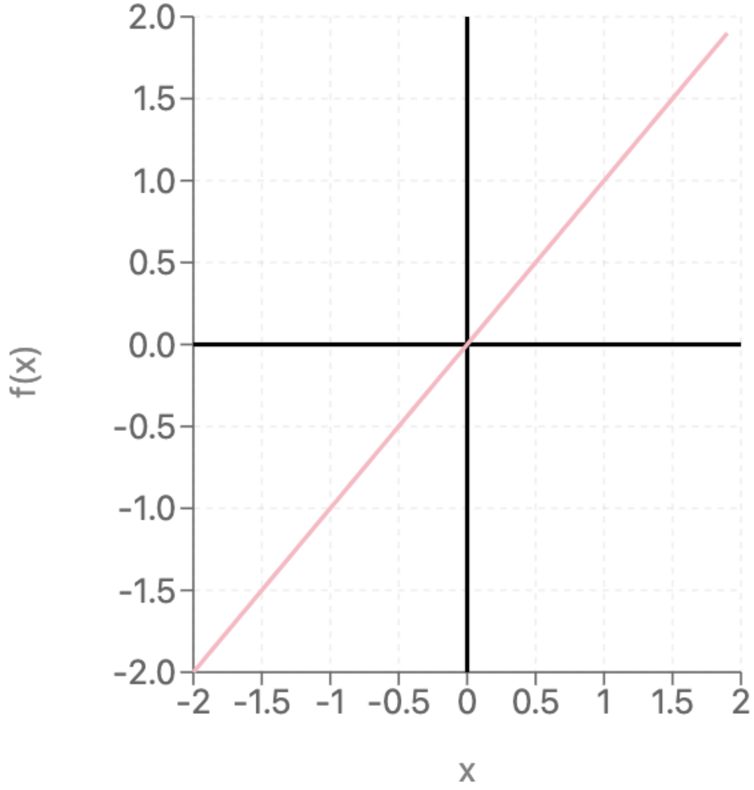
f(x) = max(0, x)



Common Activation Functions

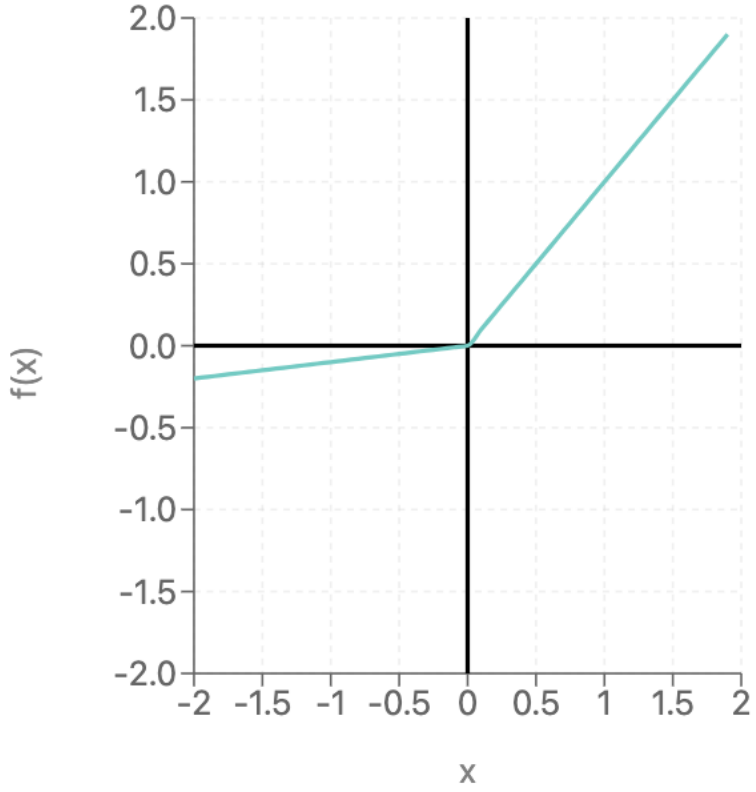
Linear (No Activation)

f(x) = x



Leaky ReLU

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

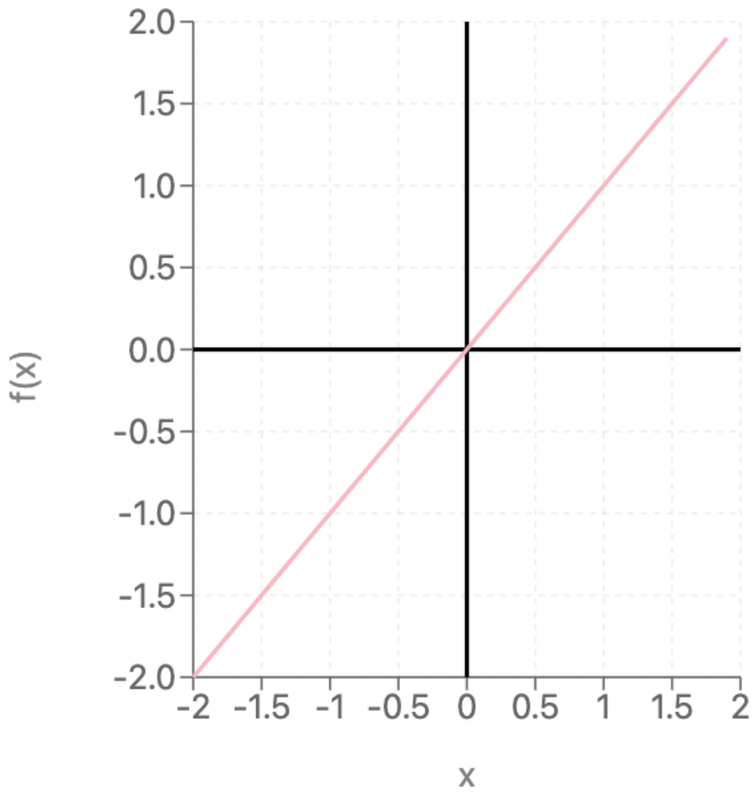


Common Activation Functions

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

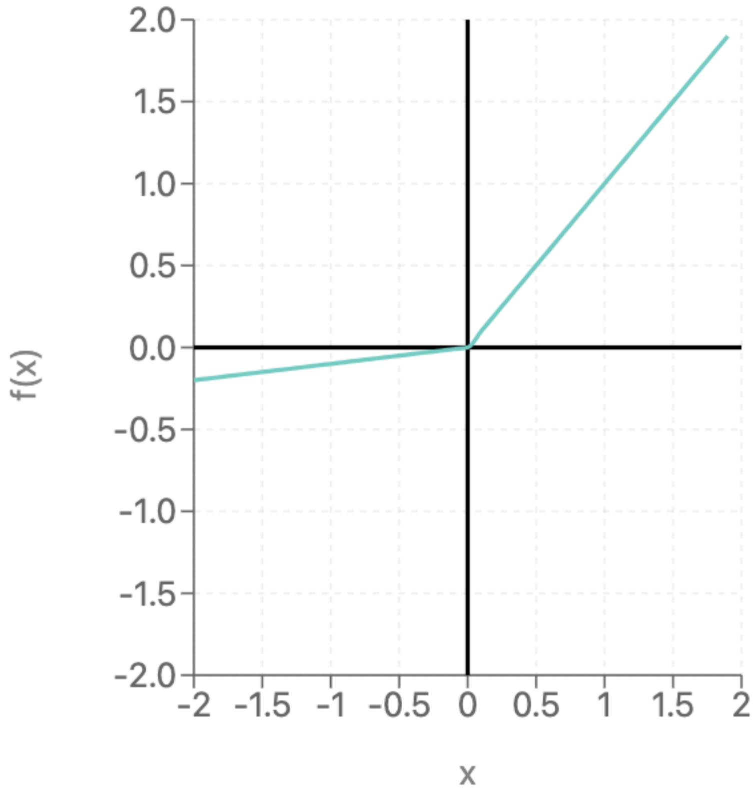
Linear (No Activation)

f(x) = x

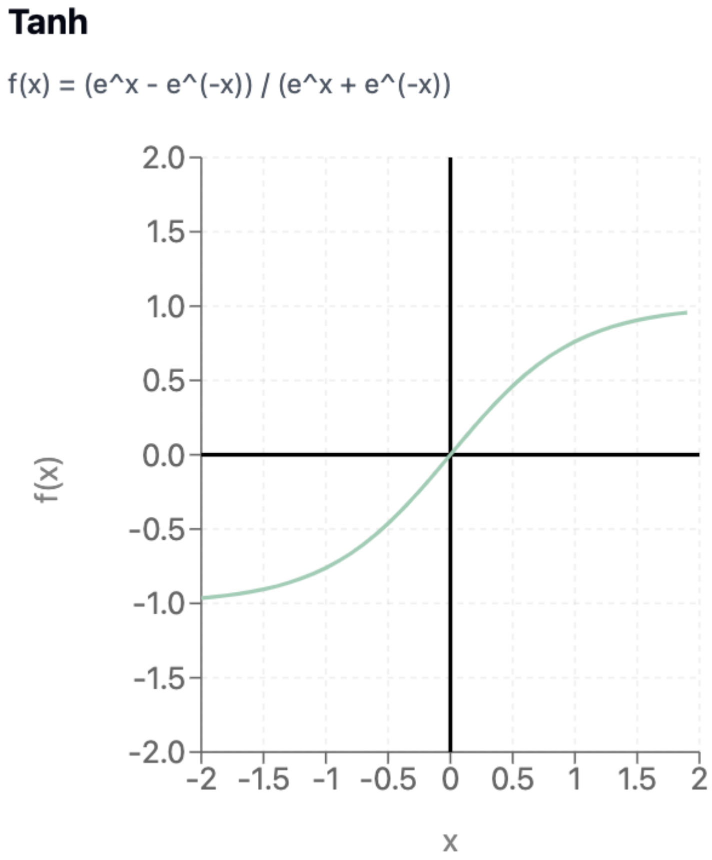


Leaky ReLU

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



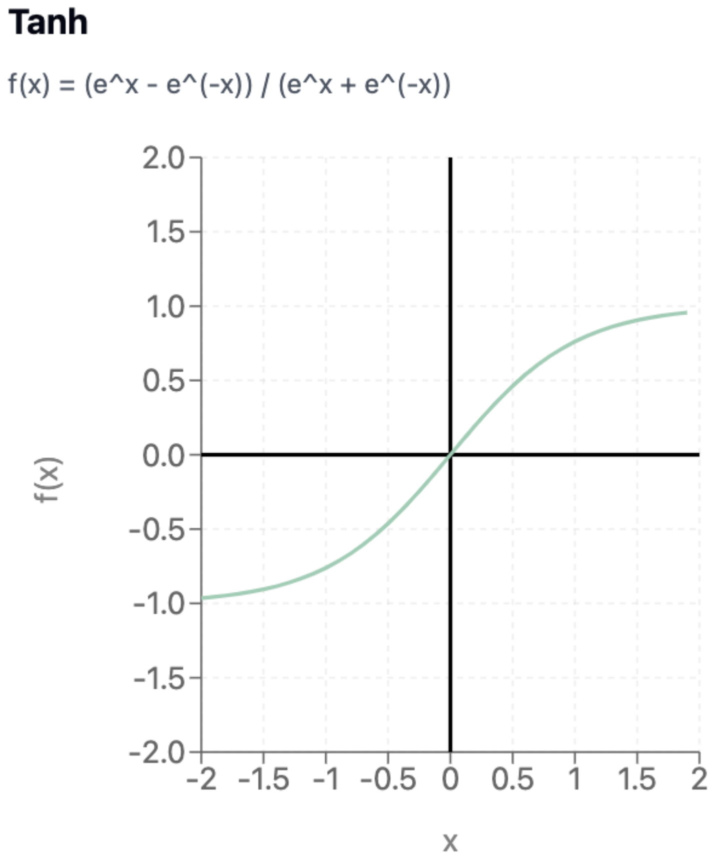
Tanh



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

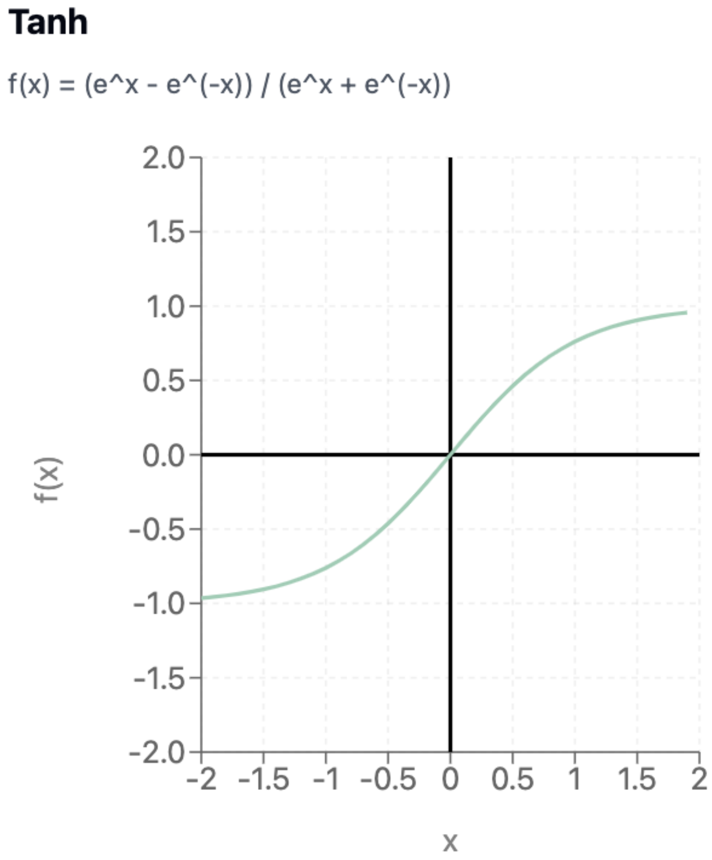
Disadvantages:
- Slower to compute
- Extreme differences in input to activation can get squashed (i.e., z=100 will be very close to z=10000)



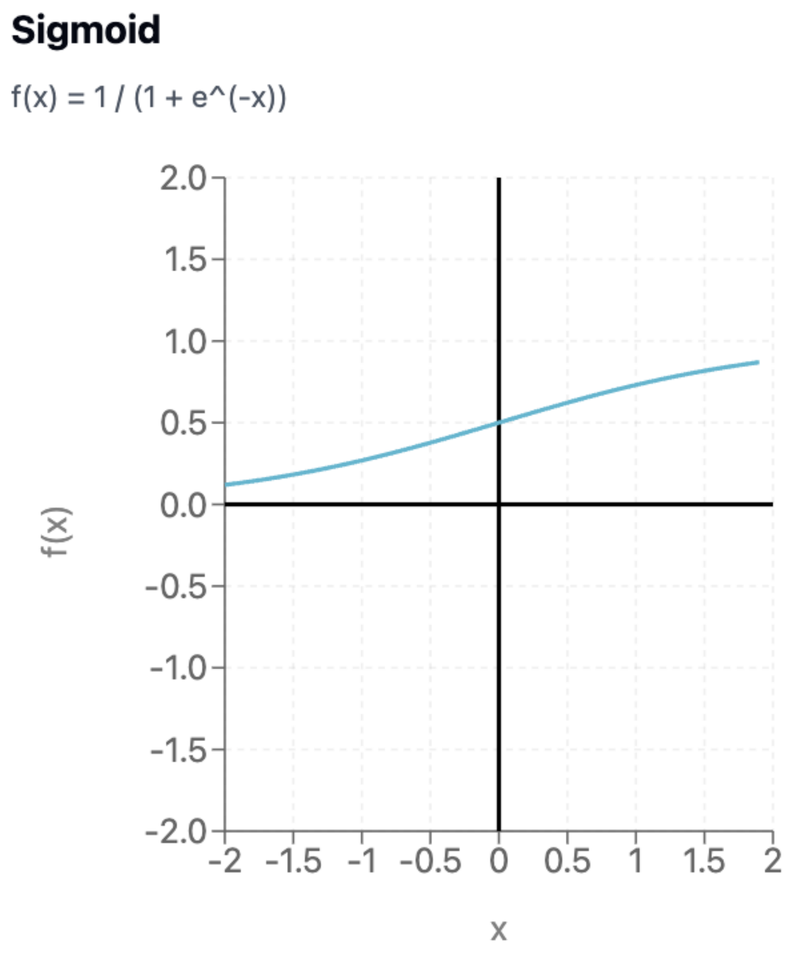
Tanh

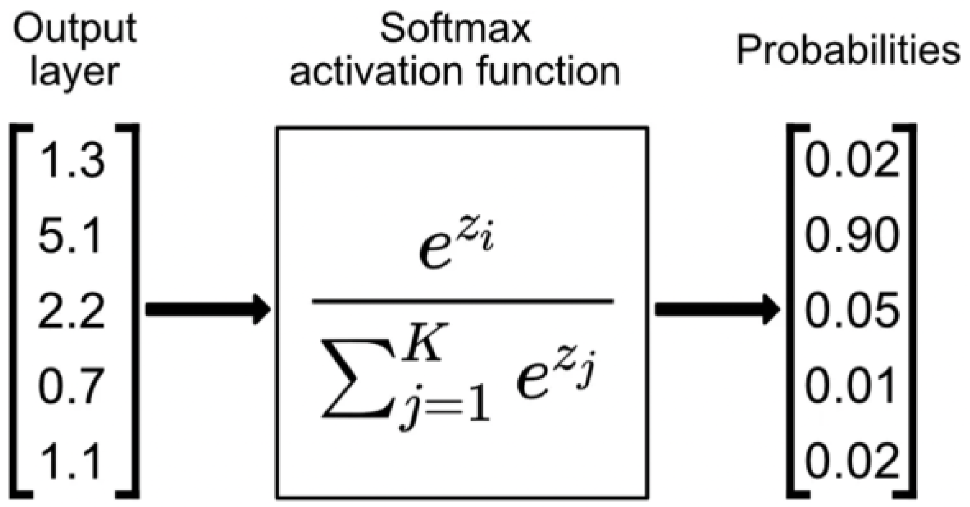
Tanh:
Advantages:
- Always maps output between -1 and 1 (learning is easier when input is normalized and this holds for intermediate layers as well)
- Continuously differentiable

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



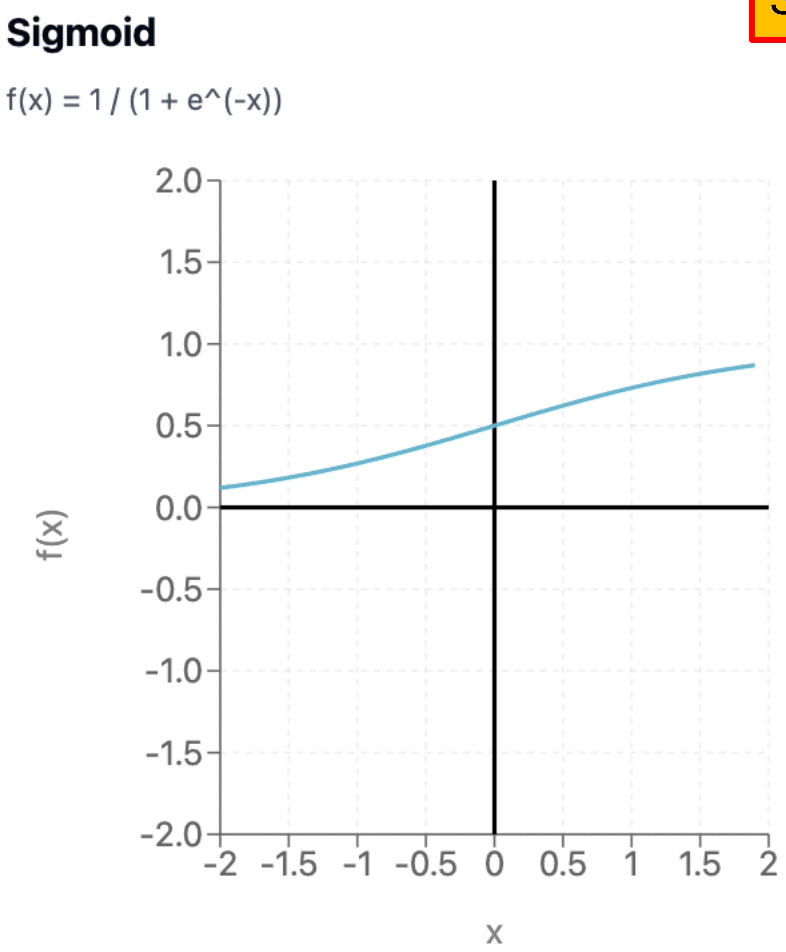
Special Activation Functions for Output

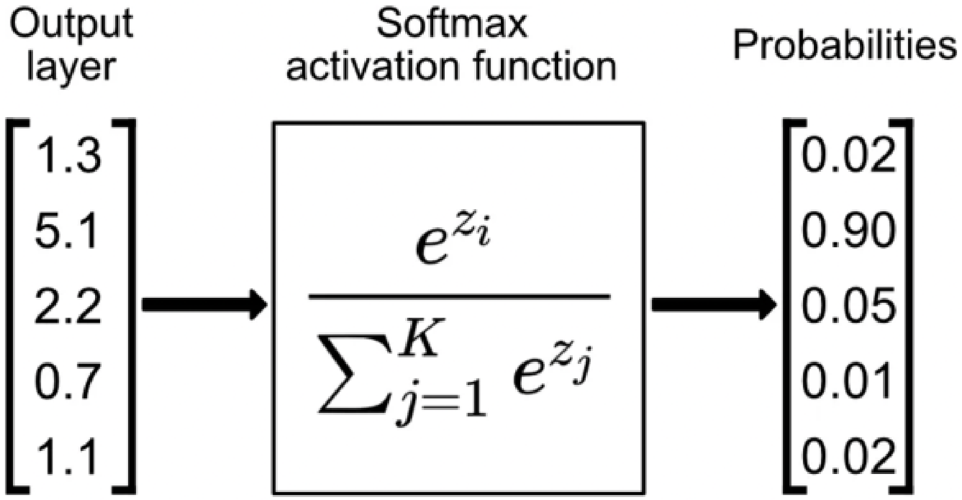




Special Activation Functions for Output

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

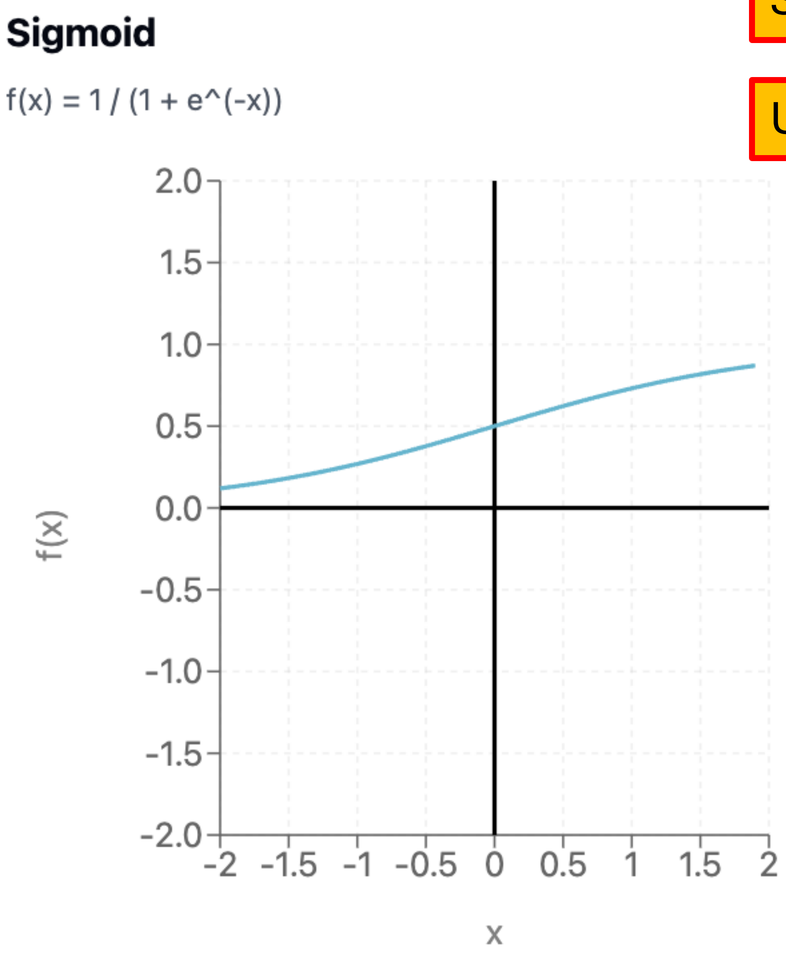


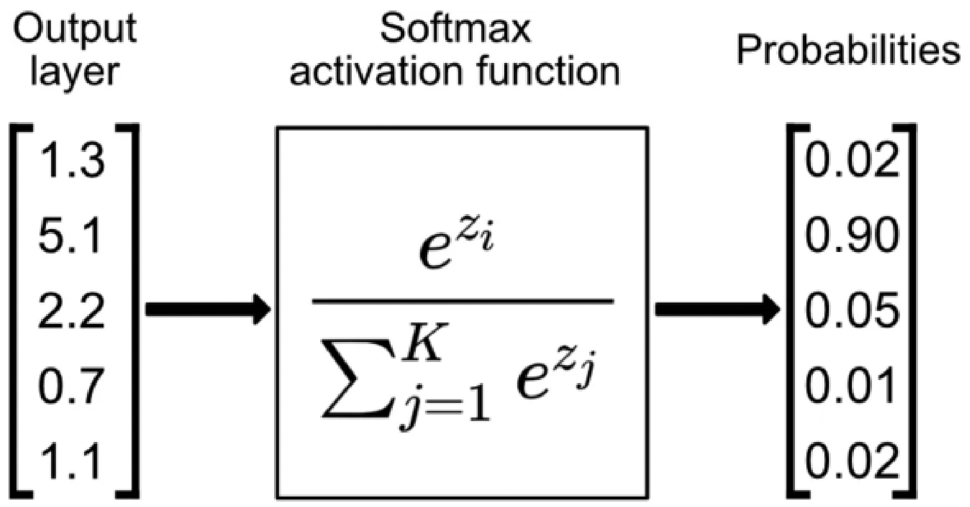


Special Activation Functions for Output

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

Used for classification tasks

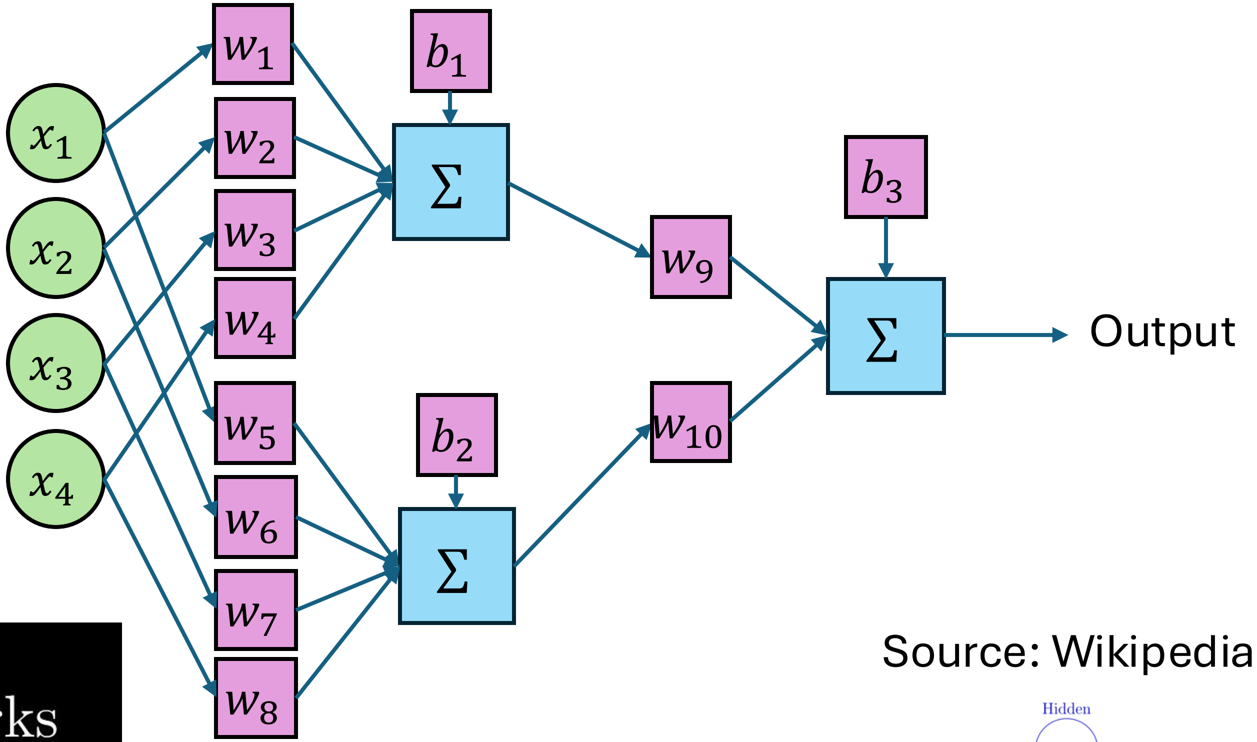




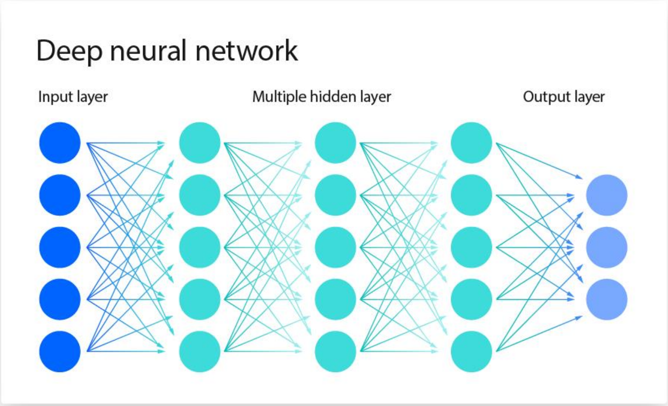
MLPs With Activation Functions

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

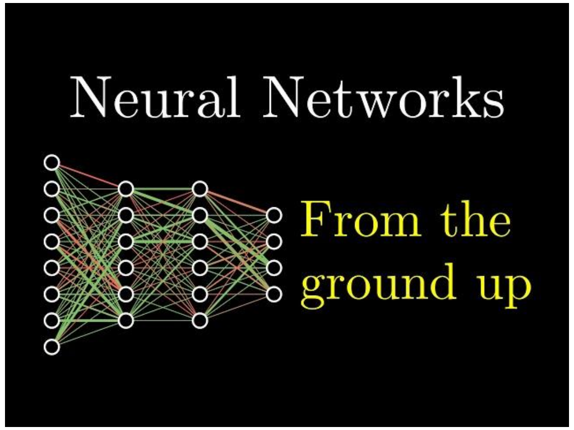
Source: Me



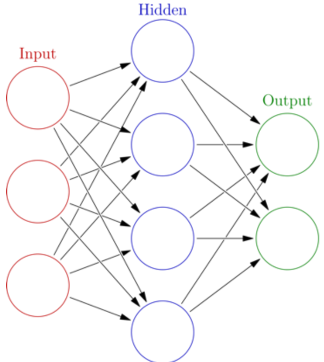
Source: IBM



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.

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

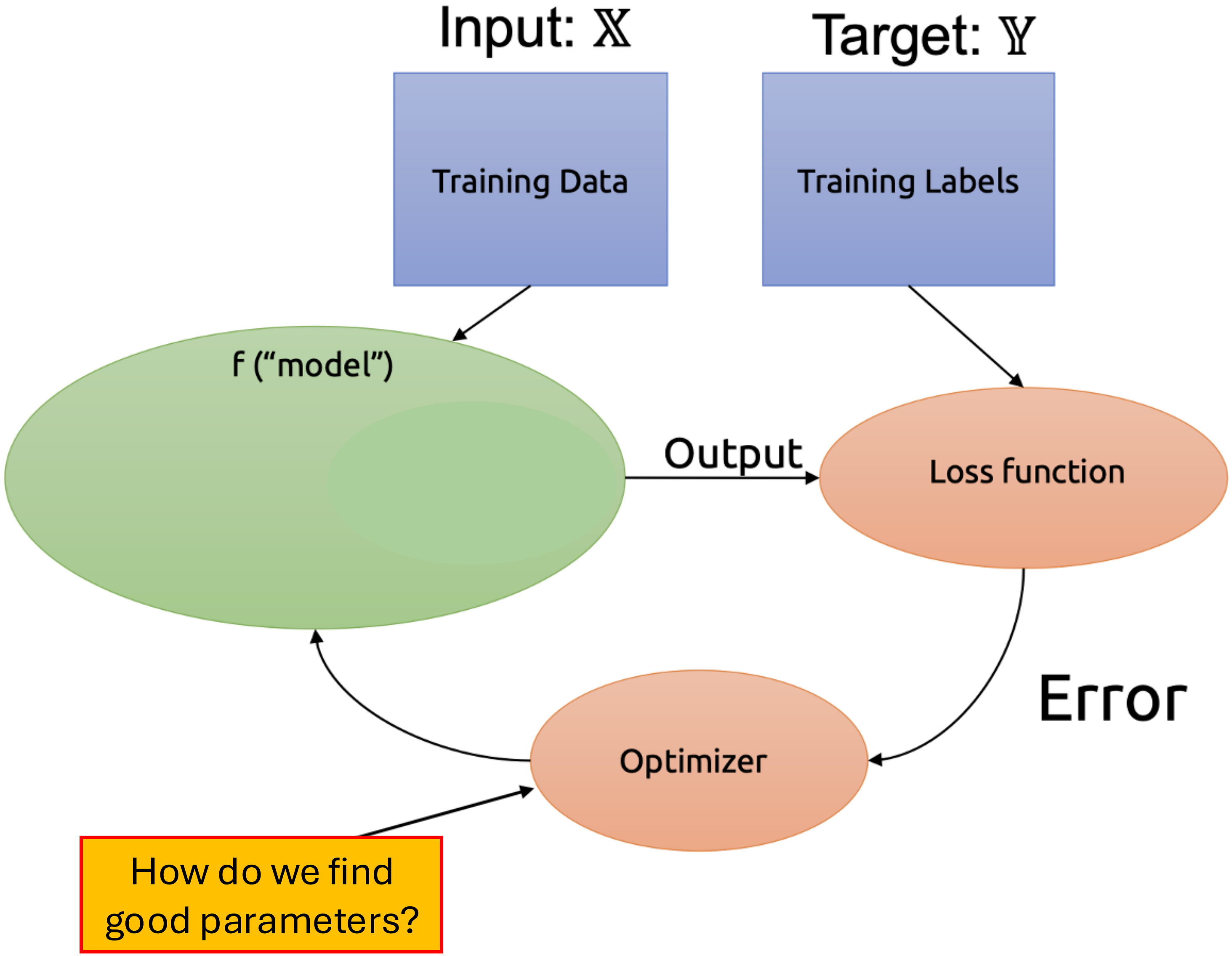
Piecewise polynomials are universal function approximators (think Taylor expansions)

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

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:

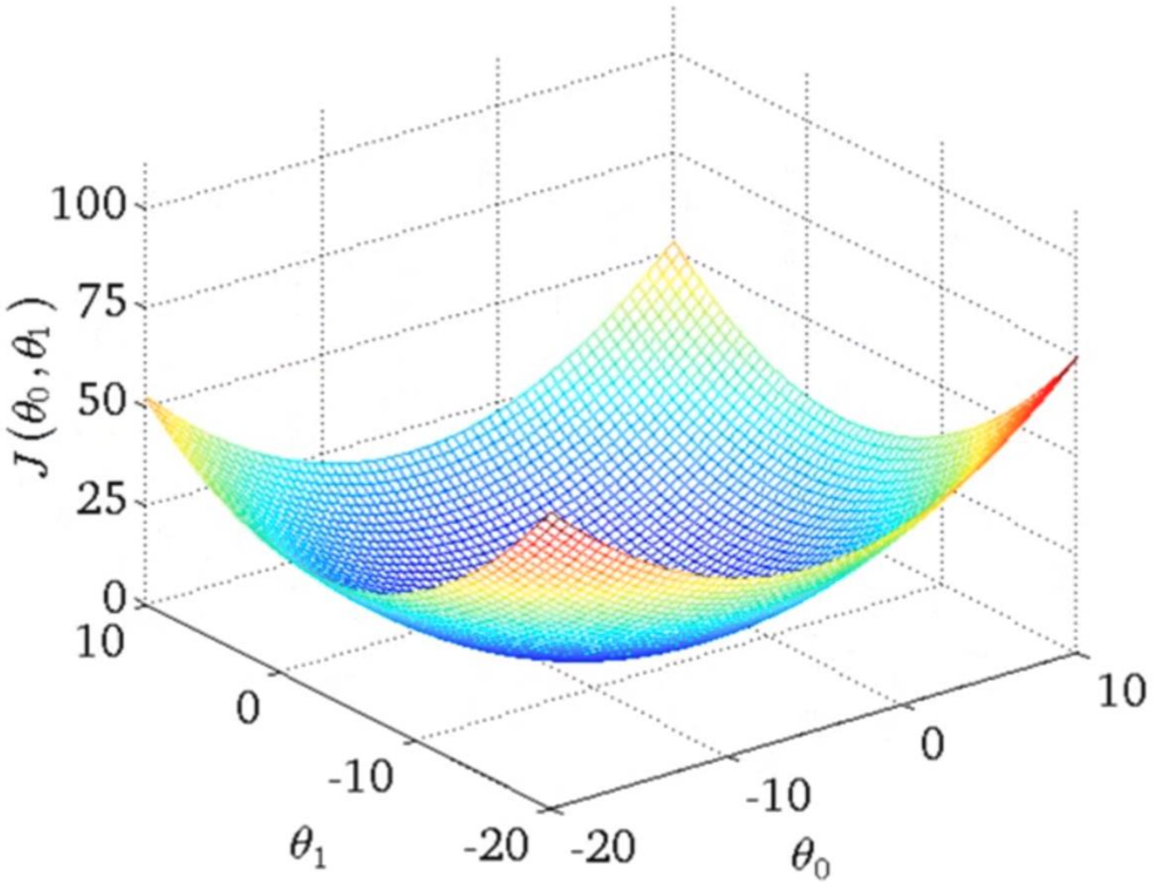
- Find derivative (or gradient) of loss function
- Set derivative to 0
- Solve for parameters

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 $\lambda \in [0, 1]$
- $\lambda f(x_1) + (1 - \lambda) f(x_2) \leq \lambda x_1 + (1 - \lambda) x_2$

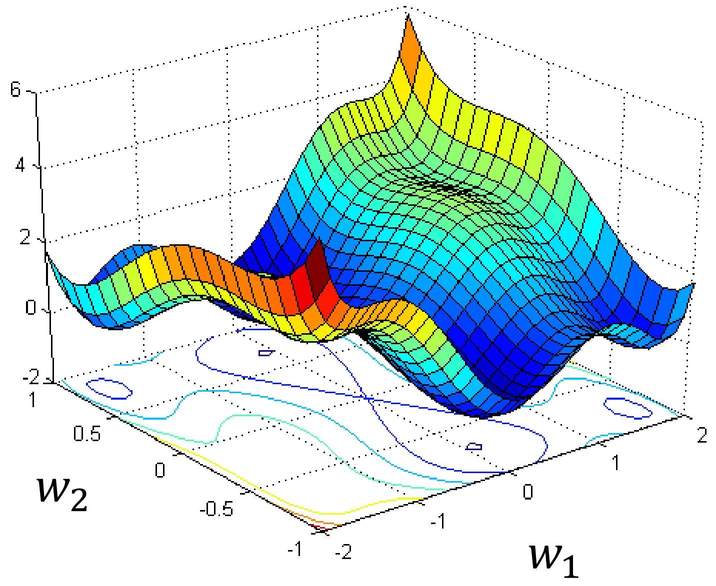
The line connecting any two points on the graph will always be above the function.

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

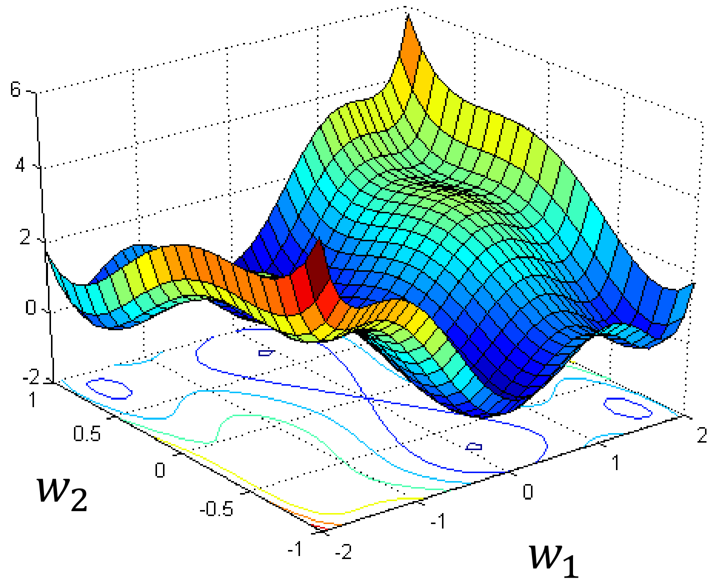
MSE is **not** convex with respect to network parameters when non-linear activations are involved.



Non-Convex Functions

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

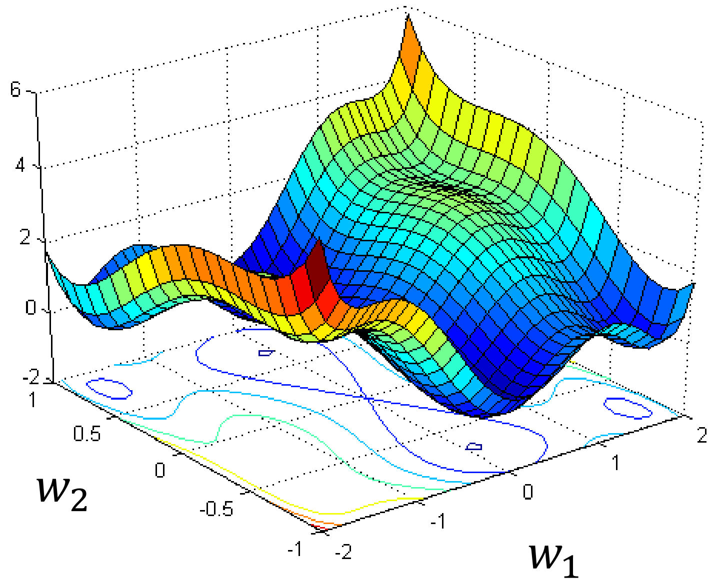
Multiple local minima



Non-Convex Functions

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

Multiple local minima

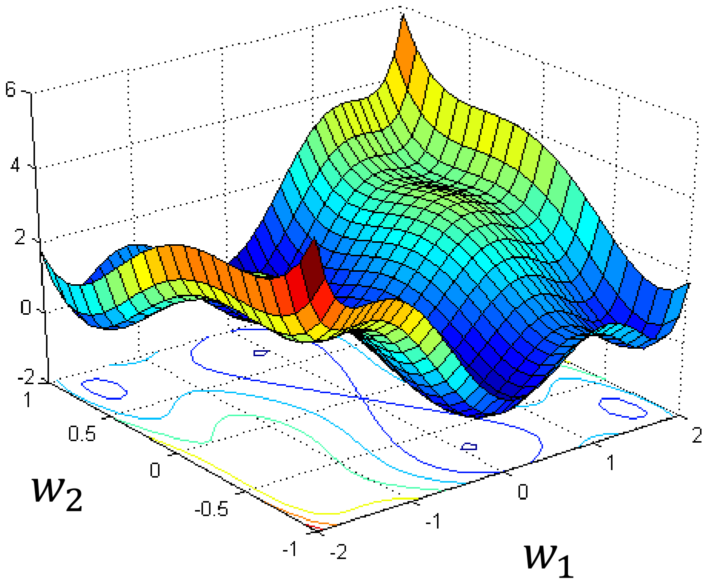


Local maxima

Non-Convex Functions

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

Multiple local minima



Saddle points

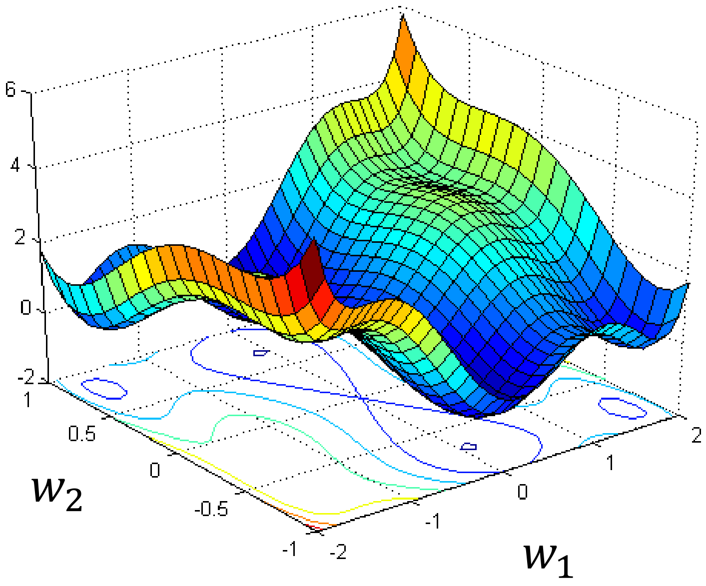
Local maxima

Non-Convex Functions

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

Multiple local minima

Saddle points

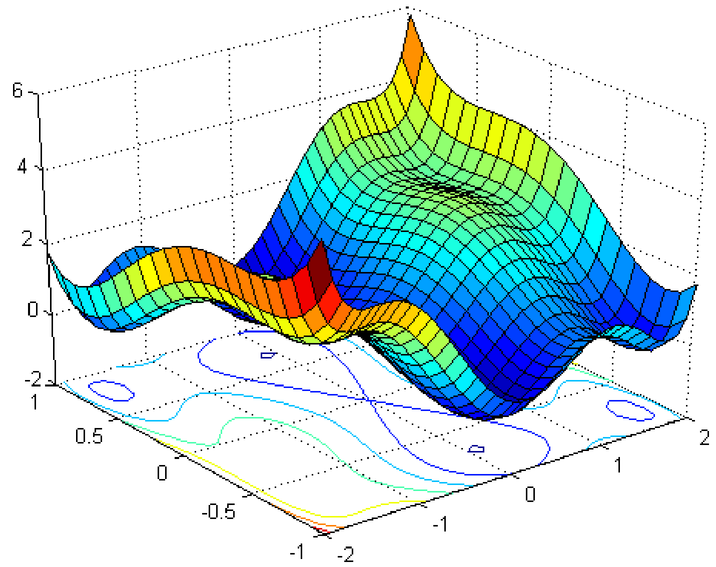


Local maxima

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

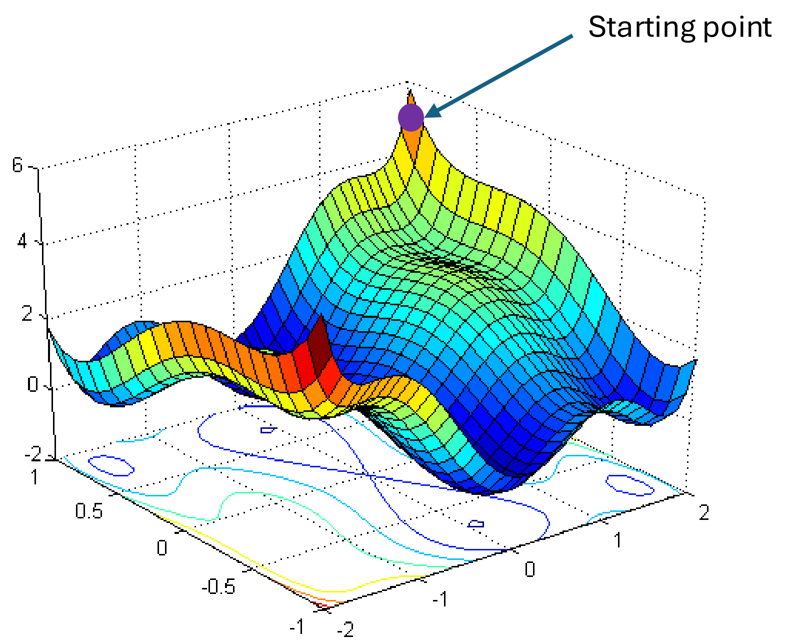
Option 2: Gradient Descent

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



Option 2: Gradient Descent

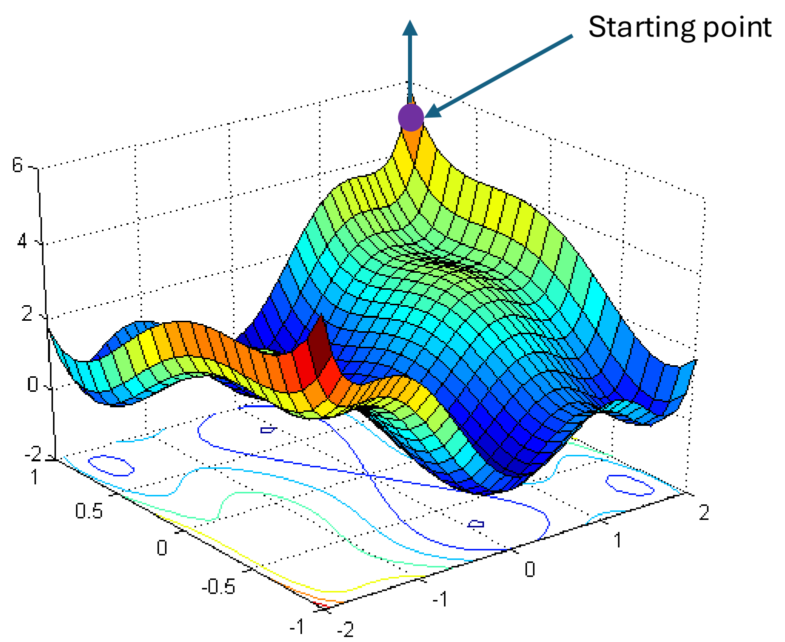
1. Start with some initial set of parameters
2. Take small step in the direction of the negative gradient
3. Repeat 2 until convergence



Option 2: Gradient Descent

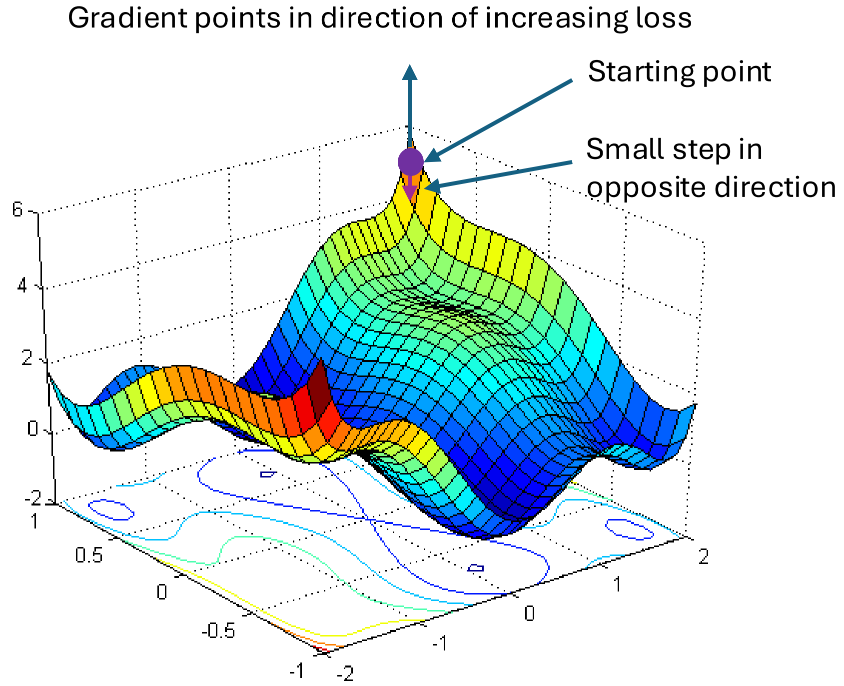
1. Start with some initial set of parameters
2. Take small step in the direction of the negative gradient
3. Repeat 2 until convergence

Gradient points in direction of increasing loss



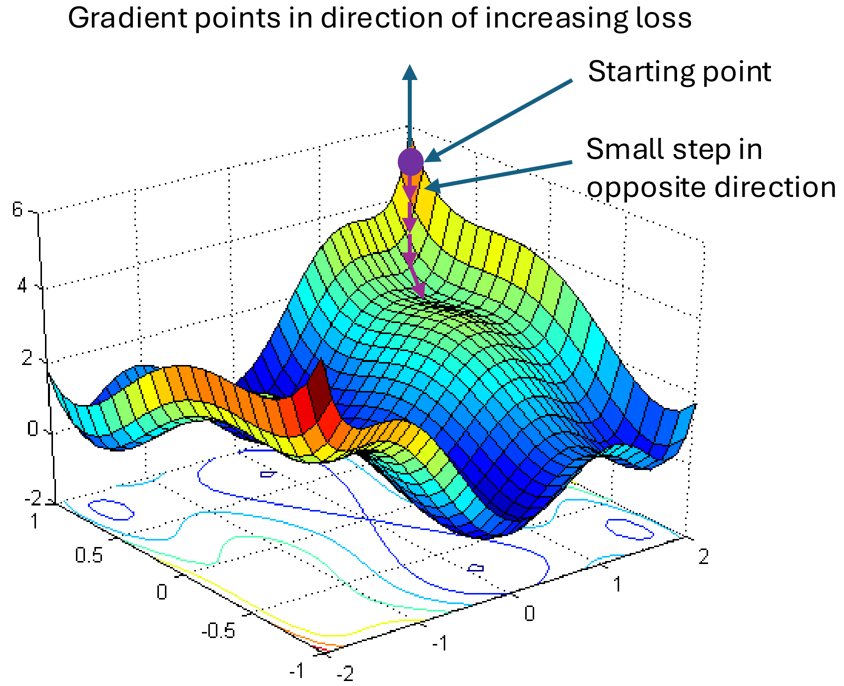
Option 2: Gradient Descent

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



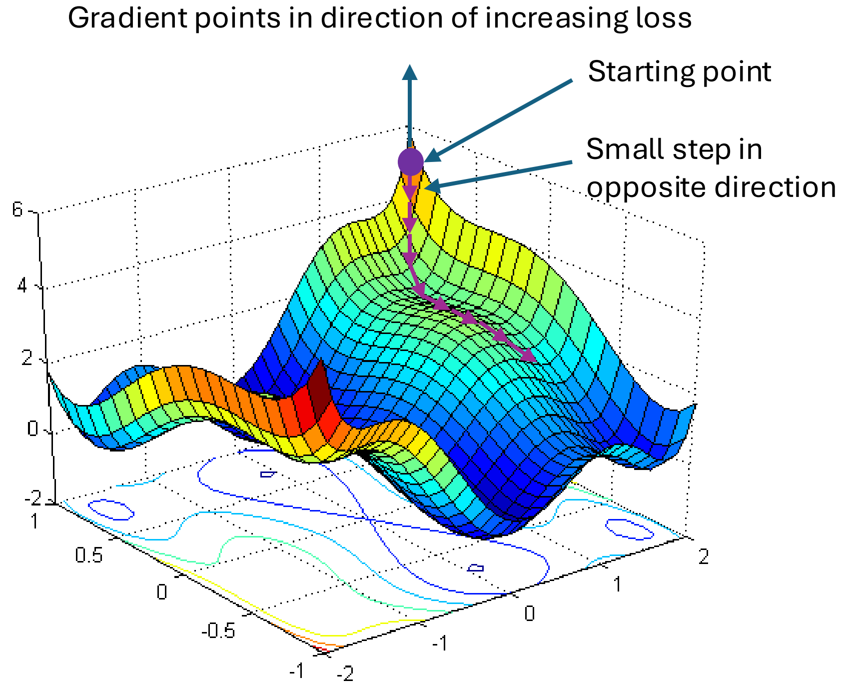
Option 2: Gradient Descent

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



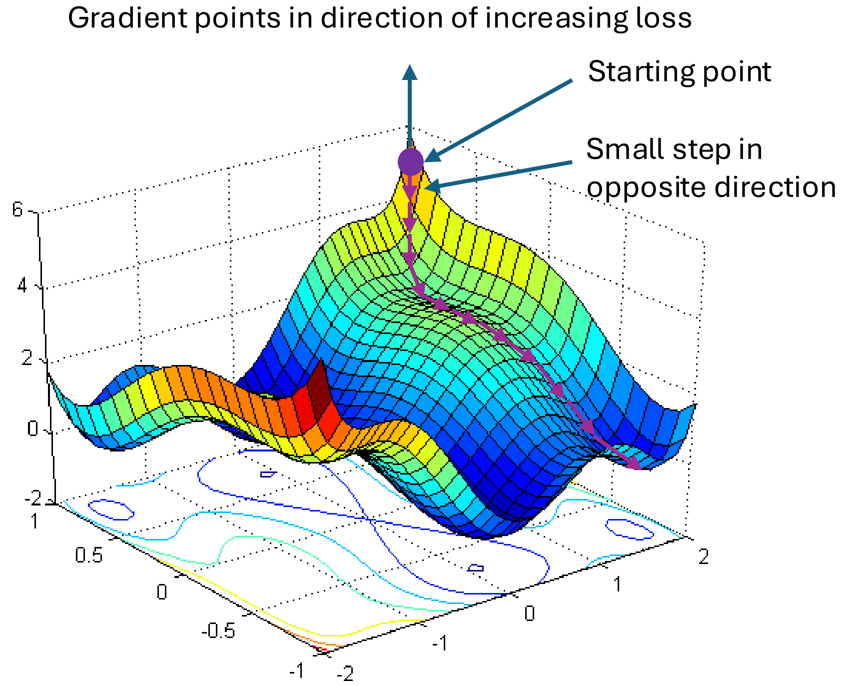
Option 2: Gradient Descent

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

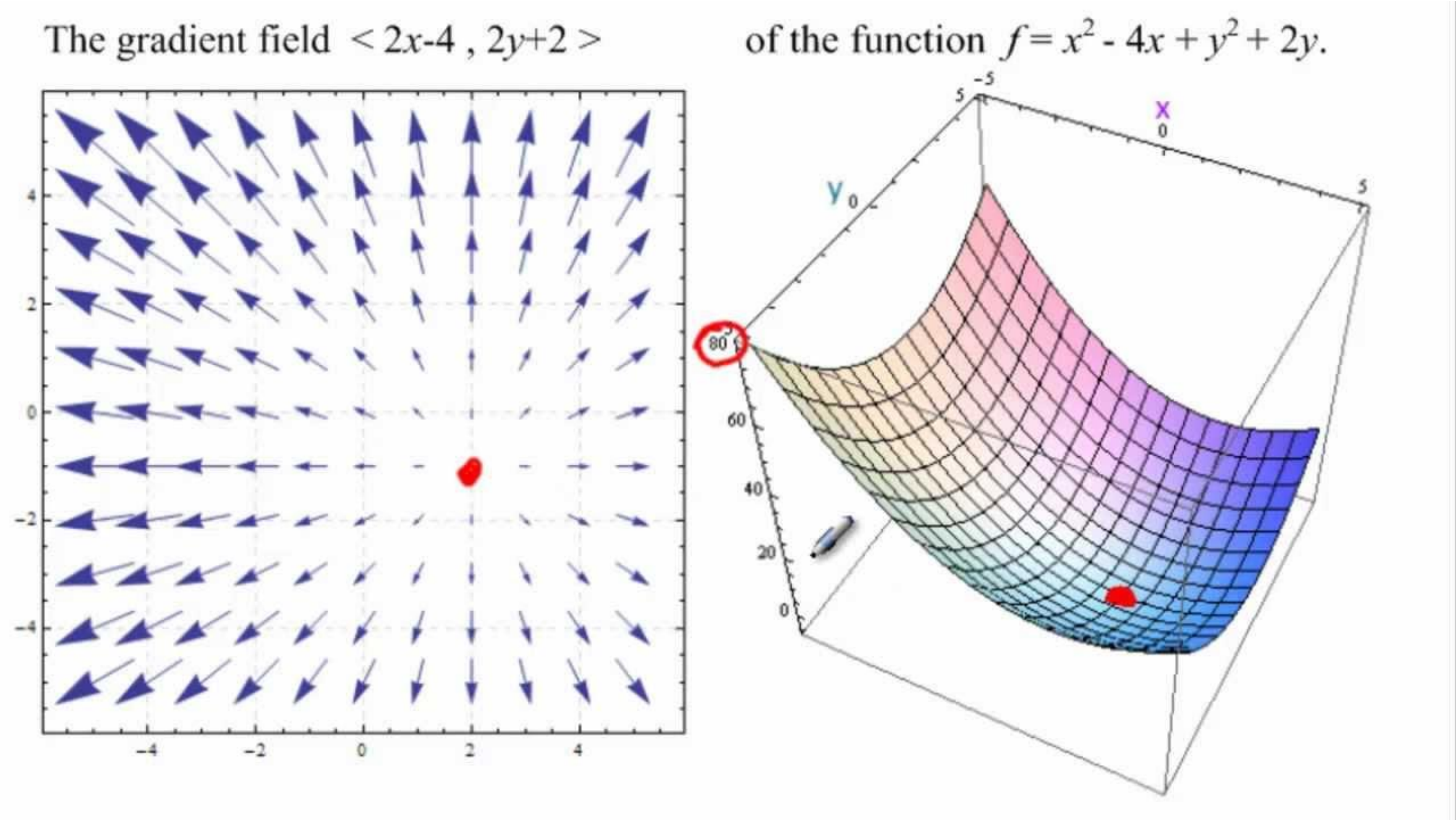
Vector “points” in direction of increasing f values.

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

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

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

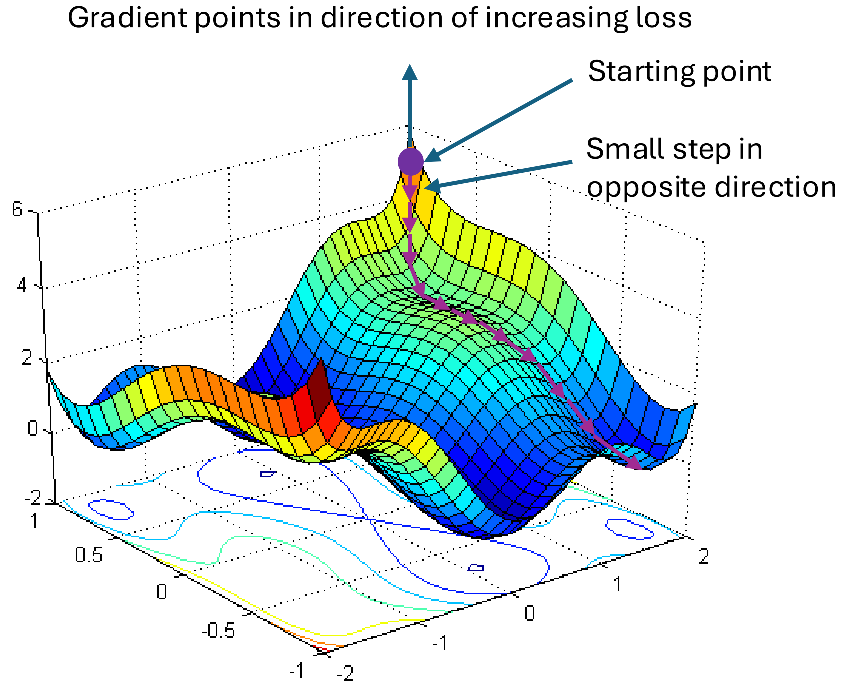
Gradients



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

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 $\Delta\theta < \epsilon$:

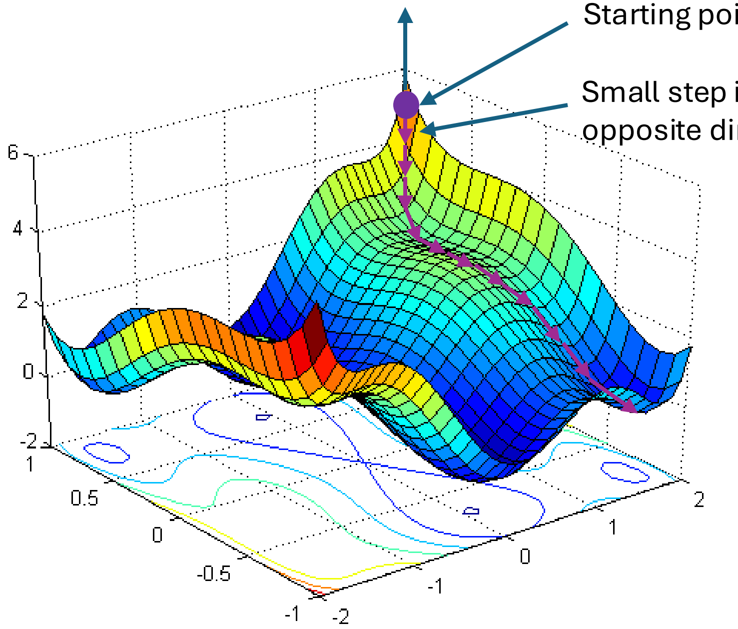
$$\vec{\theta} \leftarrow \theta - \alpha \nabla f_{\theta}$$

Gradient of what?

Gradient points in direction of increasing loss

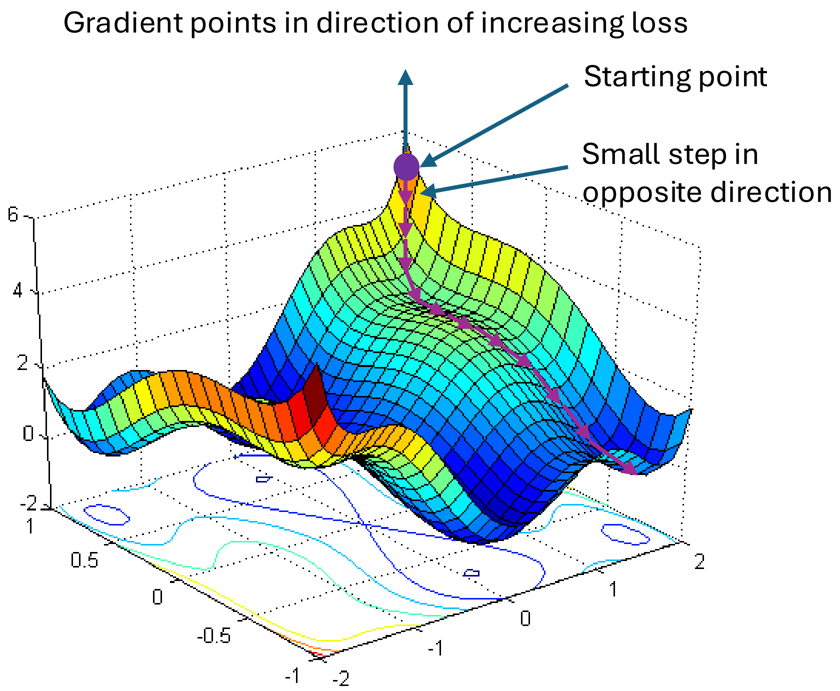
Starting point

Small step in opposite direction



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 $\Delta\theta < \epsilon$:

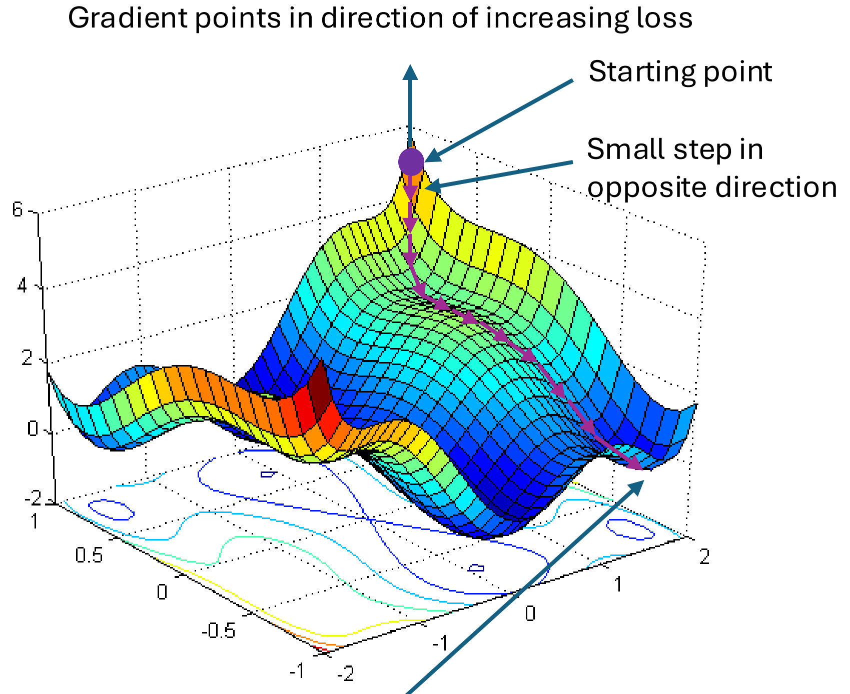
$$\vec{\theta} \leftarrow \theta - \alpha \nabla f_{\theta}$$

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 $\Delta\theta < \epsilon$:

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

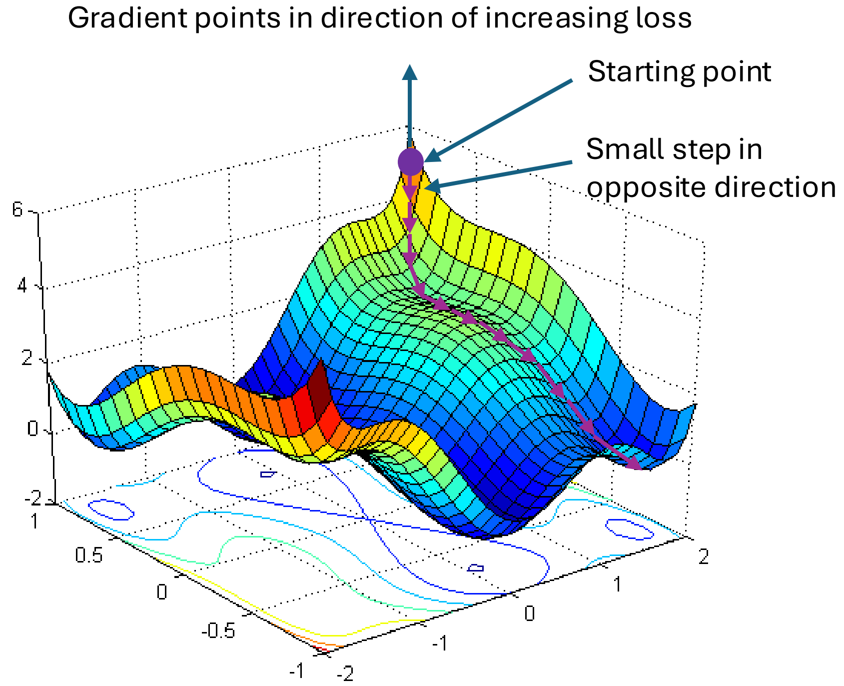
Gradient of what?

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



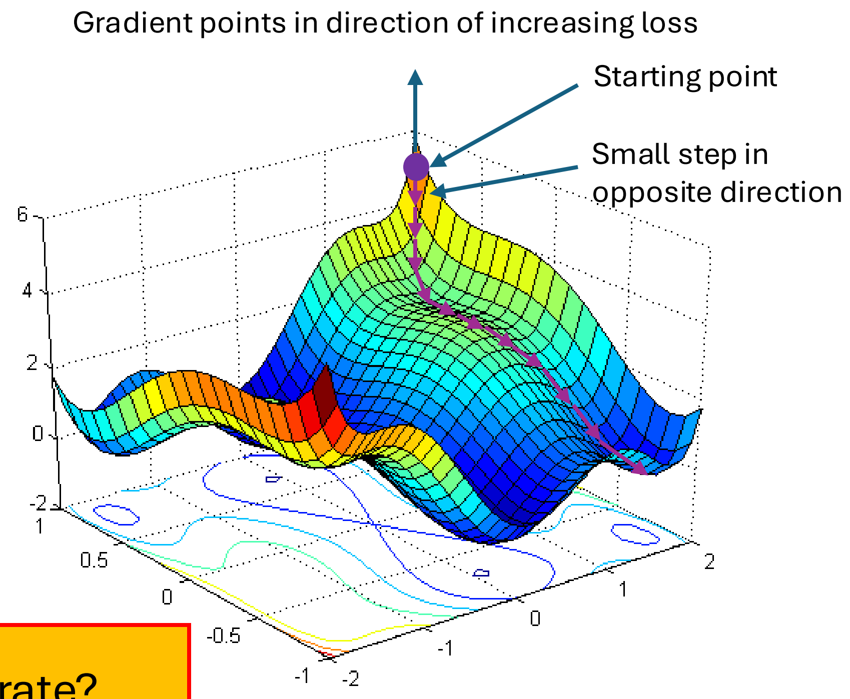
For N iterations or until $\Delta\theta < \epsilon$:

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

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 $\Delta\theta < \epsilon$:

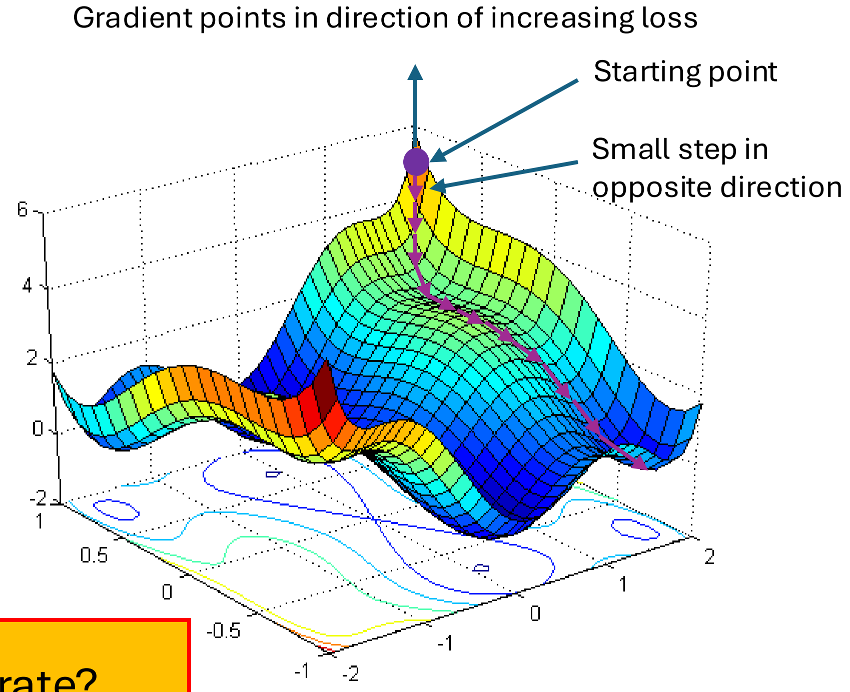
$$\vec{\theta} \leftarrow \theta - \alpha \nabla f_{\theta}$$

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 $\Delta\theta < \epsilon$:

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

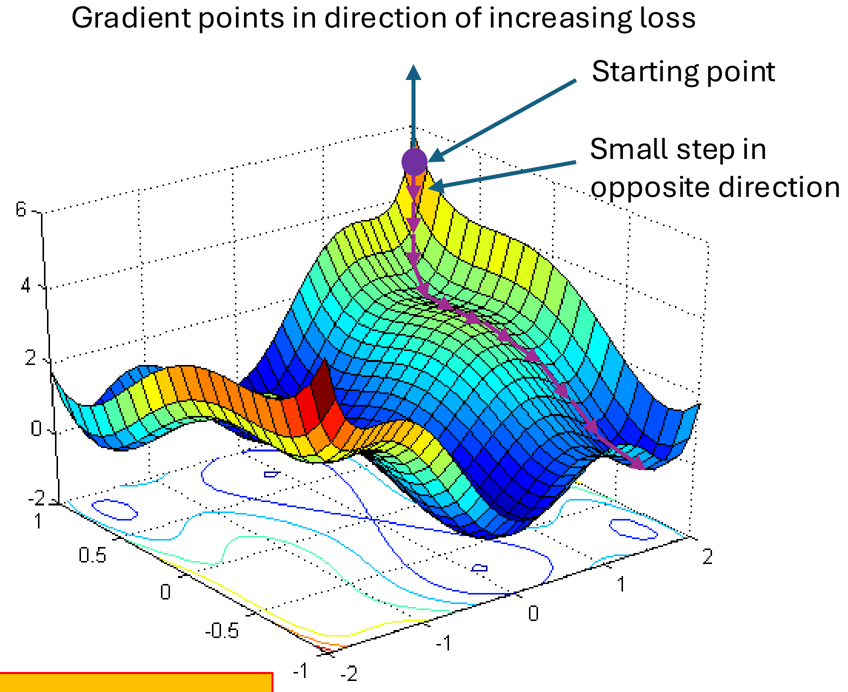
Learning Rate $\alpha \in [0,1]$

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.

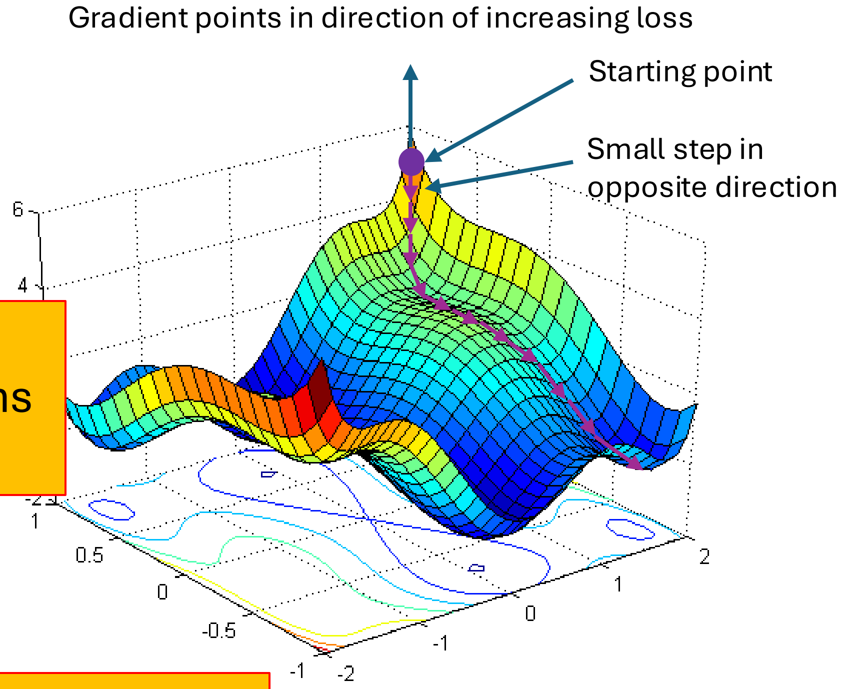
Option 2: Gradient Descent

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?

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

Gradients

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

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

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

Gradients

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

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

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

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

Gradients

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

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

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

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

But what is this?

Gradients

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

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

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

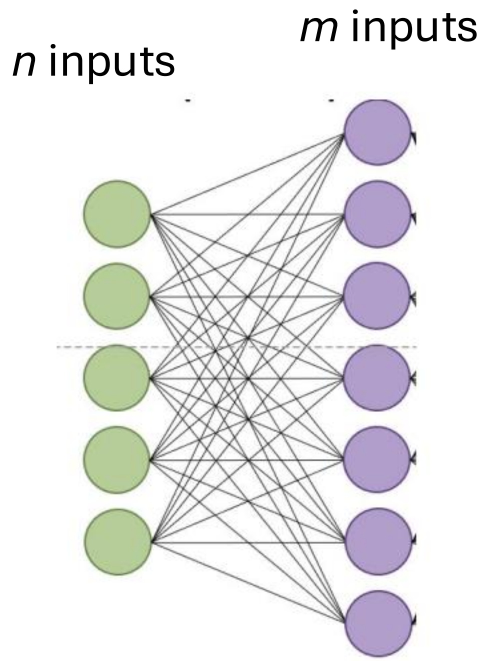
$$\nabla_\theta L = \frac{\sum_i^n 2 \cdot \nabla f_\theta(\vec{x}) + 2 \cdot \nabla f_\theta(\vec{x}) \cdot f_\theta(\vec{x})]}{n}$$

But what is this?

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

Weight Matrix

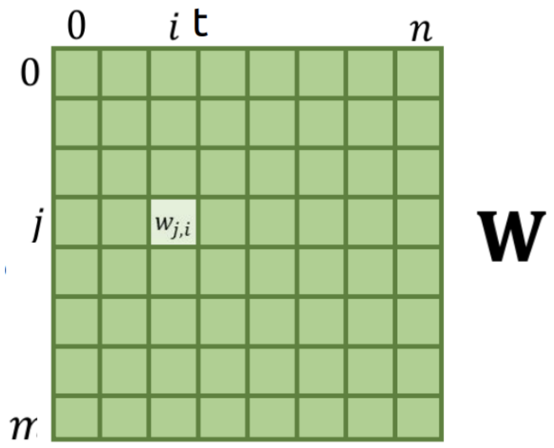
- We have an input of size *n* and we want an output vector of size *m*.
- We will represent our weights as a matrix.
  - What should the dimensions of our matrix be?

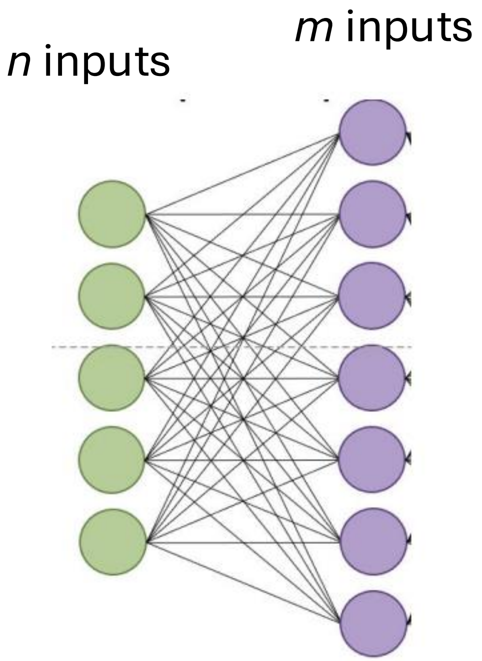


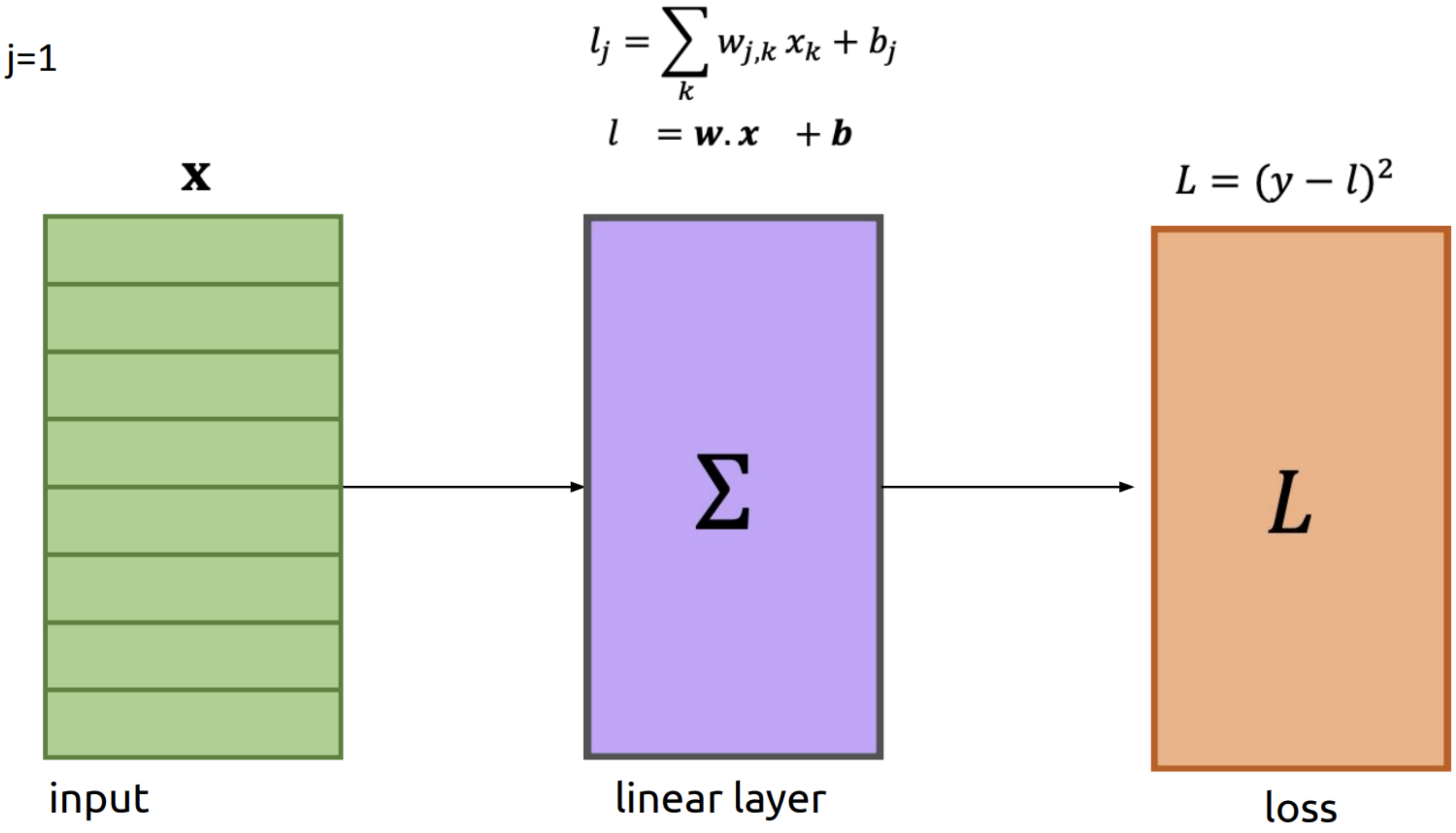
Weight Matrix

- 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?

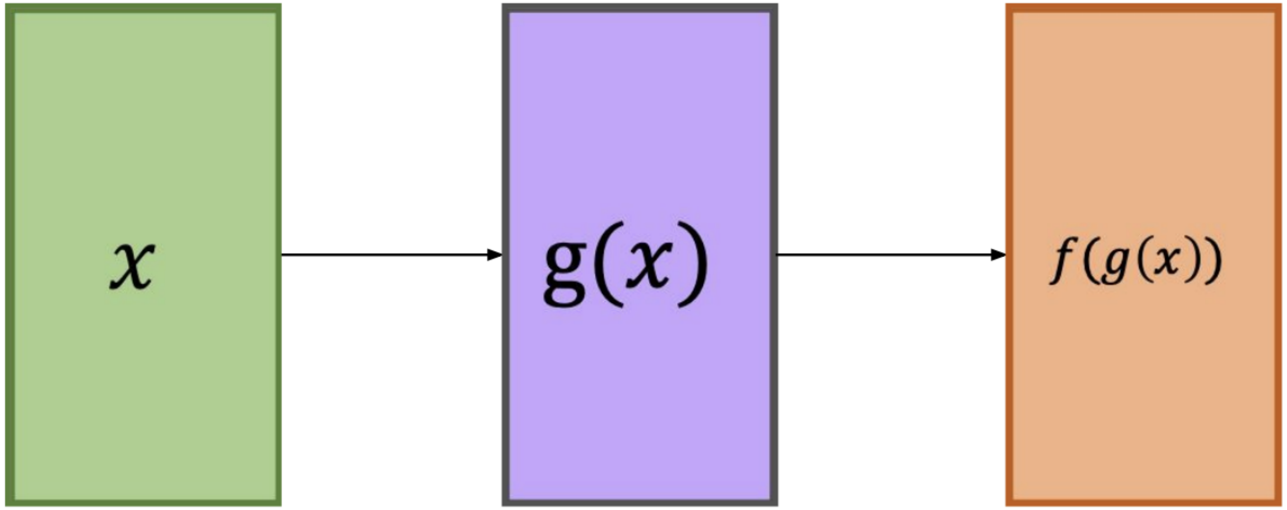
$w_{j,i}$ is the j^{th} row and the i^{th} column of our matrix, or the weight multiplied by the i^{th} index of the input which is used to create the j^{th} index in the output






j=1
$l_j = \sum_k w_{j,k} x_k + b_j$
$l = \boldsymbol{w}.\boldsymbol{x} + \boldsymbol{b}$
$\mathbf{x}$
$L = (y - l)^2$
Σ
L
input
linear layer
loss

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]

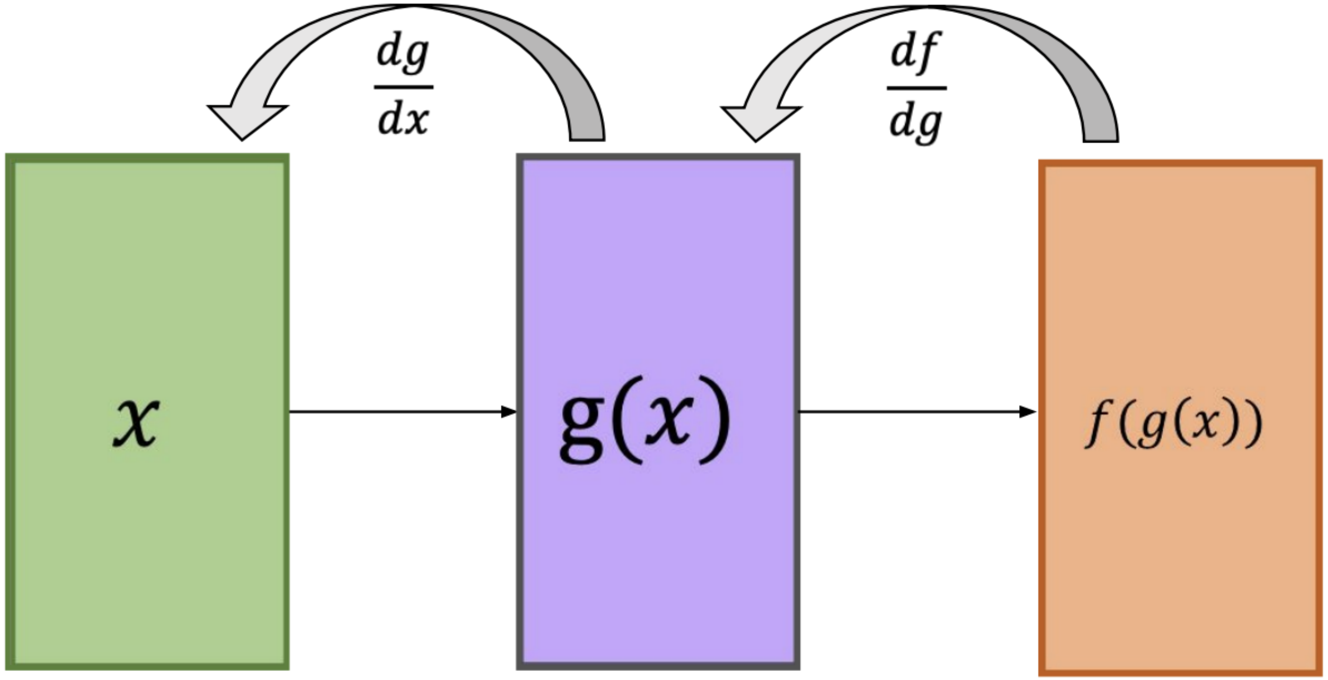
$$f(x) = x^2 \qquad \mathrm{g}(x) = (2x^2 + 1)$$

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

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

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 $\nabla \mathrm{L}_{\theta} = 0$

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)

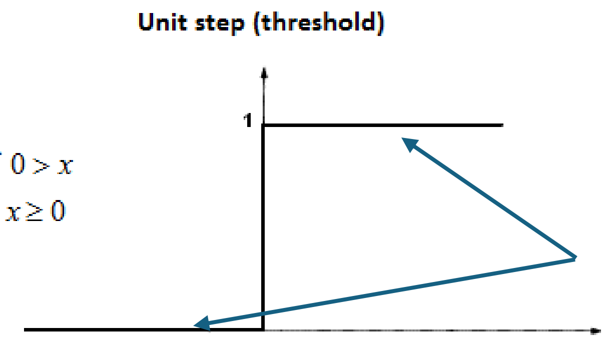
Need Loss function to be small for best model, not large.

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

$$f(x) = \begin{cases} 0 \text{ if } 0 > x \\ 1 \text{ if } x \geq 0 \end{cases}$$



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)

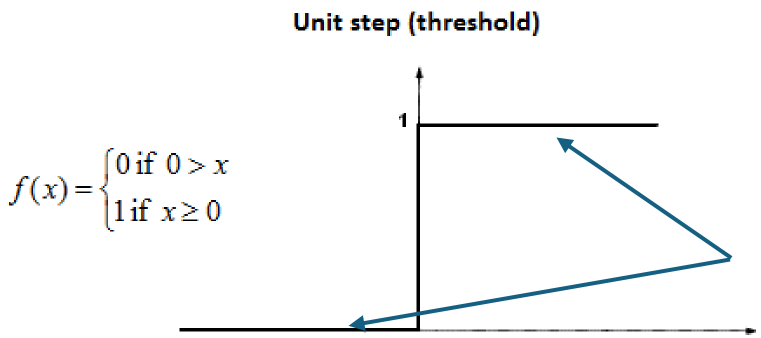
Need Loss function to be small for best model, not large.

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

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

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**



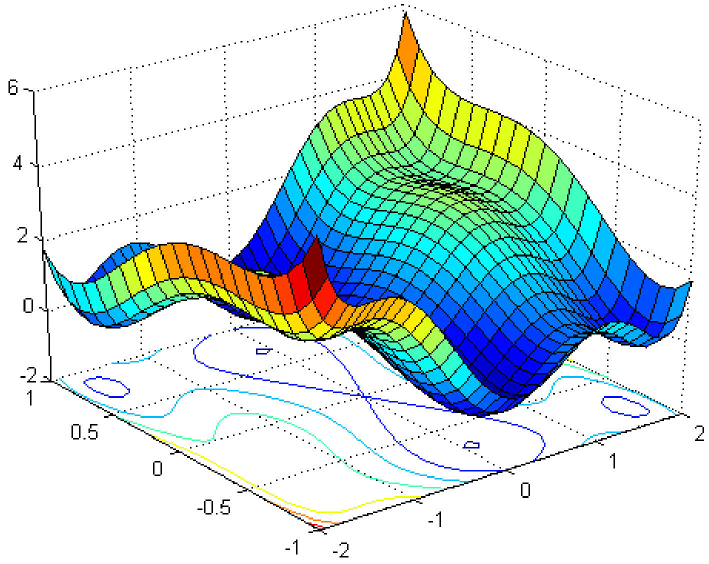
0 gradient everywhere except x=0

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.

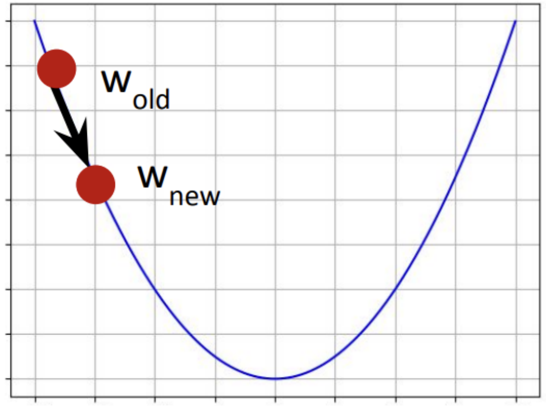
Can we do better?



Recap

Optimization

- Calculating gradients
- Gradient Descent for MSE Loss
- Convex and Non convex functions



Building a Neural Network

- Non-linear Activation Functions
- Hidden layers of neurons
- Loss Function
- Chain rule for backprop