CS-470: Machine Learning

Week 9 - Introduction to Neural Networks

Instructor: Dr. Sajjad Hussain

Department of Electrical and Computer Engineering
SEECS, NUST

November 13th, 2025

Overview

The Biological Neuron

- **Dendrites**: Input receptors
- **Nucleus**: Processing unit
- **Axon**: Output cable
- **Synapses**: Connections between neurons

Key Principle: Weighted sum and threshold mechanism

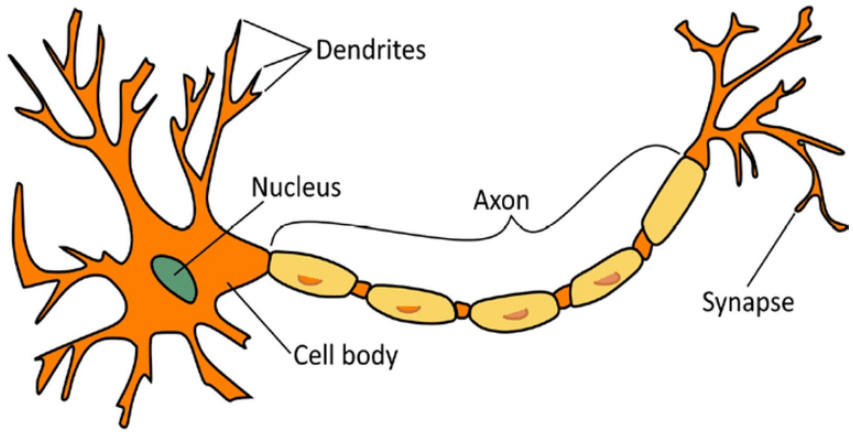
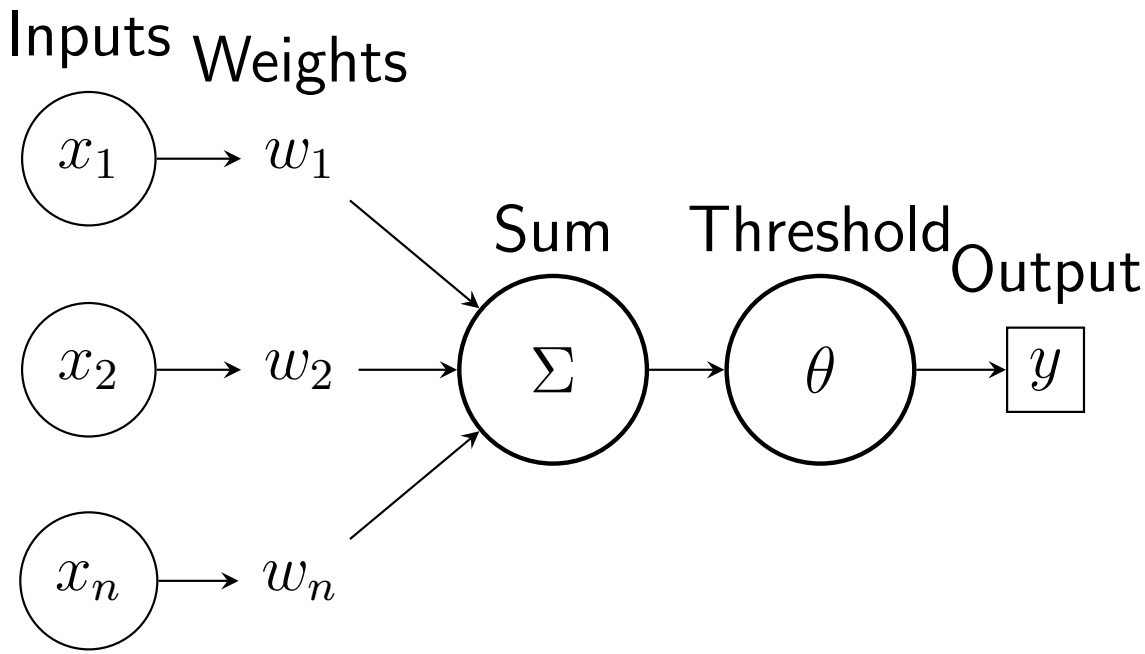


Figure: Biological neuron structure

McCulloch-Pitts Neuron: Detailed Structure

Key Characteristics

- No learning - fixed weights
- Synchronous operation
- Can compute logical functions

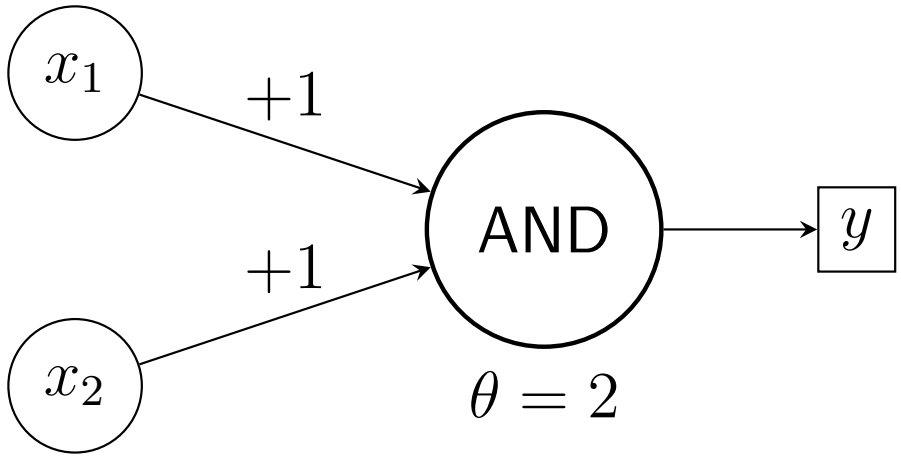


$$\text{Output} = \begin{cases} 1 & \text{if } \sum_{i=1}^{n} w_i x_i \geq \theta \\ 0 & \text{otherwise} \end{cases}$$

McCulloch-Pitts: AND Gate

AND Gate Truth Table

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1



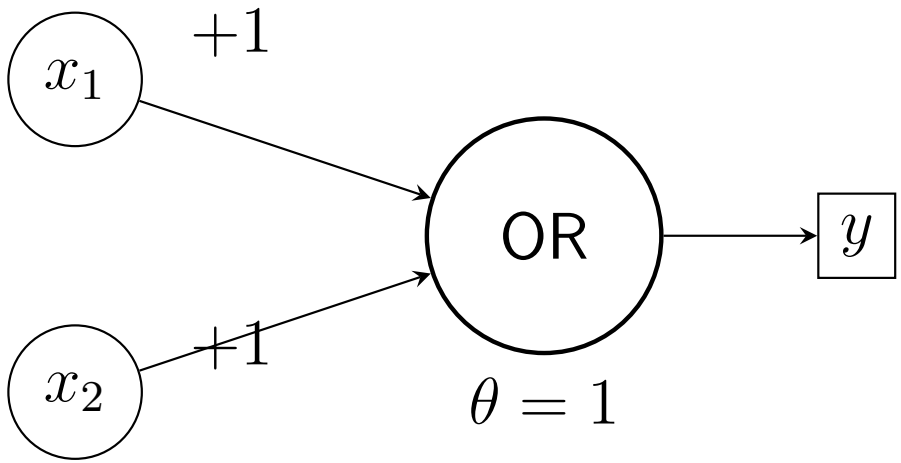
$$y = \begin{cases} 1 & \text{if } x_1 + x_2 \geq 2 \\ 0 & \text{otherwise} \end{cases}$$

$$\begin{aligned} x_1 = 1, x_2 = 1 &: 1 + 1 = 2 \geq 2 \rightarrow y = 1 \\ x_1 = 1, x_2 = 0 &: 1 + 0 = 1 < 2 \rightarrow y = 0 \\ x_1 = 0, x_2 = 1 &: 0 + 1 = 1 < 2 \rightarrow y = 0 \\ x_1 = 0, x_2 = 0 &: 0 + 0 = 0 < 2 \rightarrow y = 0 \end{aligned}$$

McCulloch-Pitts: OR Gate

OR Gate Truth Table

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1



$$y = \begin{cases} 1 & \text{if } x_1 + x_2 \geq 1 \\ 0 & \text{otherwise} \end{cases}$$

$$x_1 = 1, x_2 = 1 : 1 + 1 = 2 \geq 1 \rightarrow y = 1$$
$$x_1 = 1, x_2 = 0 : 1 + 0 = 1 \geq 1 \rightarrow y = 1$$
$$x_1 = 0, x_2 = 1 : 0 + 1 = 1 \geq 1 \rightarrow y = 1$$
$$x_1 = 0, x_2 = 0 : 0 + 0 = 0 < 1 \rightarrow y = 0$$

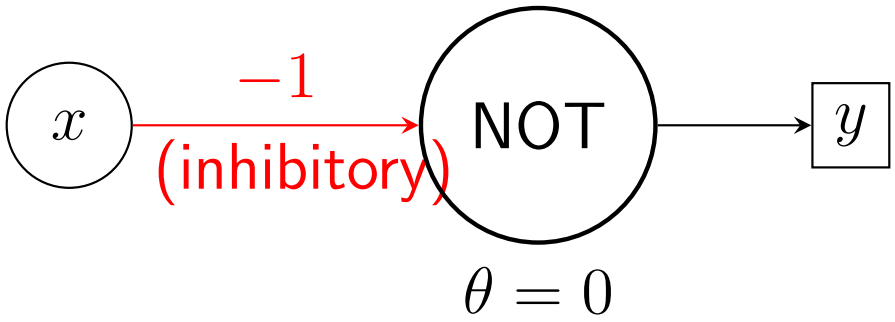
McCulloch-Pitts: NOT Gate

NOT Gate Truth Table

x	y
0	1
1	0

Parameters

- Weight: $w = -1$
- Threshold: $\theta = 0$
- Uses inhibitory connection



$$x = 1 : -1 < 0 \rightarrow y = 0$$
$$x = 0 : 0 \geq 0 \rightarrow y = 1$$

$$y = \begin{cases} 1 & \text{if } -x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

The Limitation: XOR Problem

XOR Gate Truth Table

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

Impossible with Single Neuron!

- No single line can separate the classes
- Requires **non-linear** separation
- Needs multiple layers
- No w_1, w_2, θ satisfy all conditions

Multi-Layer Solution for XOR

XOR = (x1 NAND x2) AND (x1 OR x2)

- Requires **two layers**
- Hidden layer computes intermediate functions
- Output layer combines them
- **Layer 1**: NAND + OR gates
- **Layer 2**: AND gate

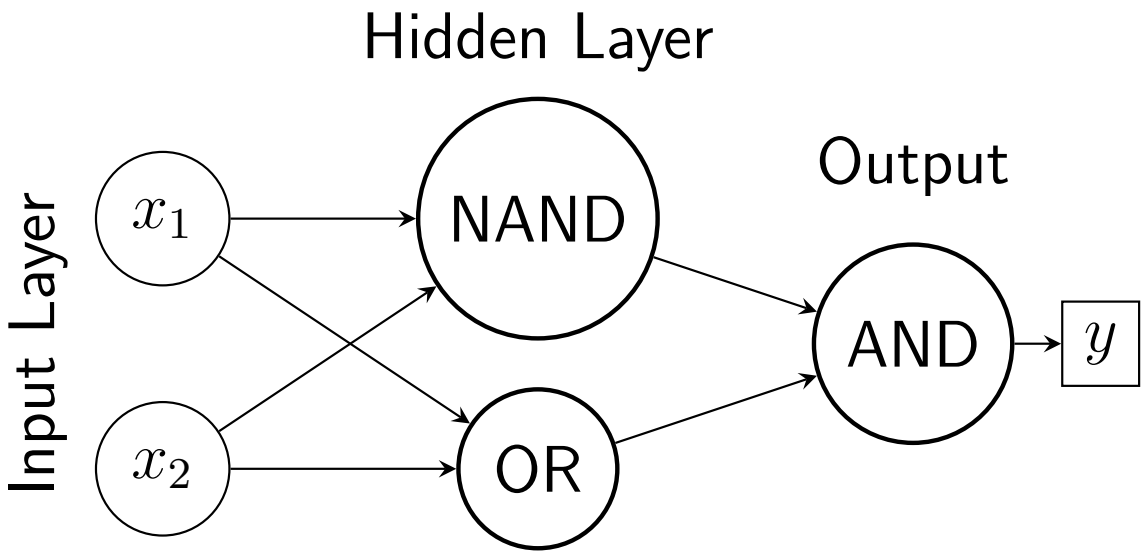


Figure: Multi-layer network solving XOR

This limitation led to the first AI winter, as Minsky & Papert proved single-layer perceptrons couldn't solve non-linearly separable problems like XOR.

The Perceptron (Frank Rosenblatt, 1958)

- First *learnable* artificial neuron
- Real-valued inputs and weights
- Includes bias term
- Step function activation

$$z = \sum_{i=1}^{n} w_i x_i + b$$

$$\text{output} = \text{step}(z)$$

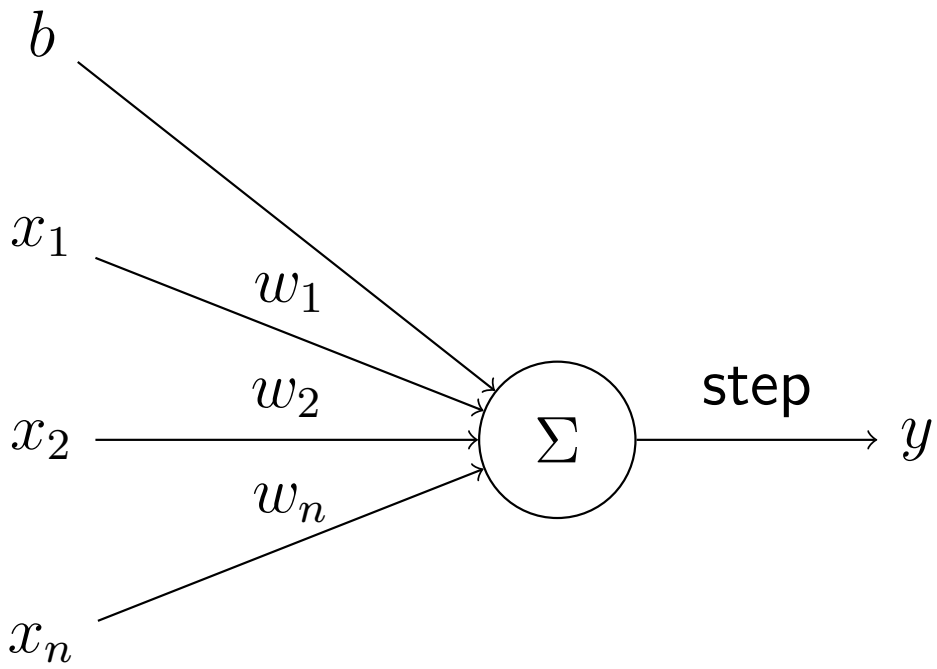


Figure: Perceptron diagram

Perceptron Learning Algorithm

Training Process

1. Initialize weights and bias (small random numbers)
2. For each training example (x, y_{true}):
   - Compute prediction: $y_{\text{pred}} = \text{step}(w \cdot x + b)$
   - Calculate error: $\text{error} = y_{\text{true}} - y_{\text{pred}}$
   - Update weights: $w_i^{\text{new}} = w_i^{\text{old}} + \eta \cdot \text{error} \cdot x_i$
   - Update bias: $b^{\text{new}} = b^{\text{old}} + \eta \cdot \text{error}$

Learning Rate η

Controls step size in weight updates. Typical values: 0.1, 0.01

What Can a Single Perceptron Do?

Linear Binary Classification

- AND gate
- OR gate
- NOT gate

Decision Boundary

$$w_1 x_1 + w_2 x_2 + b = 0$$

- Line in 2D
- Plane in 3D
- Hyperplane in n-D

The XOR Problem - A Fundamental Limitation

x_1	x_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Table: XOR truth table

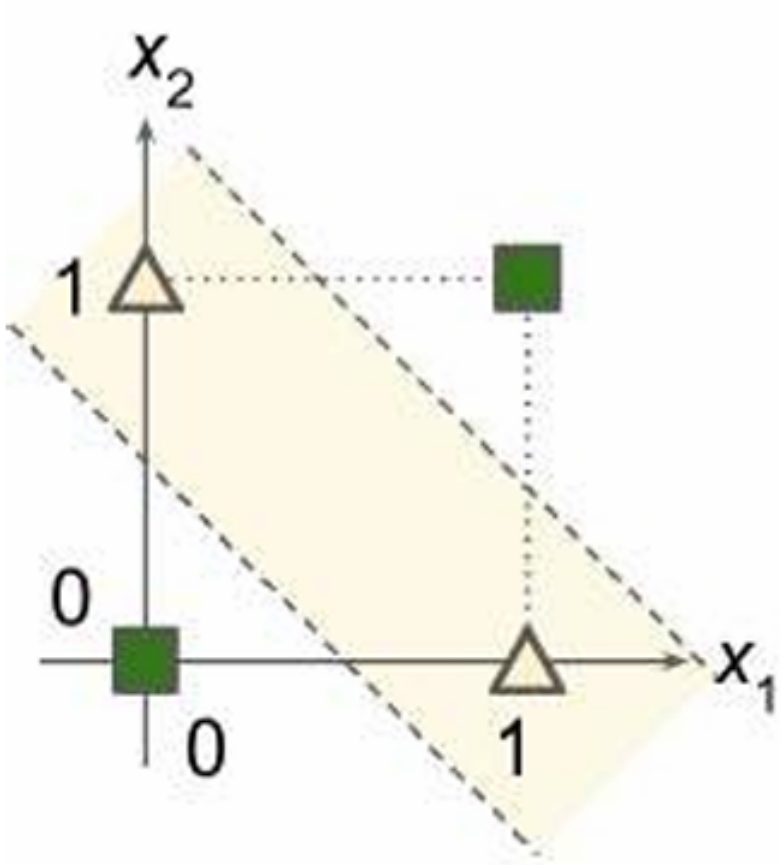


Figure: XOR is not linearly separable

Critical Insight

No single straight line can separate the classes!

The First AI Winter (1970s-1980s)

Minsky & Papert's "Perceptrons" (1969)

- Mathematical proof of Perceptron's limitations
- Cannot solve non-linearly separable problems
- Led to widespread pessimism

Consequences

- Drastic reduction in funding
- Shift to symbolic AI approaches
- Neural network research nearly abandoned

The Solution: Multi-Layer Perceptrons (MLPs)

- **Input Layer**: Receives data
- **Hidden Layer(s)**: Intermediate processing
- **Output Layer**: Final prediction

Key Insight: Stacking layers enables learning complex features

The Solution: Multi-Layer Perceptrons (MLPs)

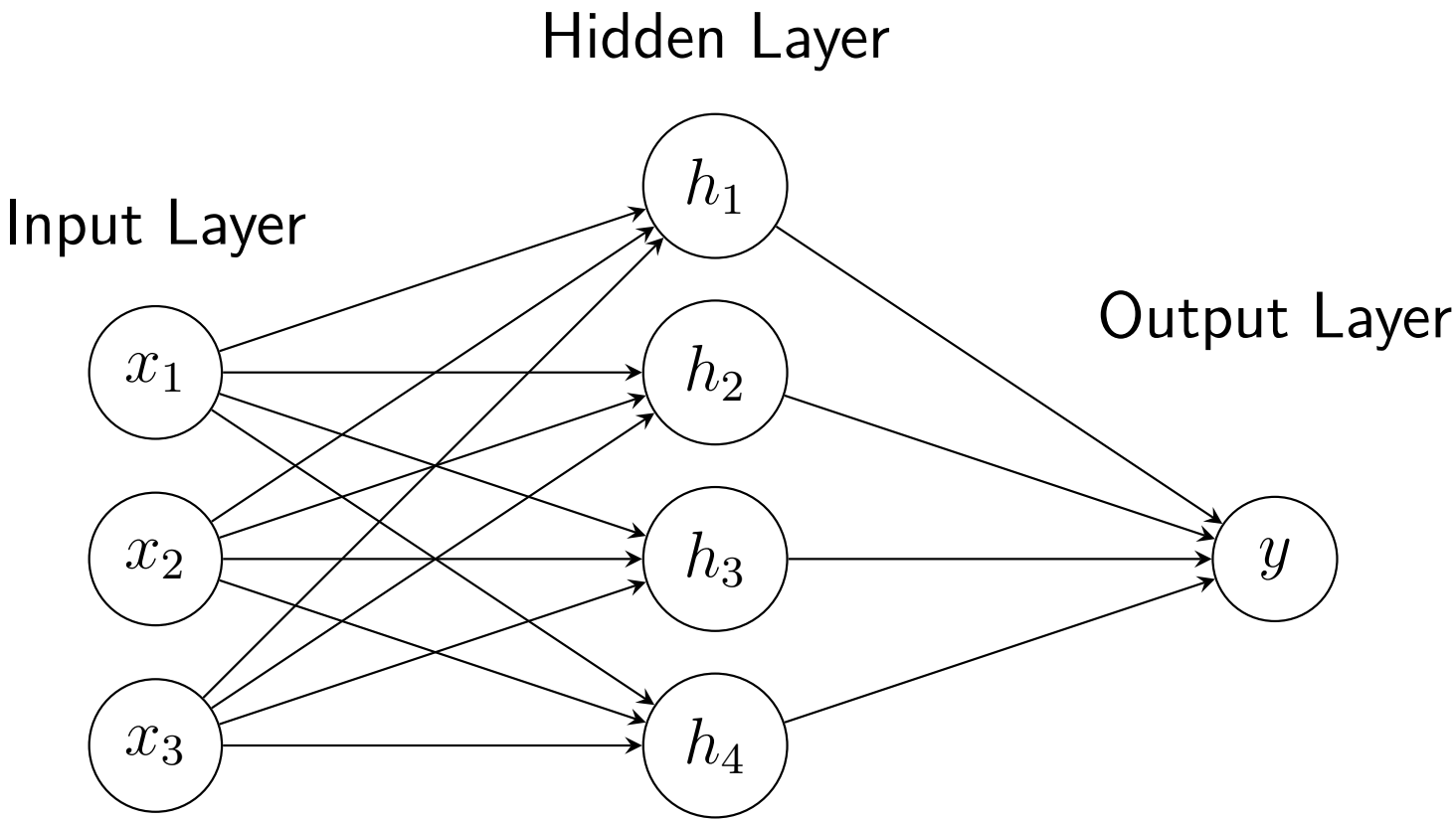


Figure: Multi-layer perceptron with one hidden layer

Multi-Layer Perceptron: Mathematical Operations

Network Architecture

- **Input Layer**: 3 neurons (x_1, x_2, x_3)
- **Hidden Layer**: 4 neurons (h_1, h_2, h_3, h_4)
- **Output Layer**: 1 neurons (y)
- **Activation**: Sigmoid for hidden layer

Notation

- $w_{ij}^{[l]}$: weight from neuron j in layer $l-1$ to neuron i in layer l
- $b_i^{[l]}$: bias for neuron i in layer l
- $z_i^{[l]}$: weighted sum for neuron i in layer l
- $a_i^{[l]}$: activated output for neuron i in layer l

Step 1: Input to Hidden Layer

Weighted Sum for Hidden Layer

For each hidden neuron h_j:

$$z_j^{[1]} = \sum_{i=1}^{3} w_{ji}^{[1]} x_i + b_j^{[1]}$$

Expanded Equations

$$z_1^{[1]} = w_{11}^{[1]} x_1 + w_{12}^{[1]} x_2 + w_{13}^{[1]} x_3 + b_1^{[1]}$$
$$z_2^{[1]} = w_{21}^{[1]} x_1 + w_{22}^{[1]} x_2 + w_{23}^{[1]} x_3 + b_2^{[1]}$$
$$z_3^{[1]} = w_{31}^{[1]} x_1 + w_{32}^{[1]} x_2 + w_{33}^{[1]} x_3 + b_3^{[1]}$$
$$z_4^{[1]} = w_{41}^{[1]} x_1 + w_{42}^{[1]} x_2 + w_{43}^{[1]} x_3 + b_4^{[1]}$$

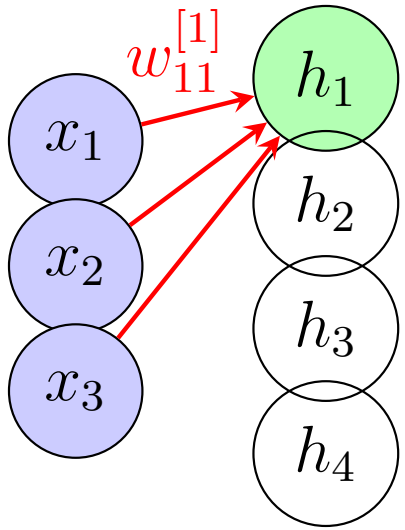


Figure: Calculating $z_1^{[1]}$

Step 2: Hidden Layer Activation

Activation Function

Apply an Activation function (such as ReLU) to each hidden neuron:

$$a_j^{[1]} = \text{ReLU}(z_j^{[1]}) = \max(0, z_j^{[1]})$$

Activated Hidden Layer Outputs

$$a_1^{[1]} = \max(0, z_1^{[1]})$$
$$a_2^{[1]} = \max(0, z_2^{[1]})$$
$$a_3^{[1]} = \max(0, z_3^{[1]})$$
$$a_4^{[1]} = \max(0, z_4^{[1]})$$

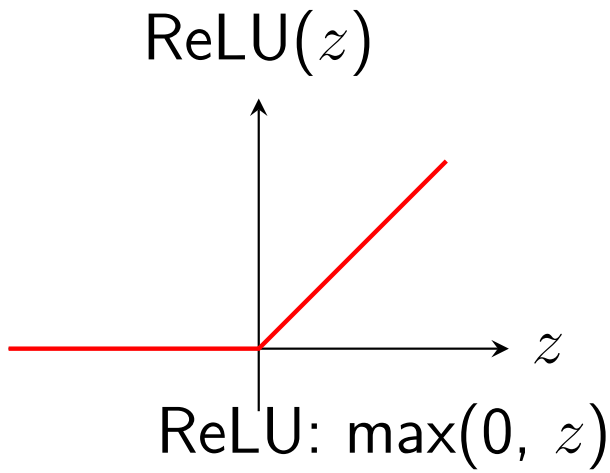


Figure: ReLU activation

Why ReLU?

- Prevents vanishing gradients (more on this later)
- Computationally efficient

Matrix Form

$$\mathbf{Z}^{[1]} = \mathbf{W}^{[1]}\mathbf{X} + \mathbf{b}^{[1]}$$

$$\mathbf{A}^{[1]} = \text{ReLU}(\mathbf{Z}^{[1]})$$

Why Activation Functions in Hidden Layers?

The Critical Question

What happens if we do not apply **any** activation function throughout the network?

Mathematical Insight

Layer 1: $\mathbf{a}^{[1]} = \mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]}$

Layer 2: $\mathbf{a}^{[2]} = \mathbf{W}^{[2]}\mathbf{a}^{[1]} + \mathbf{b}^{[2]}$

$$= \mathbf{W}^{[2]}(\mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]}) + \mathbf{b}^{[2]}$$

$$= \underbrace{\mathbf{W}^{[2]}\mathbf{W}^{[1]}}_{\mathbf{W}'}\mathbf{x} + \underbrace{\mathbf{W}^{[2]}\mathbf{b}^{[1]} + \mathbf{b}^{[2]}}_{\mathbf{b}'}$$

Key Realization

- **Any deep linear network** collapses to a **single linear layer**
- **No additional power** over simple linear regression
- **Cannot learn** non-linear relationships

The Power of Non-Linearity

Activation Functions Enable Deep Learning

- Introduce **non-linear transformations** between layers.
- Allow networks to learn **complex feature hierarchies**. For Example,
  - **Layer 1**: Simple features (edges, corners)
  - **Layer 2**: Complex features (shapes, patterns)
  - **Layer 3**: Object parts (eyes, wheels)
  - **Layer 4**: Complete objects (faces, cars)

Common Hidden Layer Activation Functions

1. Sigmoid

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

2. Tanh

$$f(z) = \tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

3. ReLU

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

Visual Comparison of Activation Functions

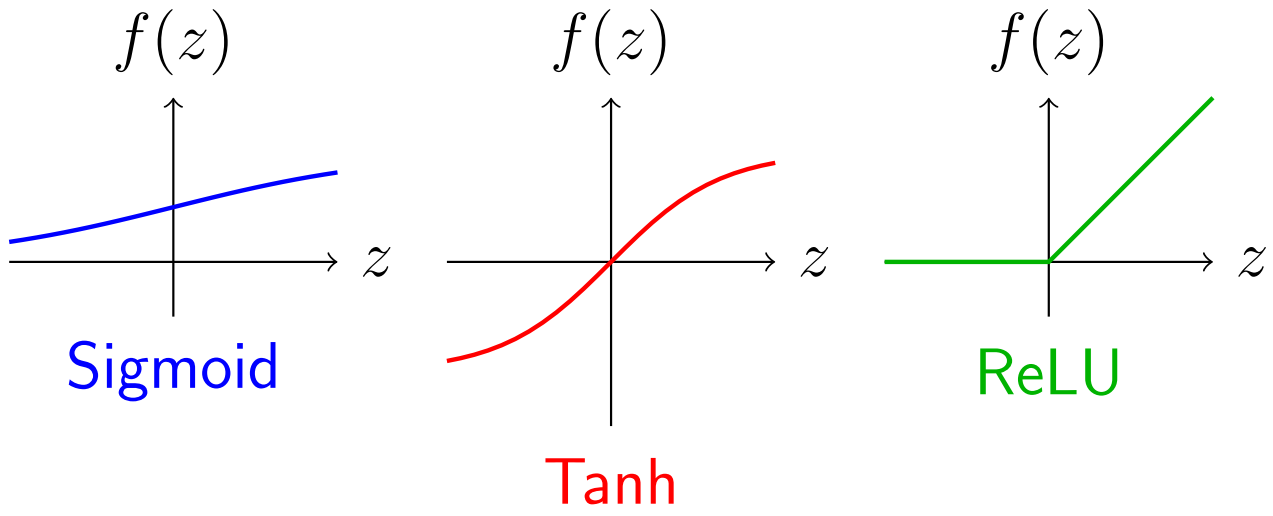


Figure: Comparison of basic activation functions

Key Properties

- **Sigmoid**: Always positive, bounded (0,1)
- **Tanh**: Zero-centered, bounded (-1,1)
- **ReLU**: Simple, computationally efficient, unbounded

Step 3: Hidden to Output Layer

Weighted Sum for Output Layer

For the output neuron y in our example:

$$z_1^{[2]} = \sum_{j=1}^{4} w_{kj}^{[2]} a_j^{[1]} + b_k^{[2]}$$

Expanded Equations

$$z_1^{[2]} = w_{11}^{[2]} a_1^{[1]} + w_{12}^{[2]} a_2^{[1]} + w_{13}^{[2]} a_3^{[1]} + w_{14}^{[2]} a_4^{[1]} + b_1^{[2]}$$

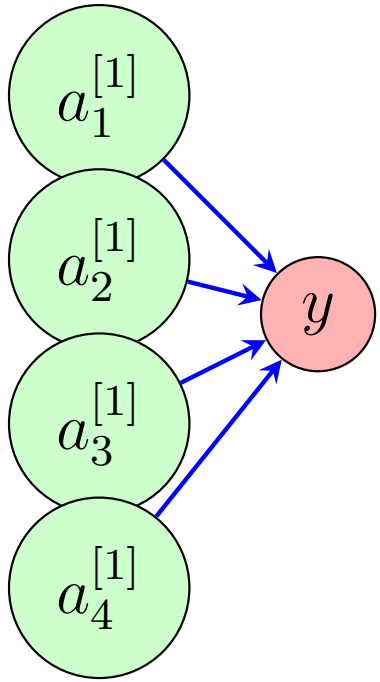


Figure: Calculating $z_1^{[2]}$

Output Layer Activation: Application Specific

The Right Choice Depends on Your Problem

- Different applications require different output interpretations
- The activation function shapes how we understand the network's predictions
- Must match the problem type and desired output format

Common Output Patterns

- **Real numbers** $\rightarrow$ Linear activation (Regression)
- **Probabilities** $\rightarrow$ Sigmoid/Softmax (Classification)
- **Positive values** $\rightarrow$ ReLU (Positive regression)
- **Multiple choices** $\rightarrow$ Softmax (Multi-class)

1. Linear/No Activation: For Regression

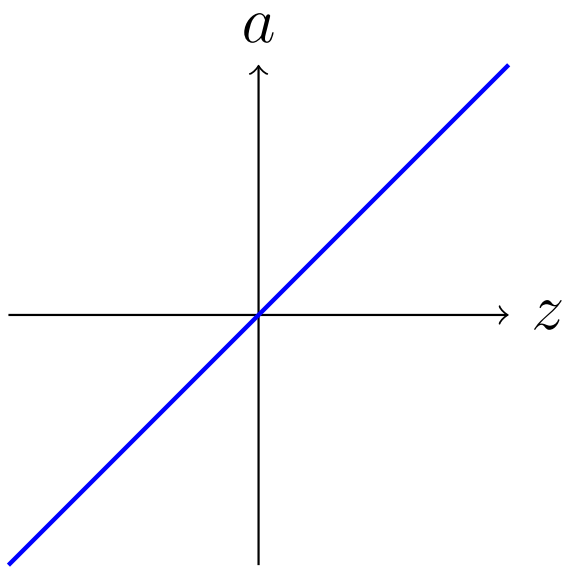
Mathematical Form

$$a = z$$

- No transformation applied
- Output: any real number
- Range: $(-\infty, +\infty)$

Applications

- House price prediction
- Stock price forecasting
- Temperature prediction



$a = z$

Figure: Linear activation preserves values

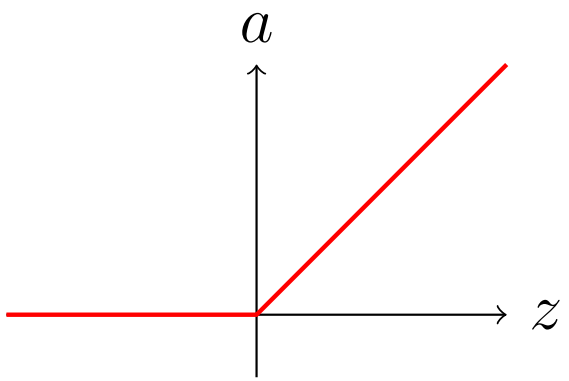
When to Use

2. ReLU Activation: Positive-Valued Regression

Mathematical Form

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

- Output: non-negative numbers
- Range: $[0, +\infty)$
- Zeros out negative values



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

Applications

- Age prediction
- Salary estimation
- Distance measurement

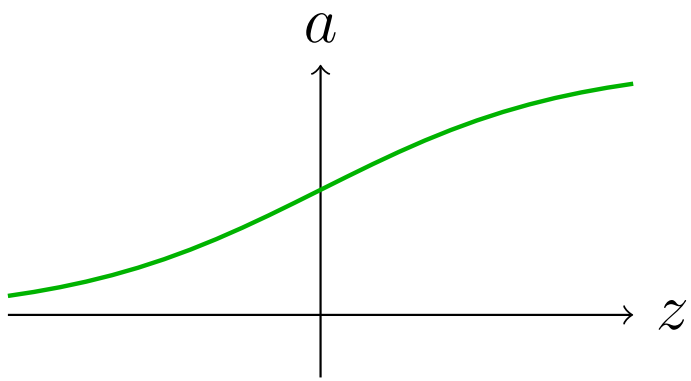
Figure: ReLU produces non-negative outputs

3. Sigmoid Activation: Binary Classification

Mathematical Form

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

- Output range: (0, 1)
- Interpret as probability
- Smooth S-shaped curve



Sigmoid: $0 < a < 1$

Applications

- Spam detection
- Medical diagnosis
- Sentiment analysis
- Any yes/no decision

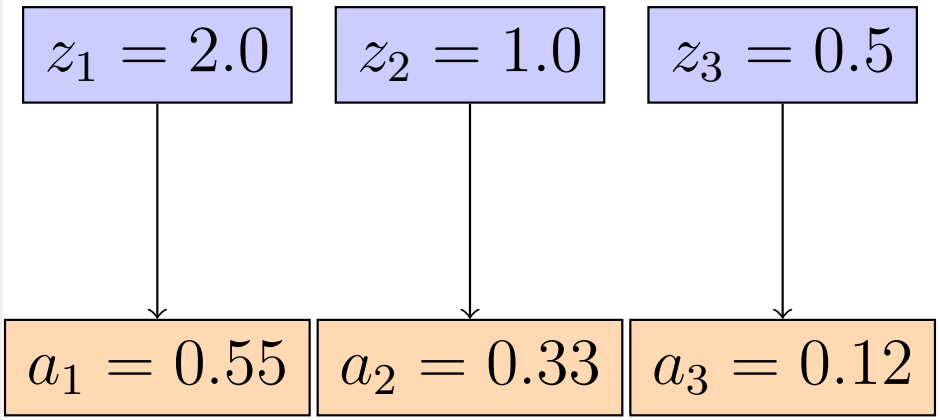
Figure: Sigmoid produces probabilities

4. Softmax Activation: Multi-Class Classification

Mathematical Form

$$a_k = \frac{e^{z_k}}{\sum_{j=1}^{C} e^{z_j}}$$

- Outputs sum to 1
- Probability distribution
- Each output: $0 < a_k < 1$



$$0.55 + 0.33 + 0.12 = 1.00$$

Probability Distribution

Applications

- Digit recognition (0-9)
- Object classification (cat/dog/car)
- Topic classification

Figure: Softmax creates probability distribution

Complete Forward Pass Summary

Step-by-Step Computation

1. **Input to Hidden**:

$$z_j^{[1]} = \sum_{i=1}^{3} w_{ji}^{[1]} x_i + b_j^{[1]} \quad \text{for } j = 1, \ldots, 4$$

2. **Hidden Activation**:

$$a_j^{[1]} = \text{ReLU}(z_j^{[1]}) \quad \text{for } j = 1, \ldots, 4$$

3. **Hidden to Output**:

$$z_k^{[2]} = \sum_{j=1}^{4} w_{kj}^{[2]} a_j^{[1]} + b_k^{[2]} \quad \text{for } k = 1, 2$$

Matrix Form: Efficient Computation

Compact Representation

$$\mathbf{Z}^{[1]} = \mathbf{W}^{[1]}\mathbf{X} + \mathbf{b}^{[1]} \qquad \mathbf{W}^{[1]} \in \mathbb{R}^{4\times 3}, \mathbf{X} \in \mathbb{R}^{3\times m}$$

$$\mathbf{A}^{[1]} = \text{ReLU}(\mathbf{Z}^{[1]}) \qquad \mathbf{A}^{[1]} \in \mathbb{R}^{4\times m}$$

$$\mathbf{Z}^{[2]} = \mathbf{W}^{[2]}\mathbf{A}^{[1]} + \mathbf{b}^{[2]} \qquad \mathbf{W}^{[2]} \in \mathbb{R}^{2\times 4}$$

$$\mathbf{A}^{[2]} = \mathbf{f}(\mathbf{Z}^{[2]}) \qquad \mathbf{A}^{[2]} \in \mathbb{R}^{1\times m}$$

Advantages

- Parallel computation on GPUs
- Efficient batch processing (batch size $= m$ in above representation)
- Clean mathematical formulation

Backpropagation: How Neural Networks Learn

The Learning Process

- **Forward Pass**: Compute predictions from input to output
- **Loss Calculation**: Measure how wrong our predictions are
- **Backward Pass**: Calculate gradients using chain rule
- **Weight Update**: Adjust weights to reduce loss

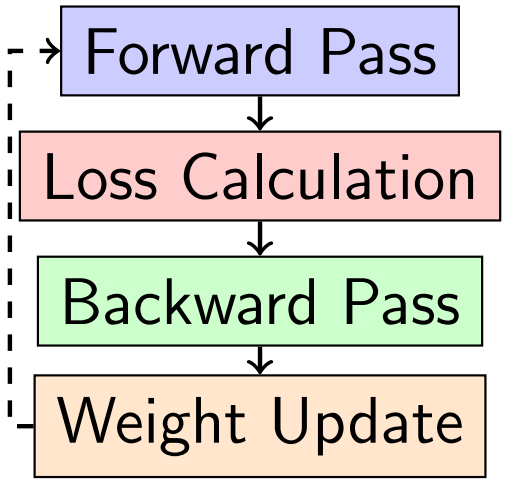


Figure: Training loop - Repeat until convergence

Backpropagation: Key Insights

The Fundamental Question

How much does each weight contribute to the total error?

Chain Rule from Calculus

- Break complex derivatives into simple steps
- Compute gradients **efficiently**
- Work **backward** from output to input

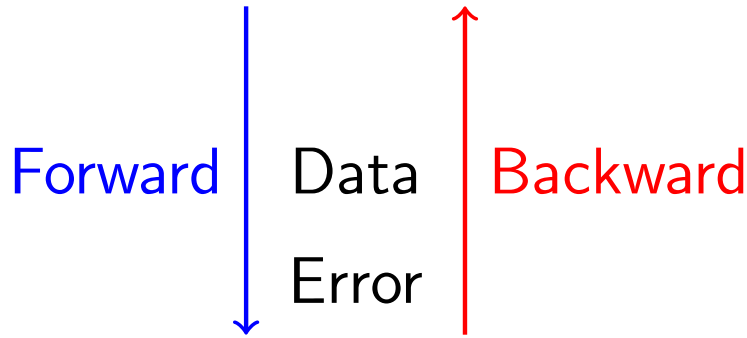


Figure: Forward vs Backward pass

Example Network Architecture

Network Specifications

- **Input**: 2 neurons (x_1, x_2)
- **Hidden Layer 1**: 2 neurons, **Hidden Layer 2**: 2 neurons
- **Output**: 1 neuron $(\hat{y})$
- **Activation**: ReLU for hidden layers
- **Output**: Linear (for regression)
- **Loss**: Mean Squared Error

Notation Guide

- $w_{ij}^{[l]}$: weight from neuron j in layer $l-1$ to neuron i in layer l
- $z^{[l]}$: weighted sum before activation
- $a^{[l]}$: activated output

Network Architecture Diagram

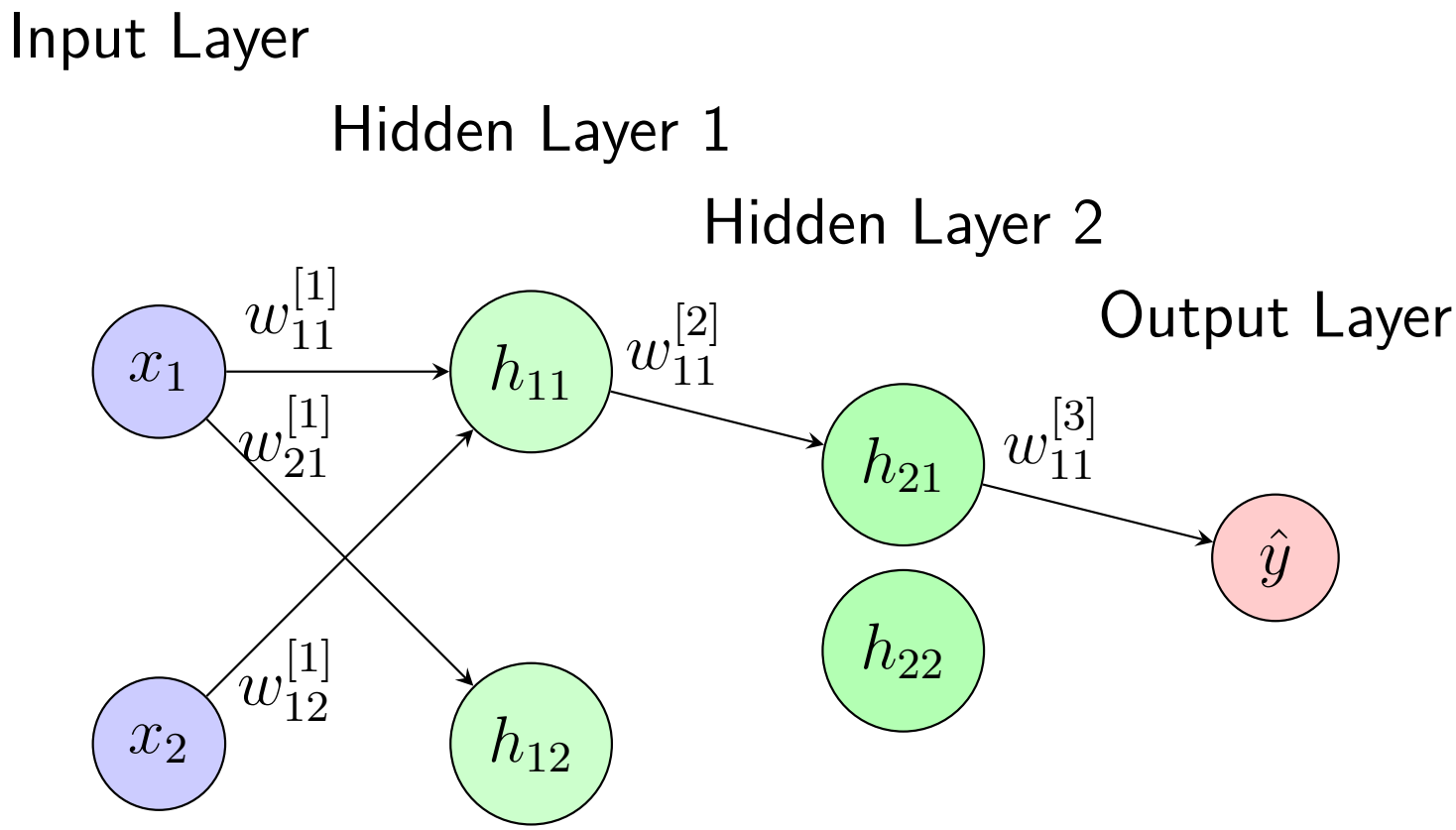


Figure: 2-2-2-1 Network Architecture

Focus Weight

Step 1: Forward Pass - Layer 1

Layer 1 Calculations

For each neuron in Hidden Layer 1:

$$z_1^{[1]} = w_{11}^{[1]} x_1 + w_{12}^{[1]} x_2 + b_1^{[1]}$$
$$a_1^{[1]} = \text{ReLU}(z_1^{[1]}) = \max(0, z_1^{[1]})$$

$$z_2^{[1]} = w_{21}^{[1]} x_1 + w_{22}^{[1]} x_2 + b_2^{[1]}$$
$$a_2^{[1]} = \text{ReLU}(z_2^{[1]})$$

ReLU Activation

$$\text{ReLU}(z) = \max(0, z) = \begin{cases} z & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases}$$

Step 1: Forward Pass - Layer 2 & Output

Layer 2 Calculations

$$z_1^{[2]} = w_{11}^{[2]} a_1^{[1]} + w_{12}^{[2]} a_2^{[1]} + b_1^{[2]}$$
$$a_1^{[2]} = \text{ReLU}(z_1^{[2]})$$

$$z_2^{[2]} = w_{21}^{[2]} a_1^{[1]} + w_{22}^{[2]} a_2^{[1]} + b_2^{[2]}$$
$$a_2^{[2]} = \text{ReLU}(z_2^{[2]})$$

Output Layer

$$z^{[3]} = w_{11}^{[3]} a_1^{[2]} + w_{12}^{[3]} a_2^{[2]} + b^{[3]}$$
$$\hat{y} = z^{[3]} \quad \text{(Linear activation for regression)}$$

Step 2: Loss Calculation

Mean Squared Error Loss

$$L = \frac{1}{2}(\hat{y} - y)^2$$

Where:

- $\hat{y}$: Network prediction
- y: True target value
- Factor $\frac{1}{2}$ simplifies derivative calculation

Example Values

- Let's say: $\hat{y} = 2.5$, $y = 1.0$
- Then: $L = \frac{1}{2}(2.5 - 1.0)^2 = \frac{1}{2}(1.5)^2 = 1.125$

Loss Function Visualization

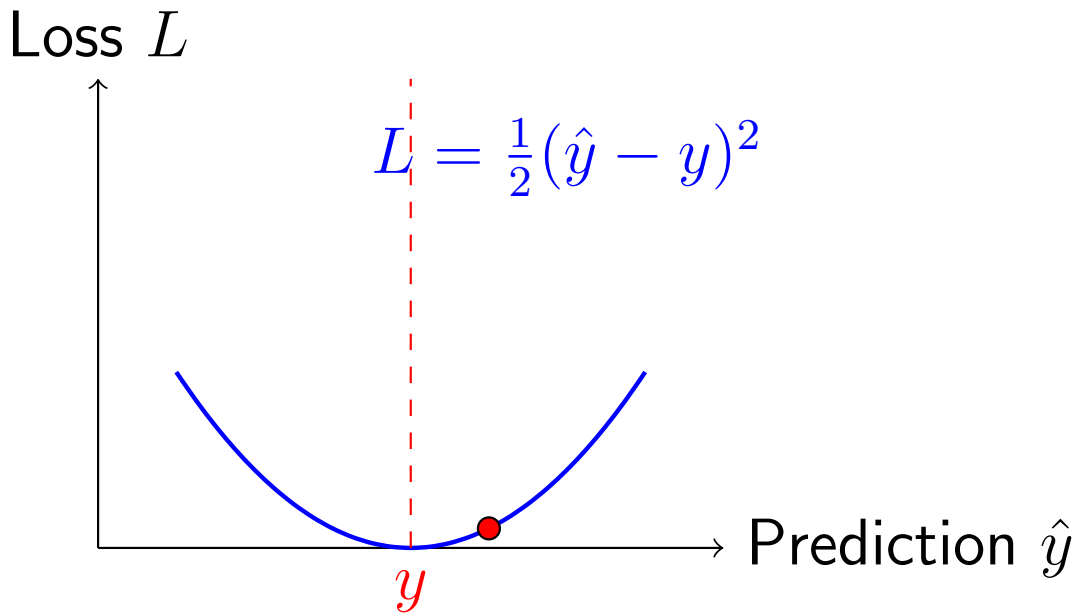


Figure: Loss function landscape

Our Goal

Find the direction to move $\hat{y}$ toward y to minimize loss

Step 3: Backward Pass - Output Layer

Gradient for Output Layer Weights

We want: $\frac{\partial L}{\partial w_{11}^{[3]}}$ and $\frac{\partial L}{\partial w_{12}^{[3]}}$

Using chain rule:

$$\frac{\partial L}{\partial w_{11}^{[3]}} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z^{[3]}} \cdot \frac{\partial z^{[3]}}{\partial w_{11}^{[3]}}$$

$$\frac{\partial L}{\partial w_{12}^{[3]}} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z^{[3]}} \cdot \frac{\partial z^{[3]}}{\partial w_{21}^{[3]}}$$

Backward Pass - Output Layer Computation

Compute Each Term

$$\frac{\partial L}{\partial \hat{y}} = \hat{y} - y, \qquad \frac{\partial \hat{y}}{\partial z^{[3]}} = 1 \quad \text{(linear activation)}$$

$$\frac{\partial z^{[3]}}{\partial w_{11}^{[3]}} = a_1^{[2]}, \qquad \frac{\partial z^{[3]}}{\partial w_{21}^{[3]}} = a_2^{[2]}$$

Final Gradients

$$\frac{\partial L}{\partial w_{11}^{[3]}} = (\hat{y} - y) \cdot a_1^{[2]}$$

$$\frac{\partial L}{\partial w_{12}^{[3]}} = (\hat{y} - y) \cdot a_2^{[2]}$$

Step 4: Backward Pass - Hidden Layer 2

Focus on $w_{11}^{[2]}$

We want: $\frac{\partial L}{\partial w_{11}^{[2]}}$

Chain rule through multiple layers:

$$\frac{\partial L}{\partial w_{11}^{[2]}} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z^{[3]}} \cdot \frac{\partial z^{[3]}}{\partial a_1^{[2]}} \cdot \frac{\partial a_1^{[2]}}{\partial z_1^{[2]}} \cdot \frac{\partial z_1^{[2]}}{\partial w_{11}^{[2]}}$$

Compute Each Term

$$\frac{\partial L}{\partial \hat{y}} = \hat{y} - y, \qquad \frac{\partial \hat{y}}{\partial z^{[3]}} = 1$$

$$\frac{\partial z^{[3]}}{\partial a_1^{[2]}} = w_{11}^{[3]}, \qquad \frac{\partial z_1^{[2]}}{\partial w_{11}^{[2]}} = a_1^{[1]}$$

Backward Pass - Hidden Layer 2 Completion

ReLU Derivative

$$\frac{\partial a_1^{[2]}}{\partial z_1^{[2]}} = \text{ReLU}'(z_1^{[2]}) = \begin{cases} 1 & \text{if } z_1^{[2]} > 0 \\ 0 & \text{if } z_1^{[2]} \leq 0 \end{cases}$$

Final Gradient for $w_{11}^{[2]}$

$$\frac{\partial L}{\partial w_{11}^{[2]}} = (\hat{y} - y) \cdot w_{11}^{[3]} \cdot \text{ReLU}'(z_1^{[2]}) \cdot a_1^{[1]}$$

Pattern Emerging

- Each term represents contribution from a layer
- ReLU derivative acts as a gate: 1 or 0
- We see the beginning of chain rule multiplication

Step 5: Backward Pass - Hidden Layer 1

Focus on $w_{11}^{[1]}$ (Our Target Weight)

We want: $\frac{\partial L}{\partial w_{11}^{[1]}}$

Chain rule through all layers:

$$\frac{\partial L}{\partial w_{11}^{[1]}} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z^{[3]}} \cdot \frac{\partial z^{[3]}}{\partial a_1^{[2]}} \cdot \frac{\partial a_1^{[2]}}{\partial z_1^{[2]}} \cdot \frac{\partial z_1^{[2]}}{\partial a_1^{[1]}} \cdot \frac{\partial a_1^{[1]}}{\partial z_1^{[1]}} \cdot \frac{\partial z_1^{[1]}}{\partial w_{11}^{[1]}}$$

Long Chain!

This is why we need backpropagation - to compute this efficiently!

Backward Pass - Hidden Layer 1 Computation

Compute Each Term

$$\frac{\partial L}{\partial \hat{y}} = \hat{y} - y$$

$$\frac{\partial \hat{y}}{\partial z^{[3]}} = 1$$

$$\frac{\partial z^{[3]}}{\partial a_1^{[2]}} = w_{11}^{[3]}$$

$$\frac{\partial a_1^{[2]}}{\partial z_1^{[2]}} = \mathsf{ReLU}'(z_1^{[2]})$$

$$\frac{\partial z_1^{[2]}}{\partial a_1^{[1]}} = w_{11}^{[2]}$$

$$\frac{\partial a_1^{[1]}}{}$$

Backward Pass - Hidden Layer 1 Final

Final Gradient for $w_{11}^{[1]}$

$$\frac{\partial L}{\partial w_{11}^{[1]}} = (\hat{y} - y) \cdot w_{11}^{[3]} \cdot \text{ReLU}'(z_1^{[2]}) \cdot w_{11}^{[2]} \cdot \text{ReLU}'(z_1^{[1]}) \cdot x_1$$

Complete Pattern

Each layer adds:

- Weight from next layer
- Activation derivative from current layer
- Input from previous layer at the end

General Formula

For any weight $w_{ij}^{[l]}$:

Chain Rule Visualization

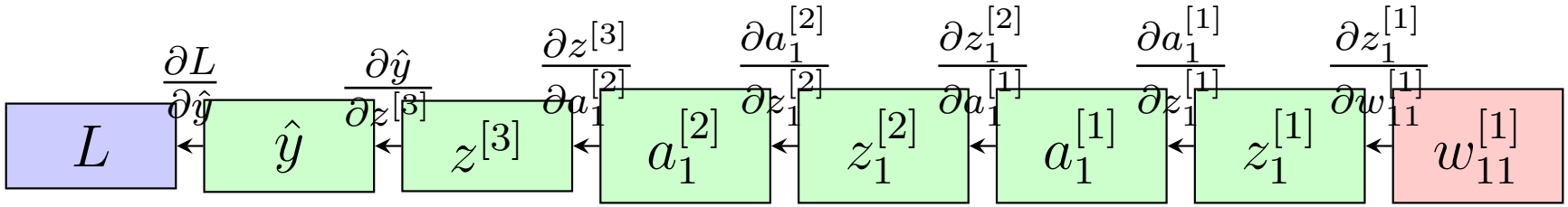


Figure: Chain rule path for $\frac{\partial L}{\partial w_{11}^{[1]}}$

Backpropagation Efficiency

- Compute gradients **backward** to reuse calculations
- Store intermediate results (δ terms)
- Avoid recomputing the same derivatives

Step 6: Weight Update with Gradient Descent

Gradient Descent Update Rule

$$w_{\text{new}} = w_{\text{old}} - \eta \cdot \frac{\partial L}{\partial w}$$

Where:

- η: Learning rate (step size)
- $\frac{\partial L}{\partial w}$: Gradient we computed via backpropagation

Update for Our Focus Weight

$$w_{11}^{[1]} \leftarrow w_{11}^{[1]} - \eta \cdot \frac{\partial L}{\partial w_{11}^{[1]}}$$

Simultaneous Update

All weights are updated **simultaneously** after computing all gradients

Learning Rate Importance

Learning Rate η

Controls step size in weight updates

Good Learning Rate

- Smooth convergence
- Reaches minimum efficiently
- Stable training

Common Values

- 0.1, 0.01, 0.001
- Often decreased over time

Complete Training Cycle Summary

One Training Iteration

1. **Forward Pass**: Compute $\hat{y}$ from inputs x_1, x_2
2. **Loss**: Calculate $L = \frac{1}{2}(\hat{y} - y)^2$
3. **Backward Pass**:
   - Compute $\frac{\partial L}{\partial w}$ for all weights
   - Use chain rule from output back to input
   - Store intermediate results efficiently
4. **Update**: Adjust all weights using gradient descent

Repeat Until Convergence

Continue until loss stops decreasing or reaches acceptable level

Key Insights and Takeaways

Why Backpropagation Works

- **Efficiency**: Computes all gradients in one backward pass
- **Modularity**: Each layer's computation is independent
- **Generality**: Works for any differentiable network
- **Scalability**: Same principle for networks of any size

The Big Picture

Gradients tell us the direction to decrease loss
Gradient descent moves weights in that direction
Backpropagation computes gradients efficiently

Foundation of Deep Learning

This same principle powers all modern deep learning systems!

The Second AI Winter (Late 1990s - Early 2000s)

Challenges

- Limited computational power
- Small datasets
- Vanishing gradient problem
- Competition from SVMs

Consequences

- Reduced funding for neural networks
- Focus on simpler, more interpretable models
- Neural networks considered impractical

The Perfect Storm: Dawn of Deep Learning

Key Developments

Hardware

- GPU computing
- Parallel processing

Data

- ImageNet (14M images)
- Big data era

Algorithms

- ReLU activation
- Dropout regularization
- Better optimizers (Adam)

Modern Neural Network Architectures

CNNs

- Convolutional Neural Networks
- Image processing
- Spatial hierarchies

RNNs/LSTMs

- Recurrent Neural Networks
- Sequence data
- Time series, NLP

Transformers

- Self-attention mechanism
- Large language models
- State-of-the-art NLP

Key Takeaways

Historical Journey

- Biological inspiration $\rightarrow$ Mathematical models
- Perceptron limitations $\rightarrow$ AI winters
- Multi-layer networks $\rightarrow$ Modern renaissance

Technical Foundations

- Weighted sum + non-linearity
- Backpropagation for learning
- Matrix operations for efficiency

The Big Picture

Why Neural Networks Work Now

- **Scale**: More data + more computation = better performance
- **Architecture**: Better understanding of network design
- **Algorithms**: Improved training techniques
- **Infrastructure**: Powerful hardware and software frameworks

Current Applications

- Computer vision (self-driving cars, medical imaging)
- Natural language processing (translation, chatbots)
- Recommendation systems (Netflix, Amazon)
- Scientific discovery (protein folding, drug discovery)