Artificial Neural Networks

CMSC 173 - Machine Learning

Course Lecture

Outline

Introduction & Motivation

What Are Neural Networks?

Artificial Neural Networks: Computing systems inspired by biological neural networks

Biological Inspiration

- **Neurons**: Basic processing units
- **Synapses**: Weighted connections
- **Learning**: Adapting connection strengths
- **Parallel processing**: Massive connectivity

Artificial Counterpart

- **Perceptrons**: Mathematical neurons
- **Weights**: Learnable parameters
- **Training**: Gradient-based optimization
- **Layers**: Organized processing units

Key Insight

Neural networks can **learn complex non-linear mappings** from data by adjusting weights through training.

Why Neural Networks?

Motivation: Limitations of Linear Models

Linear Models

- Limited to linear decision boundaries
- Cannot solve XOR problem
- Restricted representational power
- Simple but insufficient for complex data

Example: XOR Problem

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

No linear classifier can solve this!

Neural Networks

- Non-linear decision boundaries
- Universal approximation capability
- Hierarchical feature learning
- Scalable to complex problems

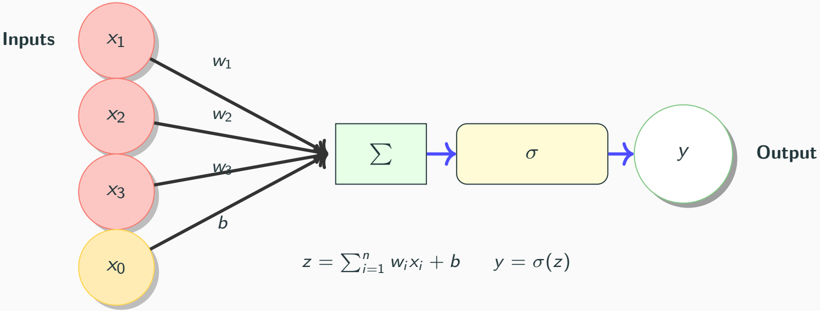
Universal Approximation Theorem: A neural network with a single hidden layer can approximate any continuous function to arbitrary accuracy (given sufficient neurons).

Key Advantages:

- Automatic feature extraction
- End-to-end learning
- Flexible architectures

The Perceptron

The Perceptron: Building Block of Neural Networks



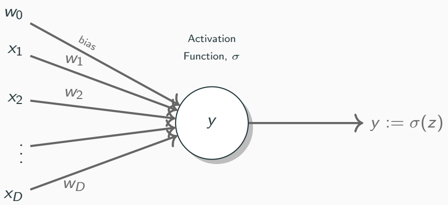
Mathematical Model

Linear Combination: $z = \sum_{i=1}^{n} w_i x_i + b = \mathbf{w}^T \mathbf{x} + b$

Activation: $y = \sigma(z)$ where σ is an activation function

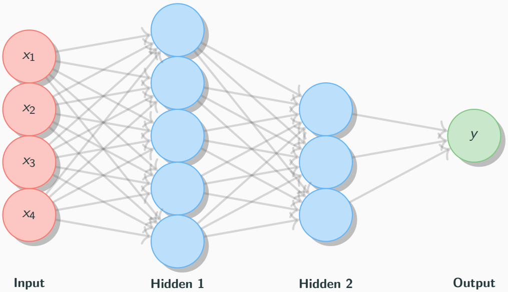
Neural Network Components and Architecture

Single Processing Unit



Single processing unit with inputs $x_1, \ldots, x_D$, weights $w_1, \ldots, w_D$, bias w_0, and activation function σ.

Multi-Layer Perceptron



Multi-layer perceptron with fully connected layers. Each connection represents a learnable weight parameter.

Key Concepts

Processing Unit: $z = \sum_{i=1}^{D} w_i x_i + w_0$, then $y = \sigma(z)$

Network: Multiple units arranged in layers with feedforward connections

Perceptron: Mathematical Formulation

Complete Mathematical Description:

$$z = \sum_{i=1}^{n} w_i x_i + b = \mathbf{w}^T \mathbf{x} + b \tag{1}$$

$$y = \sigma(z) = \sigma(\mathbf{w}^T \mathbf{x} + b) \tag{2}$$

where:

- $\mathbf{x} = [x_1, x_2, \ldots, x_n]^T$: input vector
- $\mathbf{w} = [w_1, w_2, \ldots, w_n]^T$: weight vector
- b: bias term
- $\sigma(\cdot)$: activation function

Step Function (Original)

$$\sigma(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

Problem: Not differentiable

Sigmoid Function (Modern)

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

Advantage: Smooth and differentiable

Perceptron Learning Algorithm

Goal: Learn weights $\mathbf{w}$ and bias b to minimize prediction error

Original Perceptron Rule

For misclassified point (x_i, y_i):

$$w_j := w_j + \alpha(y_i - \hat{y}_i)x_{ij}$$

$$b := b + \alpha(y_i - \hat{y}_i)$$

where α is the learning rate.

Convergence: Guaranteed for linearly separable data

Gradient Descent (Modern)

Define loss function: $L = \frac{1}{2}(y - \hat{y})^2$

Weight updates:

$$w_j := w_j - \alpha \frac{\partial L}{\partial w_j} \tag{3}$$

$$= w_j - \alpha(y - \hat{y})\sigma'(z)x_j \tag{4}$$

$$b := b - \alpha \frac{\partial L}{\partial b} \tag{5}$$

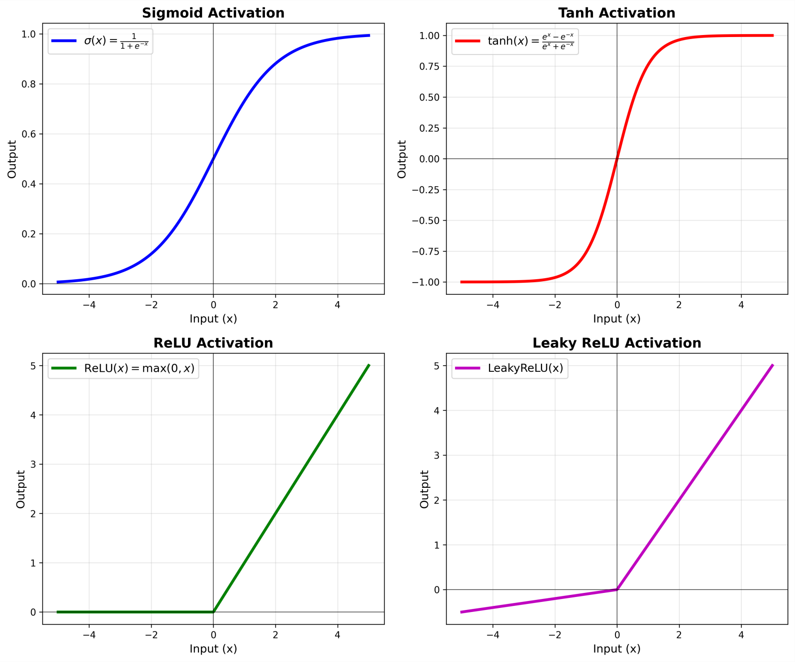
$$= b - \alpha(y - \hat{y})\sigma'(z) \tag{6}$$

Limitation

Single perceptron can only learn **linearly separable** functions. Solution: **Multi-layer networks!**

Activation Functions

Activation Functions: The Heart of Non-linearity



Activation Functions: Mathematical Properties

Sigmoid Function

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

Properties:

- Range: $(0, 1)$
- Smooth and differentiable
- Output interpretable as probability

Derivative:

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

Issues: Vanishing gradients for large $|x|$

Hyperbolic Tangent

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

Properties:

- Range: $(-1, 1)$
- Zero-centered output
- Steeper gradients than sigmoid

Derivative:

$$\tanh'(x) = 1 - \tanh^2(x)$$

Advantage: Better than sigmoid

Activation Functions: ReLU Family

ReLU (Rectified Linear Unit)

$$\text{ReLU}(x) = \max(0, x)$$

Advantages:

- Computationally efficient
- No vanishing gradient for $x > 0$
- Sparse activation
- Most popular choice

Derivative:

$$\text{ReLU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

Leaky ReLU

$$\text{LeakyReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases}$$

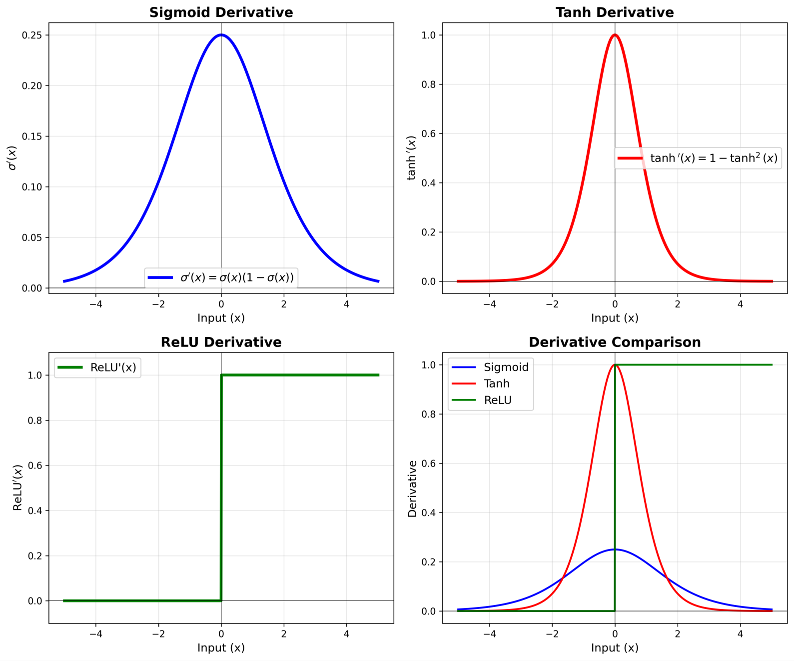
Advantages:

- Avoids "dying ReLU" problem
- Small gradient for negative inputs
- Typically $\alpha = 0.01$

Derivative:

$$\text{LeakyReLU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ \alpha & \text{if } x \leq 0 \end{cases}$$

Activation Function Derivatives



Choosing Activation Functions

Guidelines

Hidden Layers:

- **ReLU**: Default choice (fast, effective)
- **Leaky ReLU**: If dying ReLU is a problem
- **Tanh**: For zero-centered data
- **Sigmoid**: Avoid (vanishing gradients)

Output Layer:

- **Sigmoid**: Binary classification
- **Softmax**: Multi-class classification
- **Linear**: Regression
- **Tanh**: Regression (bounded output)

Common Issues

Vanishing Gradients:

- Sigmoid/Tanh derivatives $\rightarrow 0$ for large inputs
- Deep networks suffer from this
- Solution: ReLU activations

Dying ReLU:

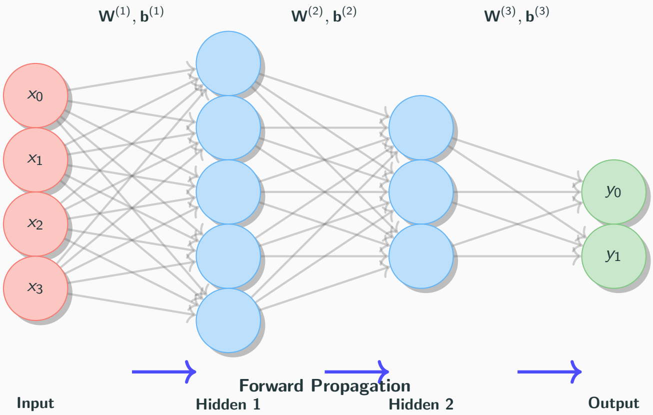
- Neurons get stuck at zero output
- No gradient flows through
- Solution: Leaky ReLU, initialization

Best Practice

Start with ReLU for hidden layers and choose output activation based on your task.

Multi-Layer Networks & Architecture

Multi-Layer Neural Network Architecture



Key Components

Layers: Input → Hidden → Hidden → ... → Output

Connections: Each neuron connects to all neurons in the next layer (fully connected)

Network Architecture: Mathematical Representation

For a network with L layers:

$$\mathbf{a}^{(0)} = \mathbf{x} \quad \text{(input layer)} \tag{7}$$

$$\mathbf{z}^{(l)} = \mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)} \quad \text{for } l = 1, 2, \ldots, L \tag{8}$$

$$\mathbf{a}^{(l)} = \sigma^{(l)}(\mathbf{z}^{(l)}) \quad \text{for } l = 1, 2, \ldots, L \tag{9}$$

$$\hat{\mathbf{y}} = \mathbf{a}^{(L)} \quad \text{(output layer)} \tag{10}$$

where:

- $\mathbf{W}^{(l)} \in \mathbb{R}^{n_l \times n_{l-1}}$: weight matrix for layer l
- $\mathbf{b}^{(l)} \in \mathbb{R}^{n_l}$: bias vector for layer l
- n_l: number of neurons in layer l
- $\sigma^{(l)}$: activation function for layer l

Network Dimensions and Parameters

Matrix Dimensions

For layer l:

- Input: $\mathbf{a}^{(l-1)}$ has shape $(n_{l-1}, 1)$
- Weights: $\mathbf{W}^{(l)}$ has shape (n_l, n_{l-1})
- Output: $\mathbf{a}^{(l)}$ has shape $(n_l, 1)$

Batch Processing:

- Input batch: $\mathbf{A}^{(l-1)}$ has shape (n_{l-1}, m)
- Output batch: $\mathbf{A}^{(l)}$ has shape (n_l, m)
- where m is the batch size

Parameter Count

Total parameters:

$$\sum_{l=1}^{L}(n_l \times n_{l-1} + n_l)$$

Example: $784 \to 128 \to 64 \to 10$

$$784 \times 128 + 128 \tag{11}$$
$$+ 128 \times 64 + 64 \tag{12}$$
$$+ 64 \times 10 + 10 \tag{13}$$
$$= 109,386 \text{ parameters} \tag{14}$$

Memory scales with:

- Network depth
- Layer width
- Batch size

Network Design Considerations

Depth vs Width

Deeper Networks:

- More layers, fewer neurons per layer
- Better feature hierarchies
- Can represent more complex functions
- Risk: vanishing gradients

Wider Networks:

- Fewer layers, more neurons per layer
- More parameters at each level
- Easier to train
- Risk: overfitting

Architecture Guidelines

Hidden Layer Size:

- Start with 1-2 hidden layers
- Size between input and output dimensions
- Rule of thumb: $\sqrt{n_{input} \times n_{output}}$

Number of Layers:

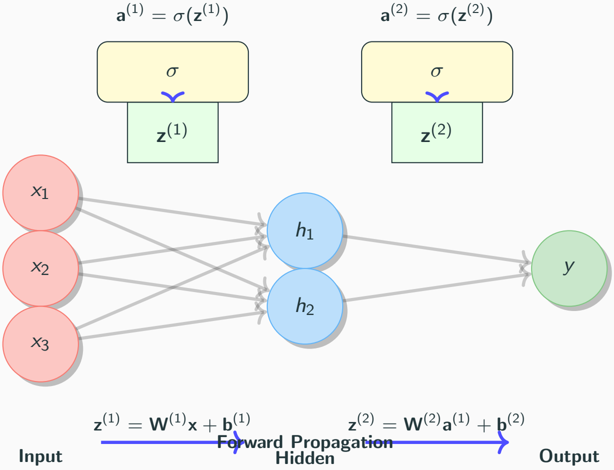
- Simple problems: 1-2 hidden layers
- Complex problems: 3+ layers
- Very deep: Requires special techniques

Rule of Thumb

Start simple and gradually increase complexity. Use validation performance to guide architecture choices.

Forward Propagation

Forward Propagation: Information Flow



Forward Pass

Information flows from **input to output**, layer by layer, to compute predictions.

Forward Propagation Algorithm

Step-by-step Process:

Algorithm 1 Forward Propagation

1: **Input:** $\mathbf{x}$, weights $\{\mathbf{W}^{(l)}\}$, biases $\{\mathbf{b}^{(l)}\}$
2: Set $\mathbf{a}^{(0)} = \mathbf{x}$
3: **for** $l = 1$ to L **do**
4: Compute pre-activation: $\mathbf{z}^{(l)} = \mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}$
5: Apply activation: $\mathbf{a}^{(l)} = \sigma^{(l)}(\mathbf{z}^{(l)})$
6: **end for**
7: **Output:** $\hat{\mathbf{y}} = \mathbf{a}^{(L)}$

Vectorized Implementation

For batch processing:

$$\mathbf{Z}^{(l)} = \mathbf{A}^{(l-1)}\mathbf{W}^{(l)T} + \mathbf{b}^{(l)} \tag{15}$$

$$\mathbf{A}^{(l)} = \sigma^{(l)}(\mathbf{Z}^{(l)}) \tag{16}$$

where $\mathbf{A}^{(l)}$ has shape (m, n_l) for m examples.

Computational Complexity: $O(L \cdot N \cdot M)$ where L = layers, N = max neurons/layer, M = batch size

Example Calculation

Network: $2 \rightarrow 3 \rightarrow 1$ Input: $\mathbf{x} = [0.5, 0.8]^T$

Layer 1: $\mathbf{z}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$ $\mathbf{a}^{(1)} = \sigma(\mathbf{z}^{(1)})$

Layer 2: $z^{(2)} = \mathbf{w}^{(2)T}\mathbf{a}^{(1)} + b^{(2)}$ $\hat{y} = \sigma(z^{(2)})$

All intermediate values $\mathbf{z}^{(l)}, \mathbf{a}^{(l)}$ are stored for backpropagation.

Forward Propagation: Implementation Details

Memory Considerations

Storage Requirements:

- Store all activations $\mathbf{a}^{(l)}$
- Store all pre-activations $\mathbf{z}^{(l)}$
- Needed for backpropagation

Memory Usage:

$$\text{Memory} \propto \sum_{l=0}^{L} n_l \times \text{batch\_size}$$

Trade-offs:

- Larger batches: More memory, better GPU utilization
- Smaller batches: Less memory, more gradient noise

Numerical Stability

Common Issues:

- **Overflow:** Large intermediate values
- **Underflow:** Very small values $\to 0$
- **NaN propagation:** Invalid operations

Solutions:

- Proper weight initialization
- Batch normalization
- Gradient clipping
- Use stable activation functions (ReLU)

Key Insight

Forward propagation is computationally straightforward, but proper implementation requires attention to **memory usage** and **numerical stability**.

Forward Pass: Handworked Example

Network: 2 inputs → 2 hidden → 1 output (sigmoid activation)

Given

Input: $\mathbf{x} = \begin{bmatrix} 0.5 \\ 0.8 \end{bmatrix}$

Weights & Biases:

$$\mathbf{W}^{(1)} = \begin{bmatrix} 0.2 & 0.4 \\ 0.3 & 0.1 \end{bmatrix}, \quad \mathbf{b}^{(1)} = \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix}$$

$$\mathbf{W}^{(2)} = \begin{bmatrix} 0.6 & 0.5 \end{bmatrix}, \quad b^{(2)} = 0.3$$

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

Step 1: Hidden Layer

$$\mathbf{z}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$$

$$= \begin{bmatrix} 0.2 & 0.4 \\ 0.3 & 0.1 \end{bmatrix} \begin{bmatrix} 0.5 \\ 0.8 \end{bmatrix} + \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix}$$

$$= \begin{bmatrix} 0.2(0.5) + 0.4(0.8) \\ 0.3(0.5) + 0.1(0.8) \end{bmatrix} + \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix}$$

$$= \begin{bmatrix} 0.1 + 0.32 \\ 0.15 + 0.08 \end{bmatrix} + \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix} = \begin{bmatrix} 0.52 \\ 0.43 \end{bmatrix}$$

Forward Pass: Handworked Example (continued)

Step 2: Hidden Activations

$$\mathbf{a}^{(1)} = \sigma(\mathbf{z}^{(1)}) = \sigma\left(\begin{bmatrix} 0.52 \\ 0.43 \end{bmatrix}\right)$$

$$a_1^{(1)} = \sigma(0.52) = \frac{1}{1 + e^{-0.52}} = \frac{1}{1 + 0.595} = 0.627$$

$$a_2^{(1)} = \sigma(0.43) = \frac{1}{1 + e^{-0.43}} = \frac{1}{1 + 0.651} = 0.606$$

$$\mathbf{a}^{(1)} = \begin{bmatrix} 0.627 \\ 0.606 \end{bmatrix}$$

Step 3: Output Layer

$$z^{(2)} = \mathbf{W}^{(2)}\mathbf{a}^{(1)} + b^{(2)}$$

$$= \begin{bmatrix} 0.6 & 0.5 \end{bmatrix} \begin{bmatrix} 0.627 \\ 0.606 \end{bmatrix} + 0.3$$

$$= 0.6(0.627) + 0.5(0.606) + 0.3$$

$$= 0.376 + 0.303 + 0.3 = 0.979$$

Final Output:

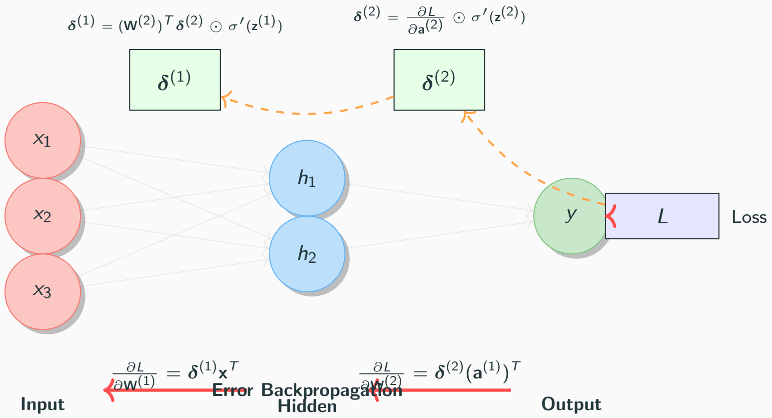
$$\hat{y} = \sigma(0.979) = \frac{1}{1 + e^{-0.979}} = 0.727$$

Summary

Input $[0.5, 0.8] \rightarrow$ Hidden $[0.627, 0.606] \rightarrow$ Output 0.727

Backpropagation Algorithm

Backpropagation: Error Flow



Backpropagation

Efficient algorithm to compute gradients by propagating errors **backward** through the network using the **chain rule**.

Mathematical Foundation: Chain Rule

Goal: Compute $\frac{\partial L}{\partial \mathbf{W}^{(l)}}$ and $\frac{\partial L}{\partial \mathbf{b}^{(l)}}$ for all layers

Chain Rule Application:

$$\frac{\partial L}{\partial \mathbf{W}^{(l)}} = \frac{\partial L}{\partial \mathbf{z}^{(l)}} \frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{W}^{(l)}} \tag{17}$$

$$\frac{\partial L}{\partial \mathbf{b}^{(l)}} = \frac{\partial L}{\partial \mathbf{z}^{(l)}} \frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{b}^{(l)}} \tag{18}$$

$$\frac{\partial L}{\partial \mathbf{a}^{(l-1)}} = \frac{\partial L}{\partial \mathbf{z}^{(l)}} \frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{a}^{(l-1)}} \tag{19}$$

Key Insight: Define error terms $\boldsymbol{\delta}^{(l)} = \frac{\partial L}{\partial \mathbf{z}^{(l)}}$

Gradient Computations

$$\frac{\partial L}{\partial \mathbf{W}^{(l)}} = \boldsymbol{\delta}^{(l)} (\mathbf{a}^{(l-1)})^T \tag{20}$$

$$\frac{\partial L}{\partial \mathbf{b}^{(l)}} = \boldsymbol{\delta}^{(l)} \tag{21}$$

$$\boldsymbol{\delta}^{(l-1)} = (\mathbf{W}^{(l)})^T \boldsymbol{\delta}^{(l)} \odot \sigma'(\mathbf{z}^{(l-1)}) \tag{22}$$

Output Layer

For output layer L:

$$\boldsymbol{\delta}^{(L)} = \frac{\partial L}{\partial \mathbf{a}^{(L)}} \odot \sigma'(\mathbf{z}^{(L)})$$

Common case (MSE + sigmoid):

$$\boldsymbol{\delta}^{(L)} = (\mathbf{a}^{(L)} - \mathbf{y}) \odot \mathbf{a}^{(L)} \odot (1 - \mathbf{a}^{(L)})$$

where $\odot$ denotes element-wise multiplication.

Backpropagation Algorithm

Algorithm 2 Backpropagation

Input: Training example $(\mathbf{x}, \mathbf{y})$, network weights
Forward Pass: Compute all $\mathbf{a}^{(l)}$ and $\mathbf{z}^{(l)}$ (store them!)
Compute Output Error: $\delta^{(L)} = \frac{\partial L}{\partial \mathbf{a}^{(L)}} \odot \sigma'(\mathbf{z}^{(L)})$
for $l = L - 1$ down to 1 **do**
    Propagate Error: $\delta^{(l)} = (\mathbf{W}^{(l+1)})^T \delta^{(l+1)} \odot \sigma'(\mathbf{z}^{(l)})$
end for
for $l = 1$ to L **do**
    Compute Gradients:
    $\frac{\partial L}{\partial \mathbf{W}^{(l)}} = \delta^{(l)} (\mathbf{a}^{(l-1)})^T$
    $\frac{\partial L}{\partial \mathbf{b}^{(l)}} = \delta^{(l)}$
end for

Computational Complexity

Time Complexity: O(number of weights)

- Same order as forward pass
- Very efficient compared to numerical gradients
- Enables training of large networks

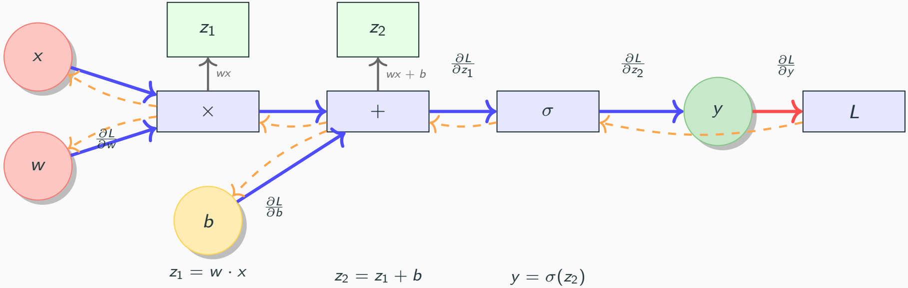
Space Complexity: O(network size)

Why Backpropagation Works

- **Efficiency:** Reuses computations via chain rule
- **Automatic:** No manual gradient derivation
- **Exact:** Computes exact gradients (no approximation)
- **General:** Works for any differentiable network

Historical Impact:

Computational Graph Perspective



Modern View

Backpropagation is **automatic differentiation** applied to computational graphs. Modern frameworks (TensorFlow, PyTorch) build these graphs automatically.

4-Layer Neural Network: Differential Equation Derivation

Network Structure: Input → Hidden1 → Hidden2 → Hidden3 → Output

Forward Pass Equations:

$$\mathbf{a}^{(0)} = \mathbf{x} \quad \text{(input)} \tag{23}$$

$$\mathbf{z}^{(1)} = \mathbf{W}^{(1)}\mathbf{a}^{(0)} + \mathbf{b}^{(1)}, \quad \mathbf{a}^{(1)} = \sigma(\mathbf{z}^{(1)}) \tag{24}$$

$$\mathbf{z}^{(2)} = \mathbf{W}^{(2)}\mathbf{a}^{(1)} + \mathbf{b}^{(2)}, \quad \mathbf{a}^{(2)} = \sigma(\mathbf{z}^{(2)}) \tag{25}$$

$$\mathbf{z}^{(3)} = \mathbf{W}^{(3)}\mathbf{a}^{(2)} + \mathbf{b}^{(3)}, \quad \mathbf{a}^{(3)} = \sigma(\mathbf{z}^{(3)}) \tag{26}$$

$$\mathbf{z}^{(4)} = \mathbf{W}^{(4)}\mathbf{a}^{(3)} + \mathbf{b}^{(4)}, \quad \mathbf{a}^{(4)} = \sigma(\mathbf{z}^{(4)}) \quad \text{(output)} \tag{27}$$

Loss Function: $L = \frac{1}{2}||\mathbf{a}^{(4)} - \mathbf{y}||^2$

Output Layer Error

Starting from the output layer:

$$\boldsymbol{\delta}^{(4)} = \frac{\partial L}{\partial \mathbf{z}^{(4)}} \tag{28}$$

$$= \frac{\partial L}{\partial \mathbf{a}^{(4)}} \odot \frac{\partial \mathbf{a}^{(4)}}{\partial \mathbf{z}^{(4)}} \tag{29}$$

$$= (\mathbf{a}^{(4)} - \mathbf{y}) \odot \sigma'(\mathbf{z}^{(4)}) \tag{30}$$

Chain Rule Application

For hidden layers ($l = 3, 2, 1$):

$$\boldsymbol{\delta}^{(l)} = \frac{\partial L}{\partial \mathbf{z}^{(l)}} \tag{31}$$

$$= \frac{\partial L}{\partial \mathbf{z}^{(l+1)}} \frac{\partial \mathbf{z}^{(l+1)}}{\partial \mathbf{a}^{(l)}} \frac{\partial \mathbf{a}^{(l)}}{\partial \mathbf{z}^{(l)}} \tag{32}$$

$$= (\mathbf{W}^{(l+1)})^T \boldsymbol{\delta}^{(l+1)} \odot \sigma'(\mathbf{z}^{(l)}) \tag{33}$$

4-Layer Network: Complete Backpropagation Derivation

Step-by-Step Gradient Computation:

Error Propagation

Layer 4 (Output):

$$\boldsymbol{\delta}^{(4)} = (\mathbf{a}^{(4)} - \mathbf{y}) \odot \sigma'(\mathbf{z}^{(4)})$$

Layer 3:

$$\boldsymbol{\delta}^{(3)} = (\mathbf{W}^{(4)})^T \boldsymbol{\delta}^{(4)} \odot \sigma'(\mathbf{z}^{(3)})$$

Layer 2:

$$\boldsymbol{\delta}^{(2)} = (\mathbf{W}^{(3)})^T \boldsymbol{\delta}^{(3)} \odot \sigma'(\mathbf{z}^{(2)})$$

Layer 1:

$$\boldsymbol{\delta}^{(1)} = (\mathbf{W}^{(2)})^T \boldsymbol{\delta}^{(2)} \odot \sigma'(\mathbf{z}^{(1)})$$

Weight and Bias Gradients

For each layer $l = 1, 2, 3, 4$:

Weight Gradients:

$$\frac{\partial L}{\partial \mathbf{W}^{(l)}} = \boldsymbol{\delta}^{(l)} (\mathbf{a}^{(l-1)})^T$$

Bias Gradients:

$$\frac{\partial L}{\partial \mathbf{b}^{(l)}} = \boldsymbol{\delta}^{(l)}$$

Update Rules:

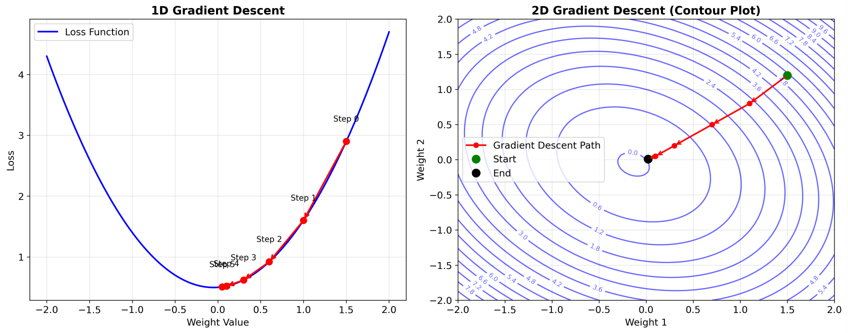
$$\mathbf{W}^{(l)} := \mathbf{W}^{(l)} - \alpha \frac{\partial L}{\partial \mathbf{W}^{(l)}} \tag{34}$$

$$\mathbf{b}^{(l)} := \mathbf{b}^{(l)} - \alpha \frac{\partial L}{\partial \mathbf{b}^{(l)}} \tag{35}$$

Key Insight

The error **flows backward** through the network, with each layer's error depending on the next layer's error multiplied by the transpose of the connecting weights.

Gradient Descent Optimization



Weight Update Rule

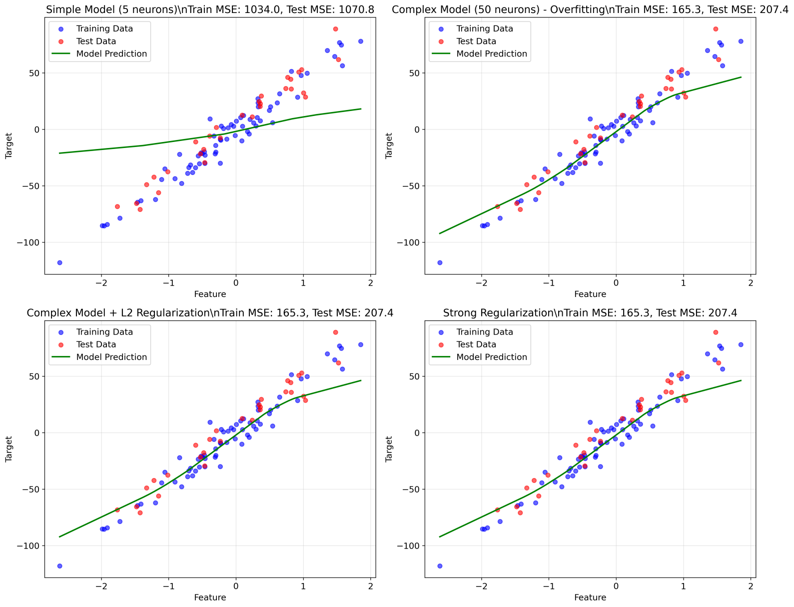
$$\mathbf{W}^{(l)} := \mathbf{W}^{(l)} - \alpha \frac{\partial L}{\partial \mathbf{W}^{(l)}}$$

$$\mathbf{b}^{(l)} := \mathbf{b}^{(l)} - \alpha \frac{\partial L}{\partial \mathbf{b}^{(l)}}$$

where α is the learning rate.

Regularization Techniques

The Overfitting Problem



Overfitting

Model learns training data too well, **memorizing noise** instead of learning generalizable patterns.

L1 and L2 Regularization

Add penalty terms to the loss function to control model complexity

L2 Regularization (Ridge)

$$L_{total} = L_{data} + \lambda \sum_l ||\mathbf{W}^{(l)}||_2^2$$

where $||\mathbf{W}^{(l)}||_2^2 = \sum_i \sum_j (W_{ij}^{(l)})^2$

Effect:

- Shrinks weights towards zero
- Uniform penalty on all weights
- Smooth weight distributions
- Preferred for most applications

Gradient Modification:

$$\frac{\partial L_{total}}{\partial \mathbf{W}^{(l)}} = \frac{\partial L_{data}}{\partial \mathbf{W}^{(l)}} + 2\lambda \mathbf{W}^{(l)}$$

L1 Regularization (Lasso)

$$L_{total} = L_{data} + \lambda \sum_l ||\mathbf{W}^{(l)}||_1$$

where $||\mathbf{W}^{(l)}||_1 = \sum_i \sum_j |W_{ij}^{(l)}|$

Effect:

- Promotes sparsity (many weights $\to 0$)
- Automatic feature selection
- Creates sparse networks
- Useful for interpretability

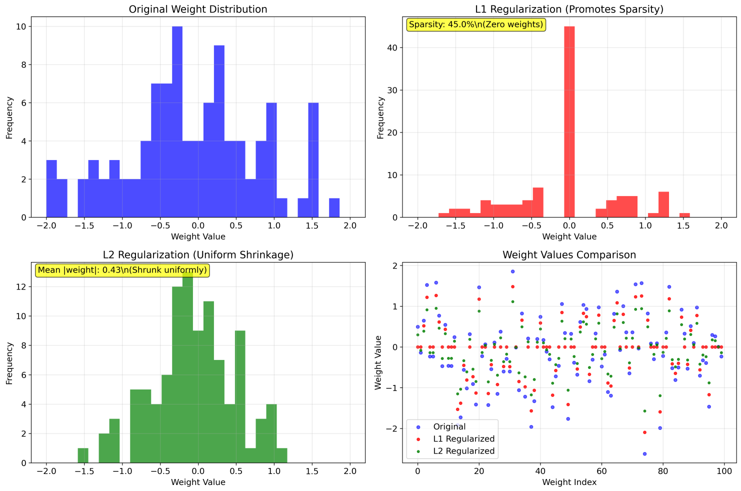
Gradient Modification:

$$\frac{\partial L_{total}}{\partial \mathbf{W}^{(l)}} = \frac{\partial L_{data}}{\partial \mathbf{W}^{(l)}} + \lambda \text{sign}(\mathbf{W}^{(l)})$$

Hyperparameter λ

Controls regularization strength: **larger** $\lambda \to$ more regularization $\to$ simpler model

L1 vs L2 Regularization Comparison



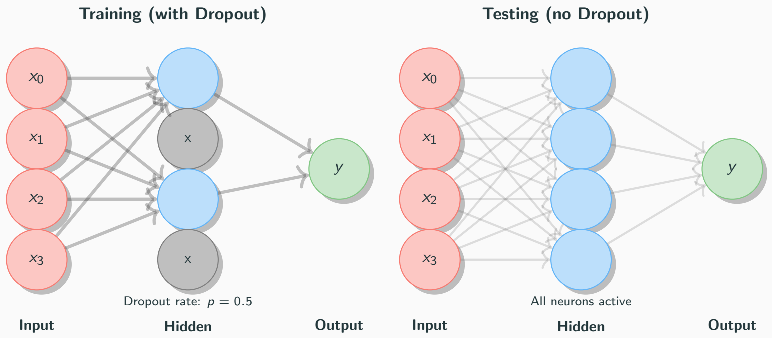
When to Use L2

- General-purpose regularization
- All features potentially relevant
- Want smooth weight shrinkage

When to Use L1

- Feature selection needed
- Many irrelevant features
- Want sparse models

Dropout: A Different Approach



Dropout Technique

Randomly **set neurons to zero** during training to prevent co-adaptation and improve generalization.

Dropout: Mathematical Formulation

Training Phase:

$$\mathbf{r}^{(l)} \sim \text{Bernoulli}(p) \quad \text{(dropout mask)} \tag{36}$$

$$\tilde{\mathbf{a}}^{(l)} = \mathbf{r}^{(l)} \odot \mathbf{a}^{(l)} \quad \text{(apply mask)} \tag{37}$$

$$\mathbf{z}^{(l+1)} = \mathbf{W}^{(l+1)}\tilde{\mathbf{a}}^{(l)} + \mathbf{b}^{(l+1)} \tag{38}$$

Testing Phase:

$$\mathbf{z}^{(l+1)} = p \cdot \mathbf{W}^{(l+1)}\mathbf{a}^{(l)} + \mathbf{b}^{(l+1)} \quad \text{(scale weights)} \tag{39}$$

Dropout Benefits

- **Prevents overfitting:** Reduces complex co-adaptations
- **Model averaging:** Approximates ensemble of networks
- **Robust features:** Forces redundant representations
- **Easy to implement:** Simple modification to forward pass

Typical rates: 0.2-0.5 for hidden layers, 0.1-0.2 for input

Implementation Notes

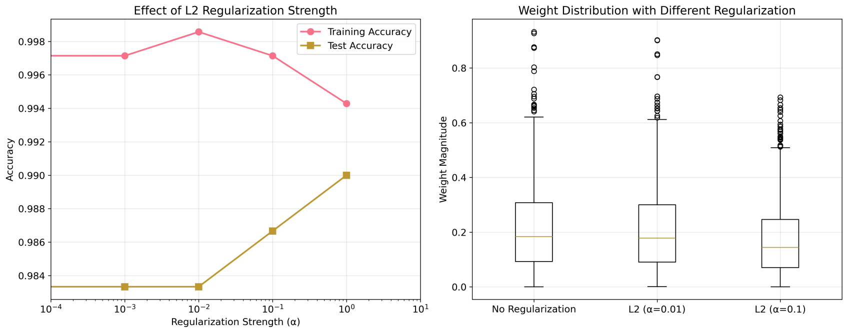
Training vs Testing:

- Training: Randomly drop neurons
- Testing: Use all neurons but scale outputs
- Modern frameworks handle this automatically

Why Scaling Works:

- Training: Each neuron is "on" with probability p
- Testing: All neurons are "on"
- Scaling by p maintains expected activation levels

Regularization Comparison



Choosing Regularization

Start with:

- L2 regularization ($\lambda = 0.01$)
- Dropout (rate $= 0.5$)
- Early stopping

If still overfitting:

- Increase regularization strength
- Add more dropout

Other Techniques

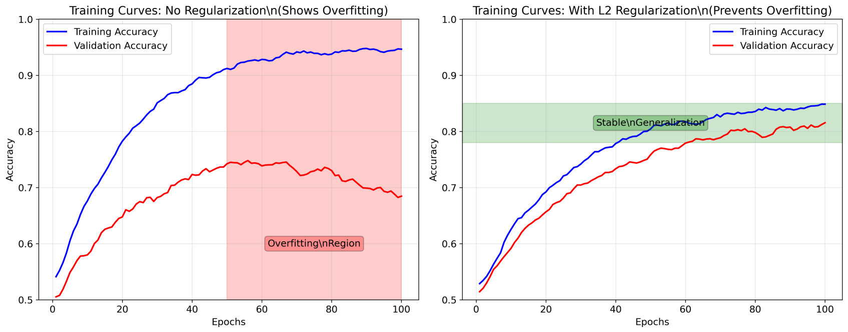
Early Stopping:

- Monitor validation loss
- Stop when it starts increasing
- Simple and effective

Data Augmentation:

- Artificially increase training data
- Add noise, rotations, etc.

Training Curves with Regularization



Monitoring Training

Use **validation curves** to detect overfitting and choose appropriate regularization strength.

Training Best Practices

Weight Initialization

Proper initialization is crucial for successful training

Poor Initialization

All zeros: No learning (symmetry)

$$W_{ij} = 0 \Rightarrow \text{no gradient flow}$$

Too large: Exploding gradients

$$W_{ij} \sim \mathcal{N}(0, 1) \Rightarrow \text{saturation}$$

Too small: Vanishing gradients

$$W_{ij} \sim \mathcal{N}(0, 0.01) \Rightarrow \text{weak signals}$$

Good Initialization

Xavier/Glorot (Sigmoid/Tanh):

$$W_{ij} \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n_{in} + n_{out}}}\right)$$

He initialization (ReLU):

$$W_{ij} \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n_{in}}}\right)$$

Bias initialization:

$$b_i = 0 \text{ (usually sufficient)}$$

Why These Work

Maintain **activation variance** and **gradient variance** across layers during initialization.

Learning Rate and Optimization

Learning Rate Selection

Too high: Overshooting, instability

- Loss explodes or oscillates
- Network doesn't converge
- Weights become very large

Too low: Slow convergence

- Training takes forever
- Gets stuck in local minima
- Poor final performance

Good range: Typically 10^{-4} to 10^{-1}

Advanced Optimizers

SGD with Momentum:

$$\mathbf{v}_t = \beta \mathbf{v}_{t-1} + (1-\beta)\nabla L$$

$$\mathbf{W} := \mathbf{W} - \alpha \mathbf{v}_t$$

Adam (Adaptive Moments):

$$\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\nabla L \tag{40}$$

$$\mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1-\beta_2)(\nabla L)^2 \tag{41}$$

$$\mathbf{W} := \mathbf{W} - \alpha \frac{\mathbf{m}_t}{\sqrt{\mathbf{v}_t} + \epsilon} \tag{42}$$

Default choice: Adam with $\alpha = 0.001$

Learning Rate Scheduling

Decay strategies: Step decay, exponential decay, cosine annealing. Start high, reduce during training.

Training Diagnostics

Monitor these metrics during training:

Loss Monitoring

- **Training loss**: Should decrease monotonically
- **Validation loss**: Should decrease, then stabilize
- **Gap**: Indicates overfitting if too large

Warning Signs:

- Loss increases: Learning rate too high
- Loss plateaus early: Learning rate too low
- Validation loss increases: Overfitting

Activation Monitoring

- **Activation statistics**: Mean, std, sparsity
- **Dead neurons**: Always output zero
- **Saturated neurons**: Always in saturation region

Healthy activations:

- Reasonable variance (not too small/large)
- Some sparsity (for ReLU)
- No layers completely dead

Gradient Monitoring

- **Gradient norms**: Should be reasonable (10^{-6} to 10^{-1})
- **Vanishing**: Gradients $\rightarrow 0$ in early layers
- **Exploding**: Gradients become very large

Weight Monitoring

- **Weight distributions**: Should be reasonable
- **Weight updates**: $|\Delta W|/|W| \approx 10^{-3}$
- **Layer-wise learning rates**: May need adjustment

Tools

Use **TensorBoard**, **Weights & Biases**, or similar tools for comprehensive monitoring and visualization

Common Problems and Solutions

Problem: Vanishing Gradients

Symptoms:

- Early layers don't learn
- Gradients approach zero
- Training stalls

Solutions:

- Use ReLU activations
- Proper weight initialization
- Batch normalization

Problem: Exploding Gradients

Symptoms:

- Loss becomes NaN
- Weights blow up
- Training becomes unstable

Solutions:

- Gradient clipping
- Lower learning rate
- Better initialization

Problem: Overfitting

Symptoms:

- Training accuracy ¿¿ validation accuracy
- Validation loss increases

Solutions:

- Add regularization (L2, dropout)
- Reduce model complexity
- More training data

Problem: Slow Convergence

Symptoms:

- Loss decreases very slowly
- Gets stuck in plateaus

Solutions:

- Increase learning rate
- Use adaptive optimizers (Adam)
- Learning rate scheduling

Summary & Applications

Neural Networks: Key Takeaways

Core Concepts

- **Perceptron**: Basic building block
- **Multi-layer**: Enable complex mappings
- **Activation functions**: Provide non-linearity
- **Forward propagation**: Compute predictions
- **Backpropagation**: Compute gradients efficiently
- **Regularization**: Prevent overfitting

Mathematical Foundation

- Matrix operations for efficiency
- Chain rule for gradient computation
- Optimization theory for training
- Probability theory for interpretation

Best Practices

- **Architecture**: Start simple, add complexity gradually
- **Initialization**: Xavier/He for proper gradient flow
- **Optimization**: Adam optimizer with proper learning rate
- **Regularization**: L2 + Dropout for generalization
- **Monitoring**: Track loss, gradients, activations
- **Debugging**: Systematic approach to problems

When to Use Neural Networks

- Large datasets available
- Complex non-linear patterns
- End-to-end learning desired
- Feature engineering is difficult

Modern Deep Learning

These fundamentals scale to modern architectures: **CNNs, RNNs, Transformers, ResNets, etc.**

Applications & Real-World Impact

Computer Vision

- **Image classification**: ResNet, EfficientNet
- **Object detection**: YOLO, R-CNN
- **Segmentation**: U-Net, Mask R-CNN
- **Face recognition**: DeepFace, FaceNet
- **Medical imaging**: Cancer detection, radiology

Natural Language Processing

- **Language models**: GPT, BERT, T5
- **Translation**: Google Translate, DeepL
- **Chatbots**: ChatGPT, virtual assistants
- **Text analysis**: Sentiment, summarization

Other Domains

- **Speech**: Recognition, synthesis, processing
- **Recommendation**: Netflix, Amazon, Spotify
- **Games**: AlphaGo, OpenAI Five, StarCraft
- **Robotics**: Control, perception, planning
- **Finance**: Trading, fraud detection, risk
- **Science**: Drug discovery, climate modeling

Emerging Areas

- **Generative AI**: DALL-E, Midjourney, Stable Diffusion
- **Multimodal**: CLIP, GPT-4V
- **Reinforcement Learning**: Autonomous systems
- **Scientific Computing**: Physics, chemistry, biology

Impact

Neural networks have **revolutionized AI** and are now fundamental to most modern machine learning applications.

Looking Forward: Advanced Topics

What's Next After This Foundation?

Specialized Architectures

- **Convolutional Neural Networks (CNNs)**
  - Spatial structure exploitation
  - Translation invariance
  - Computer vision applications
- **Recurrent Neural Networks (RNNs)**
  - Sequential data processing
  - Memory and temporal dynamics
  - LSTM, GRU variants
- **Transformer Networks**
  - Attention mechanisms
  - Parallel processing
  - Modern NLP backbone

Advanced Techniques

- **Batch Normalization**
  - Internal covariate shift
  - Training acceleration
- **Residual Connections**
  - Very deep networks
  - Gradient flow improvement
- **Attention Mechanisms**
  - Selective focus
  - Long-range dependencies
- **Generative Models**
  - VAEs, GANs, Diffusion
  - Creative AI applications

Next Steps

Practice implementation, experiment with **real datasets**, and explore **specialized architectures** for your domain of interest.

Questions?