Neural Networks and Function Approximation

Krishna Kumar

University of Texas at Austin

krishnak@utexas.edu

Overview

Outline

The 1D Poisson Equation: Our Benchmark Problem

Consider the one-dimensional Poisson equation on $[0, 1]$:

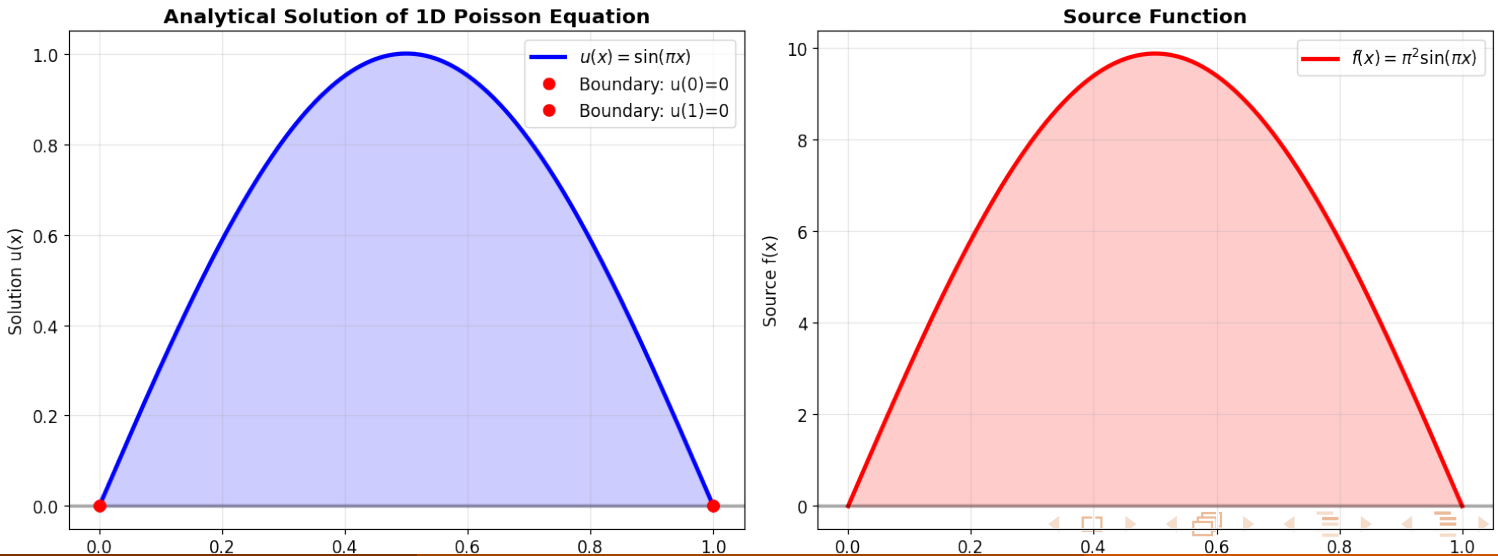
$$-\frac{d^2u}{dx^2} = f(x), \quad x \in [0, 1]$$

with boundary conditions:

$$u(0) = 0, \quad u(1) = 0$$

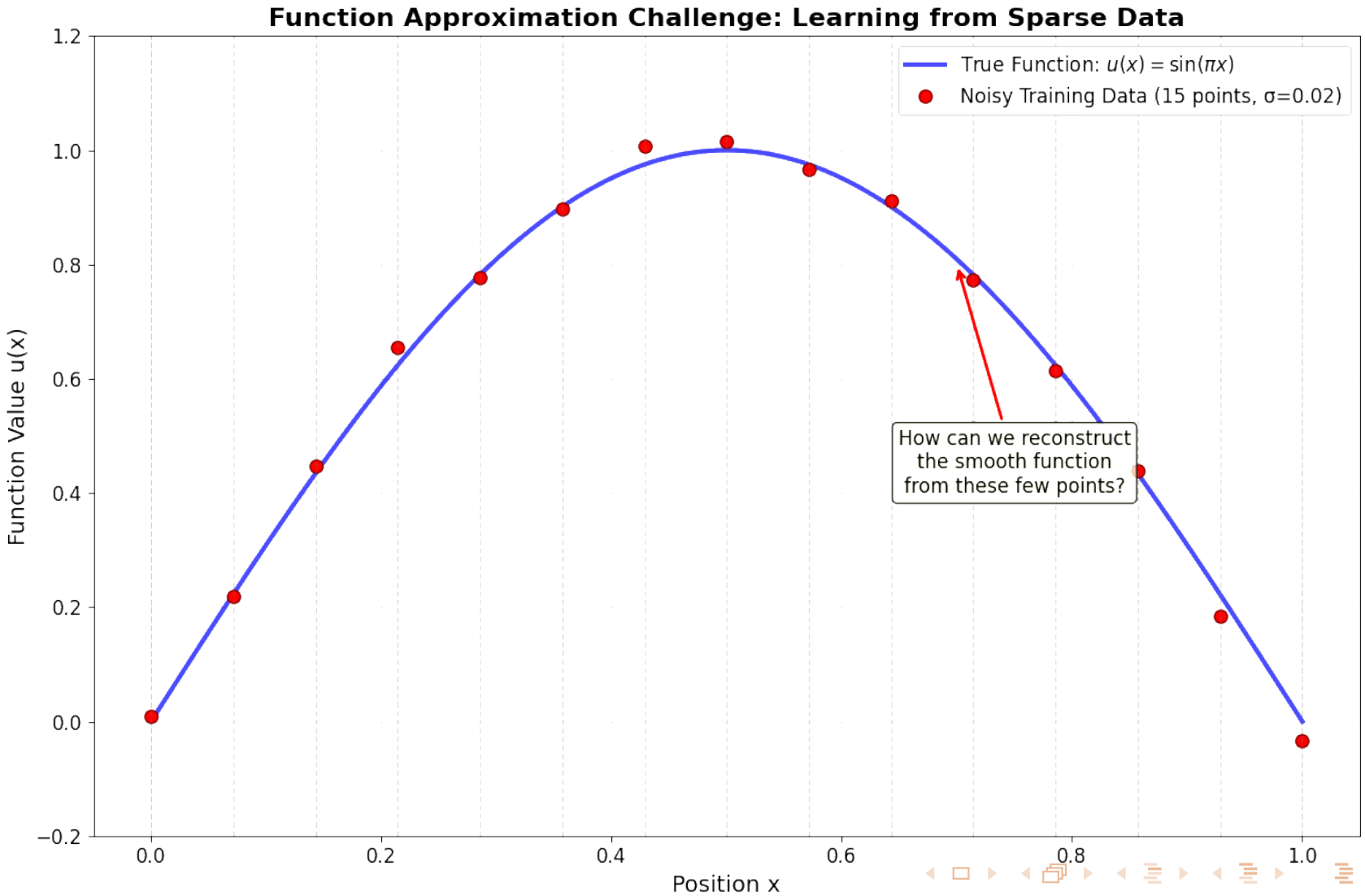
For $f(x) = \pi^2 \sin(\pi x)$, the analytical solution is:

$$u(x) = \sin(\pi x)$$



The Function Approximation Challenge

Key Question: Can we learn to approximate $u(x) = \sin(\pi x)$ from sparse data?



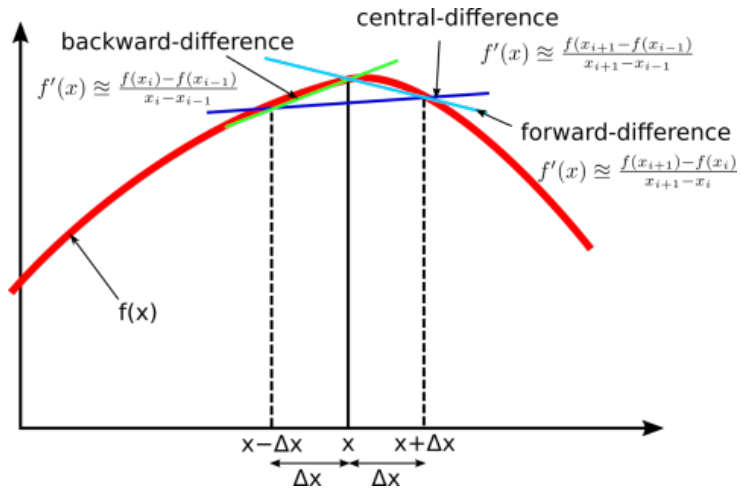
Traditional vs Neural Network Approaches

Finite Difference:

- Discretize domain into grid points
- Solve for values at specific locations
- Result: Discrete representation

Neural Network Approach:

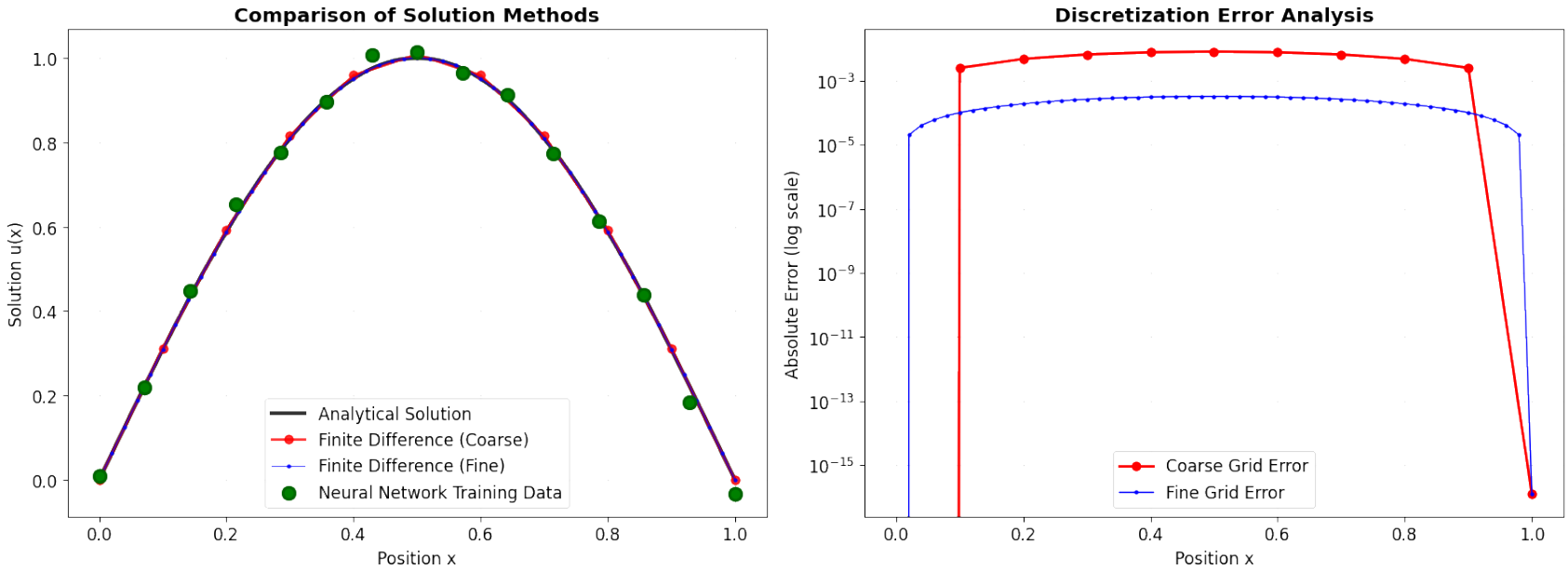
- Learn continuous function $u_{NN}(x;\theta)$
- Approximate solution over entire domain
- Parameters θ trained from sparse data



Finite Difference Approximation

$$\frac{d^2u}{dx^2} \approx \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2}$$

Finite Difference vs Neural Networks



Comparison of solution methods and discretization errors

Key Insight: Neural networks learn continuous functions, not just discrete values.

Outline

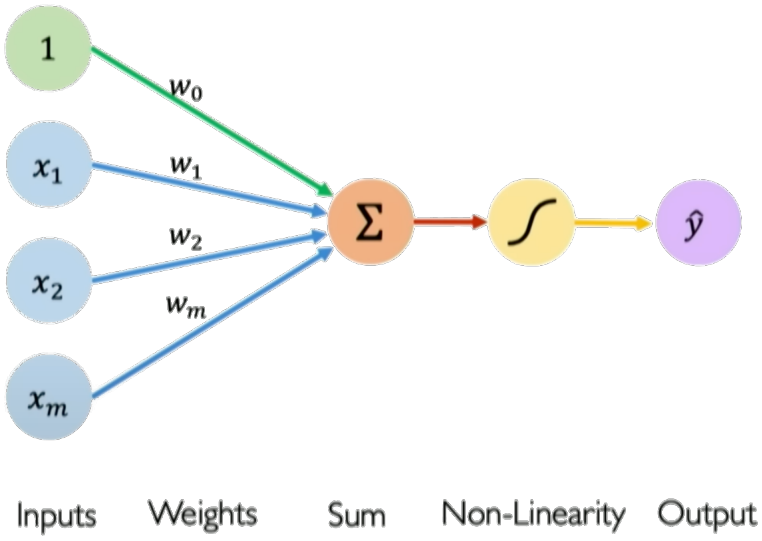
The Perceptron: Basic Building Block

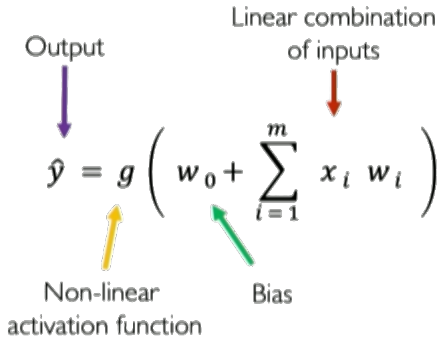
A perceptron computes:

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

$$\hat{y} = g(z)$$

where g is the activation function.





Perceptron architecture

Key components: Input vector, weights, bias, activation function, output.

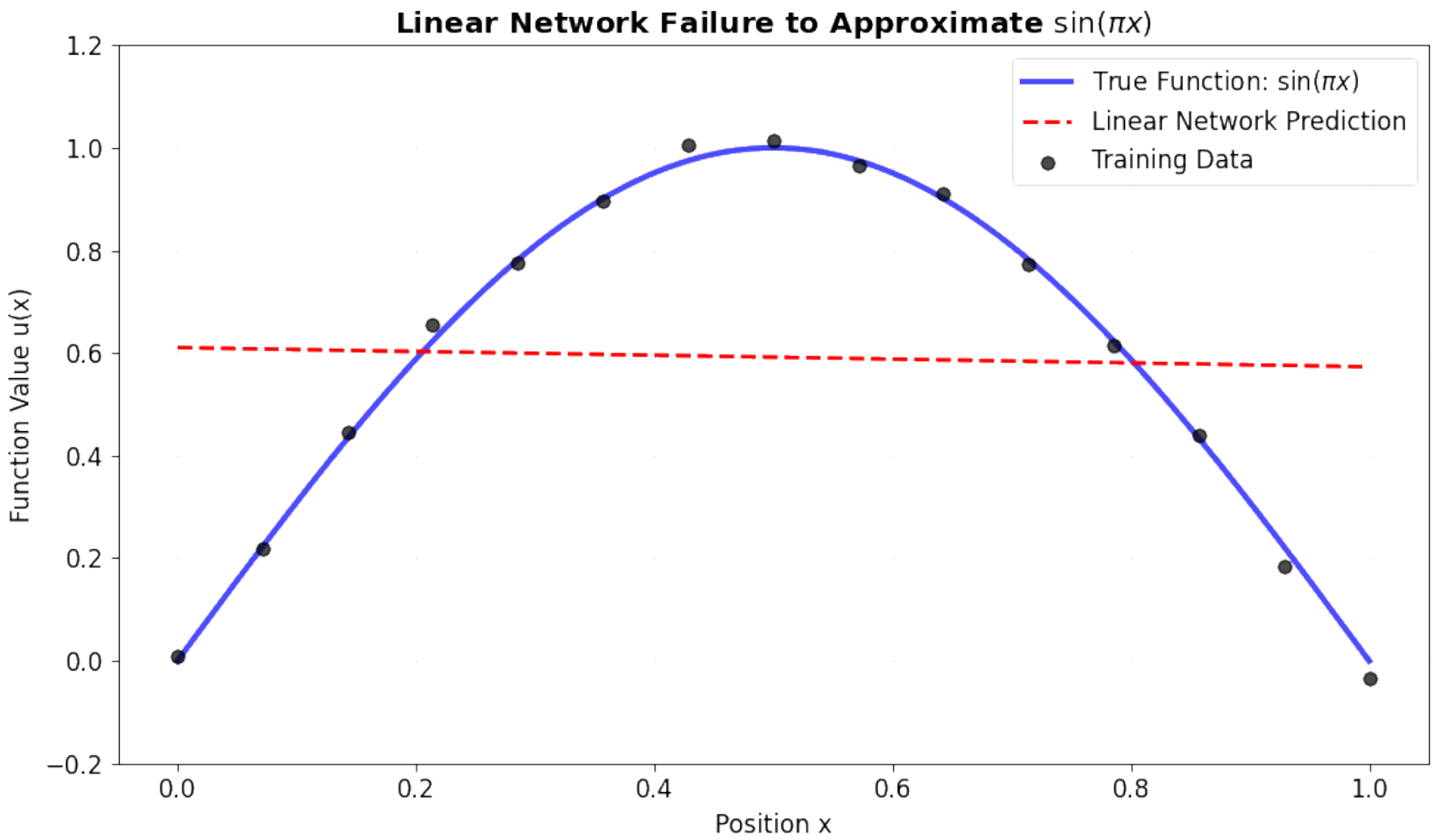
The Critical Role of Nonlinearity

Without activation functions: Multiple linear layers collapse to single linear transformation.
Consider two linear layers:

$$\begin{aligned} h_1 &= W_1 x + b_1 \\ h_2 &= W_2 h_1 + b_2 = W_2(W_1 x + b_1) + b_2 \\ &= (W_2 W_1)x + (W_2 b_1 + b_2) \end{aligned}$$

This is equivalent to: $h_2 = W_{eq} x + b_{eq}$ (still linear!)
Problem: Linear networks can only learn $y = mx + b$, but $\sin(\pi x)$ is curved!

Demonstrating Linear Network Failure



Linear network cannot approximate $\sin(\pi x)$

Conclusion: Nonlinearity is essential for complex function approximation.

Common Activation Functions

Sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$ (squashes to (0,1))

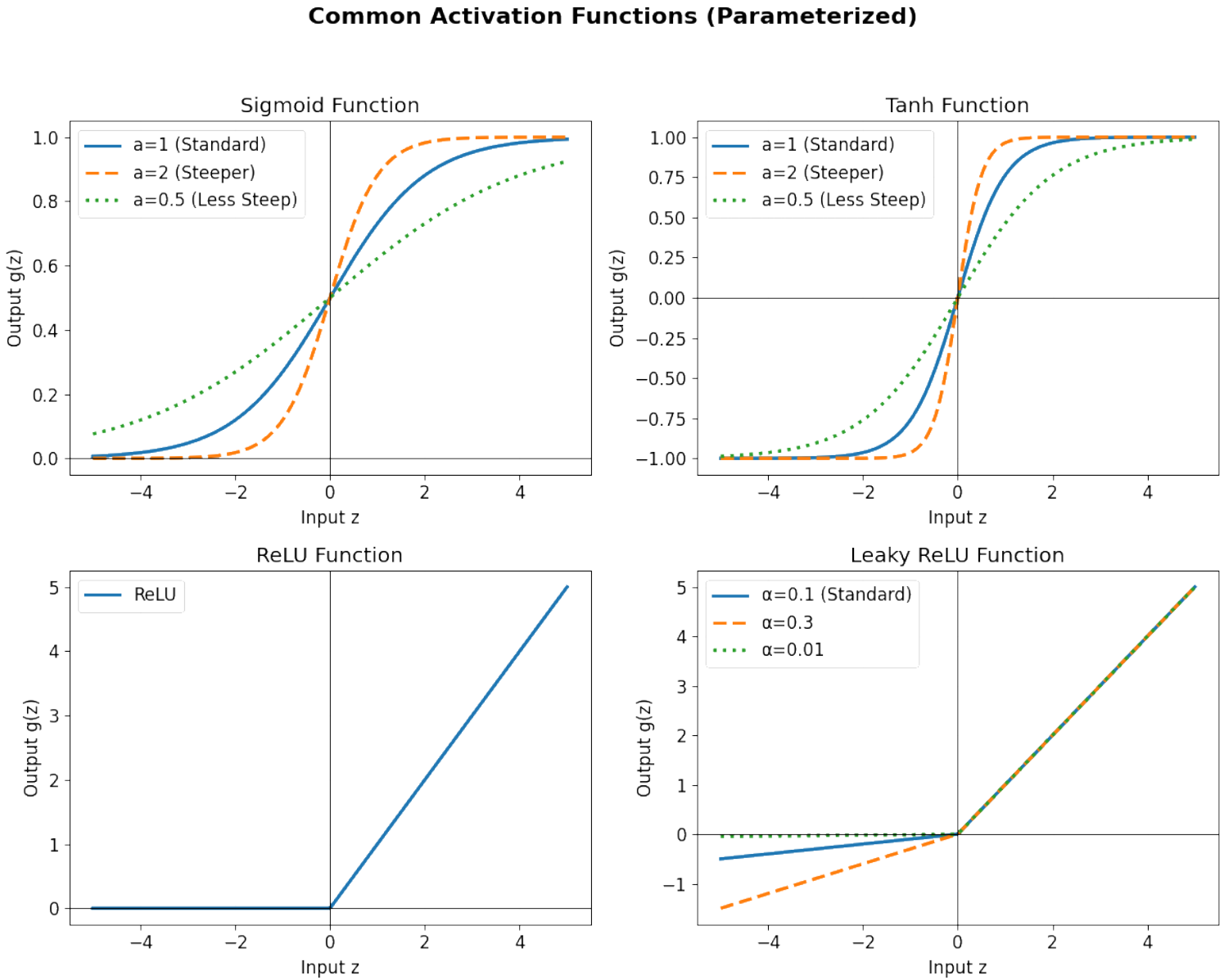
Tanh: $\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$ (squashes to (-1,1))

ReLU: $f(x) = \max(0, x)$ (efficient, prevents vanishing gradients)

Leaky ReLU: $f(x) = \max(\alpha x, x)$ (prevents dead neurons)

▸ ReLU Demo

Common Activation Functions



Outline

Universal Approximation Theorem

Theorem (Cybenko, 1989): A single hidden layer network with sufficient neurons can approximate any continuous function to arbitrary accuracy.

$$F(x) = \sum_{i=1}^{N} w_i \sigma(v_i x + b_i) + w_0$$

Mathematical statement: For any continuous $f : [0, 1] \to \mathbb{R}$ and $\epsilon > 0$, there exists N and parameters such that $|F(x) - f(x)| < \epsilon$ for all $x \in [0, 1]$.

Key Questions:

- How many neurons N do we need?
- Is this practical?
- Can we verify experimentally?

Single Hidden Layer Architecture

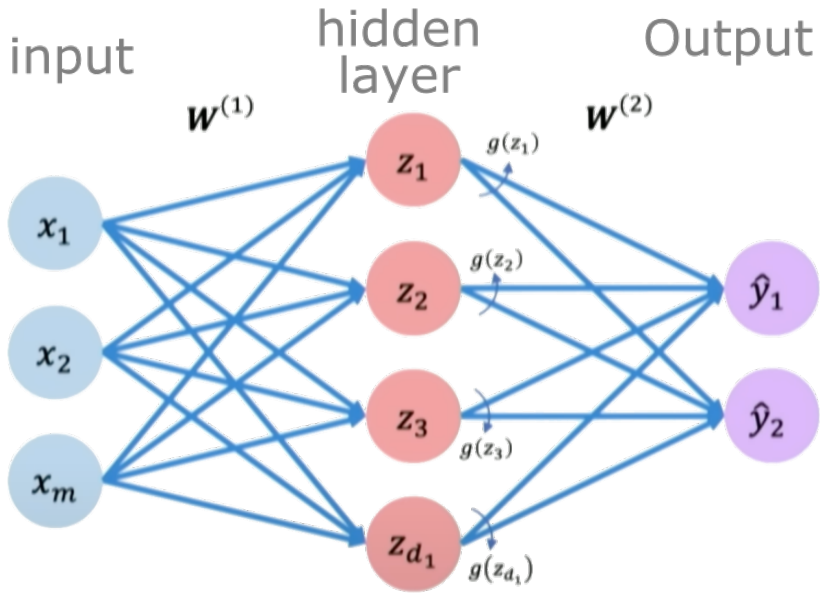
For 1D input x, a single-layer network with N_h hidden neurons:

$$\mathbf{z}^{(1)} = W^{(1)}x + \mathbf{b}^{(1)} \quad \text{(pre-activation)}$$

$$\mathbf{h} = g(\mathbf{z}^{(1)}) \quad \text{(hidden layer output)}$$

$$z^{(2)} = W^{(2)}\mathbf{h} + b^{(2)} \quad \text{(output layer)}$$

$$\hat{y} = z^{(2)} \quad \text{(final prediction)}$$



Single hidden layer neural network

Outline

Training Process

Goal: Find optimal parameters θ^* that minimize loss function.
Loss function (MSE):

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} (u_{NN}(x_i; \theta) - u_i)^2$$

Optimization problem:

$$\theta^* = \arg\min_{\theta} \mathcal{L}(\theta)$$

Training Steps:

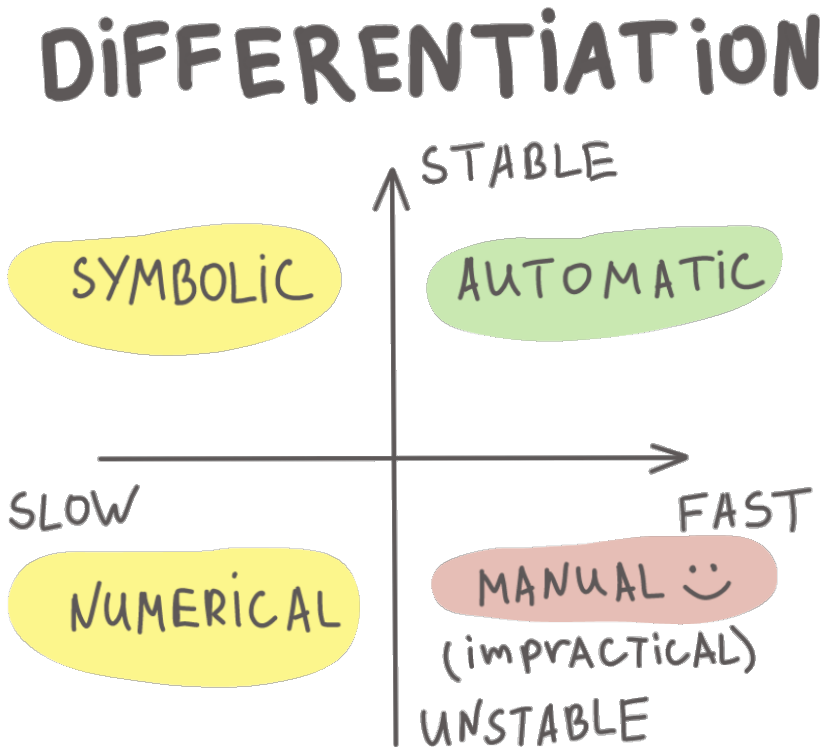
1. Forward pass: compute predictions
2. Calculate loss
3. Backward pass: compute gradients
4. Update parameters
5. Repeat until convergence

The Gradient Problem

Training neural networks requires gradients: $\frac{\partial \mathcal{L}}{\partial \theta}$ for all parameters θ.

Traditional approaches have fundamental flaws:

- **Manual:** Exact, but error-prone and doesn't scale
- **Symbolic:** Exact, but "expression swell"
- **Numerical:** $f'(x) \approx \frac{f(x+h)-f(x-h)}{2h}$
  - Inaccurate (rounding errors)
  - Expensive (multiple evaluations)



For a neural network with thousands of parameters:

$$\theta = \{W^{(1)}, \mathbf{b}^{(1)}, W^{(2)}, \mathbf{b}^{(2)}, \ldots\}$$

We need a fourth approach: Automatic Differentiation!

Automatic Differentiation: The Solution

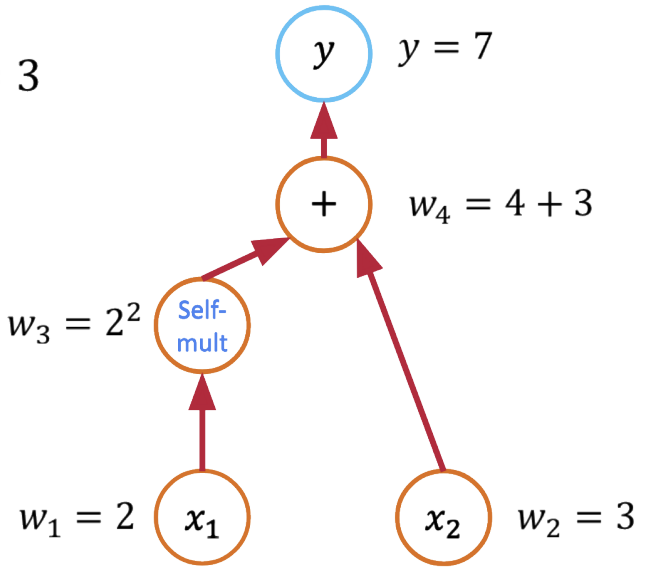
Every function is a computational graph of elementary operations.

Consider: $y = x_1^2 + x_2$

Evaluation Trace:

1. $v_1 = x_1^2$
2. $y = v_1 + x_2$

$x_1 = 2, x_2 = 3$



$y = x_1^2 + x_2$

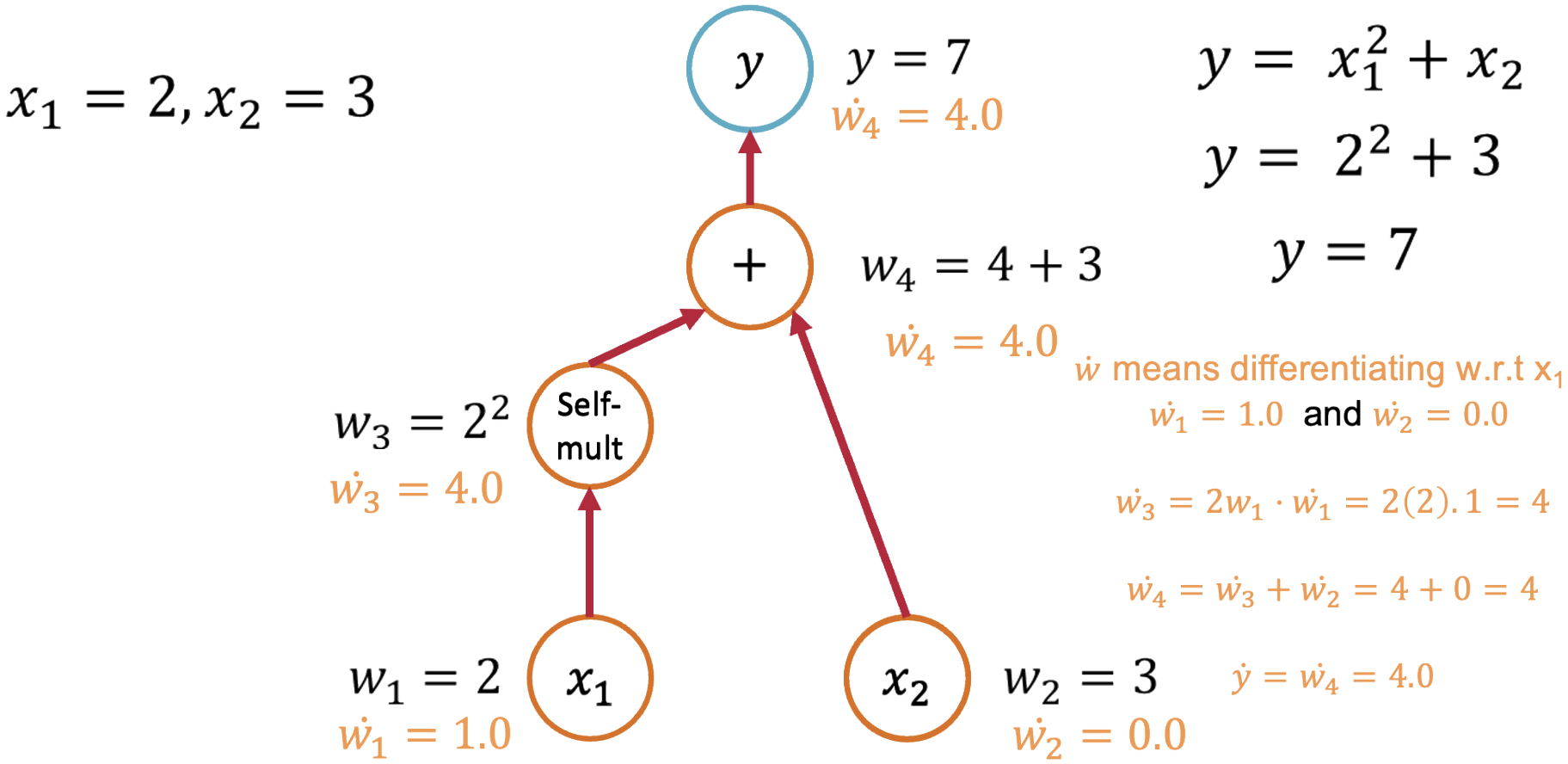
$y = 2^2 + 3$

$y = 7$

This decomposition is the key that makes AD possible. By applying the chain rule to each elementary step, we can compute exact derivatives.

Forward Mode Automatic Differentiation

Forward mode AD propagates derivative information **forward** through the graph, alongside the function evaluation.



▸ AD Demo

Forward Mode: Step-by-Step Example

Let's compute $\frac{\partial y}{\partial x_1}$ for $y = x_1^2 + x_2$.

1. Seed the input w.r.t. x_1

Set $\dot{x}_1 = 1$ and $\dot{x}_2 = 0$.

2. Forward Propagation

- $v_1 = x_1^2 \implies \dot{v}_1 = 2x_1 \cdot \dot{x}_1 = 2x_1 \cdot 1 = 2x_1$
- $y = v_1 + x_2 \implies \dot{y} = \dot{v}_1 + \dot{x}_2 = 2x_1 + 0 = 2x_1$

Result

The final propagated tangent $\dot{y}$ is the derivative: $\frac{\partial y}{\partial x_1} = 2x_1$.

Key Idea: To get the derivative w.r.t. x_2, we would need a *new pass* with seeds $\dot{x}_1 = 0, \dot{x}_2 = 1$.

Reverse Mode Automatic Differentiation

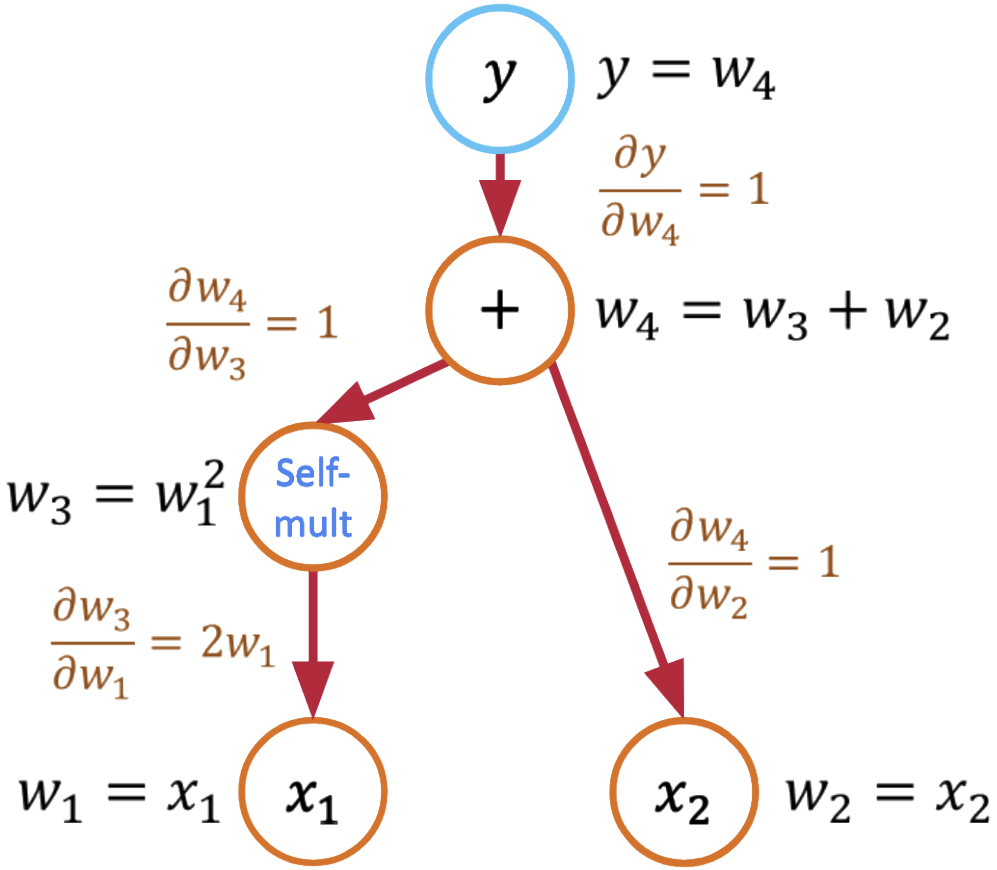
Reverse mode AD (or **backpropagation**) computes derivatives by propagating information **backward** from the output.

Algorithm:

1. **Forward Pass:** Evaluate the function and store all intermediate values.
2. **Backward Pass:** Starting from the output, use the chain rule to propagate "adjoints" ($\bar{v} = \frac{\partial y}{\partial v}$) backward through the graph.

Reverse Mode: One Pass for All Gradients

The backward pass systematically applies the chain rule.
Let's trace the computation for $y = x_1^2 + x_2$:



$$y = x_1^2 + x_2$$

$$\nabla y = \left(\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2}\right)$$

$$\boxed{\nabla y = (2x_1, 1)}$$

We can find the gradient with respect to the output y

Using $x_1 = 2, x_2 = 3$:

$$\nabla y = (4, 1)$$

Reverse mode computes **all** partial derivatives in a single backward pass.

When to Use Forward vs. Reverse Mode

The choice depends on the dimensions of your function $f : \mathbb{R}^n \to \mathbb{R}^m$.

Forward Mode

- **Cost:** One pass per *input* variable.
- **Efficient when:** Few inputs, many outputs ($n \ll m$).
- **Computes:** Jacobian-vector products ($J \cdot v$).

Reverse Mode

- **Cost:** One pass per *output* variable.
- **Efficient when:** Many inputs, few outputs ($n \gg m$).
- **Computes:** Vector-Jacobian products ($v^T \cdot J$).

Machine Learning context: We have millions of parameters (inputs, n) and a single scalar loss function (output, $m = 1$).

Reverse mode is the clear winner!

▸ AD Notebook

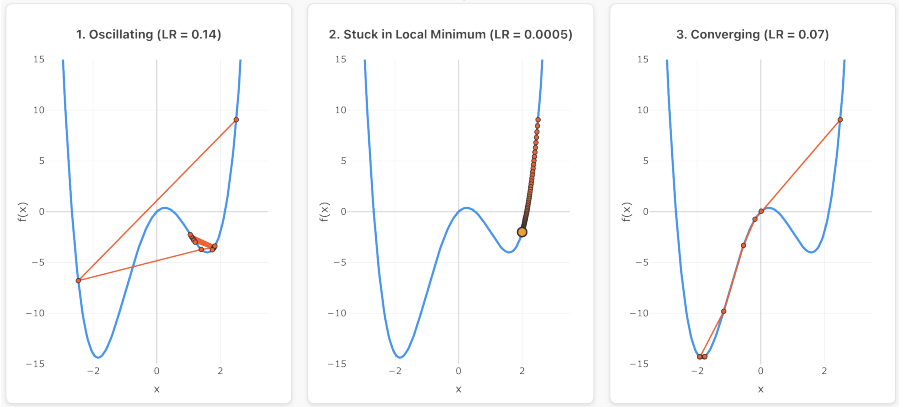
Gradient Descent Algorithm

Basic Algorithm:

1. Initialize weights randomly
2. Loop until convergence:
3. Compute gradient $\frac{\partial \mathcal{L}}{\partial \theta}$
4. Update weights:
   $\theta \leftarrow \theta - \eta \frac{\partial \mathcal{L}}{\partial \theta}$
5. Return weights

$$\theta_{new} = \theta_{old} - \eta \nabla \mathcal{L}(\theta)$$

where η is the learning rate.



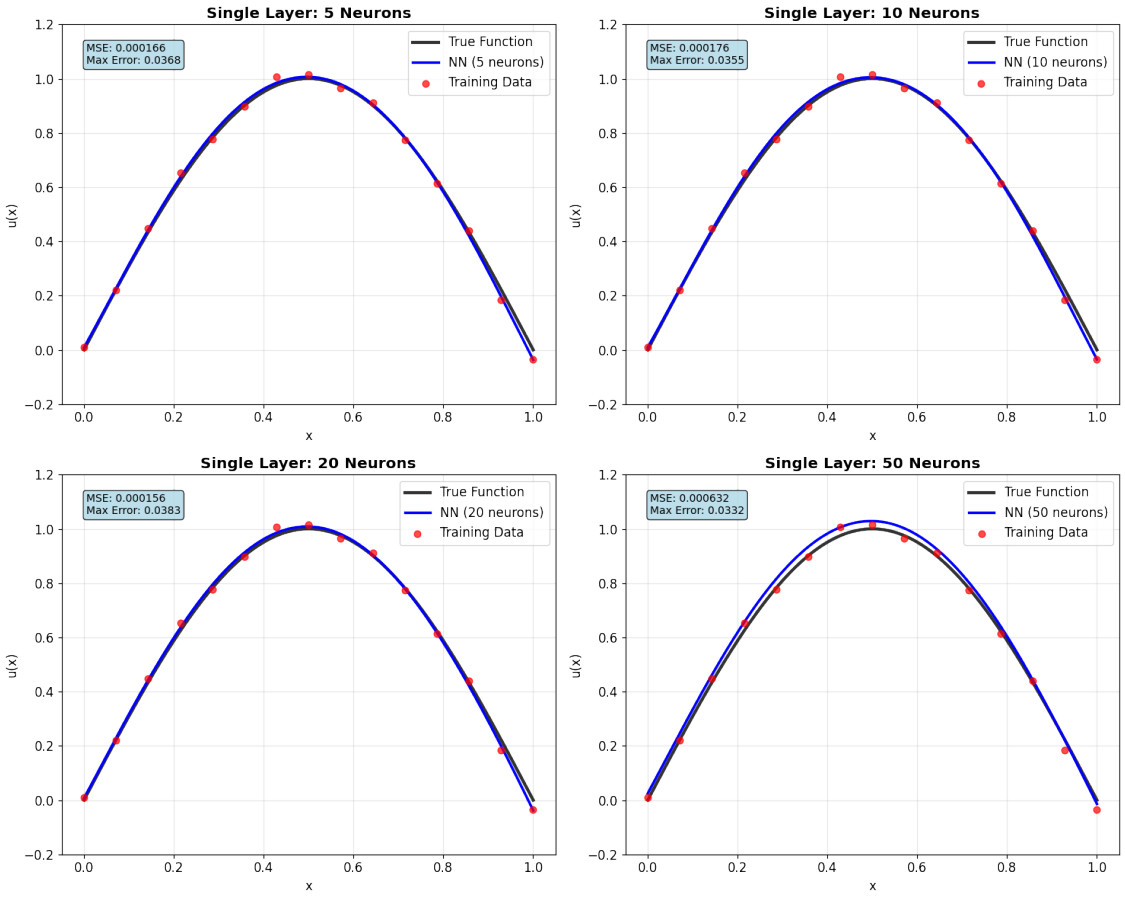
Gradient descent optimization

▸ SGD Demo

Width vs Approximation Quality

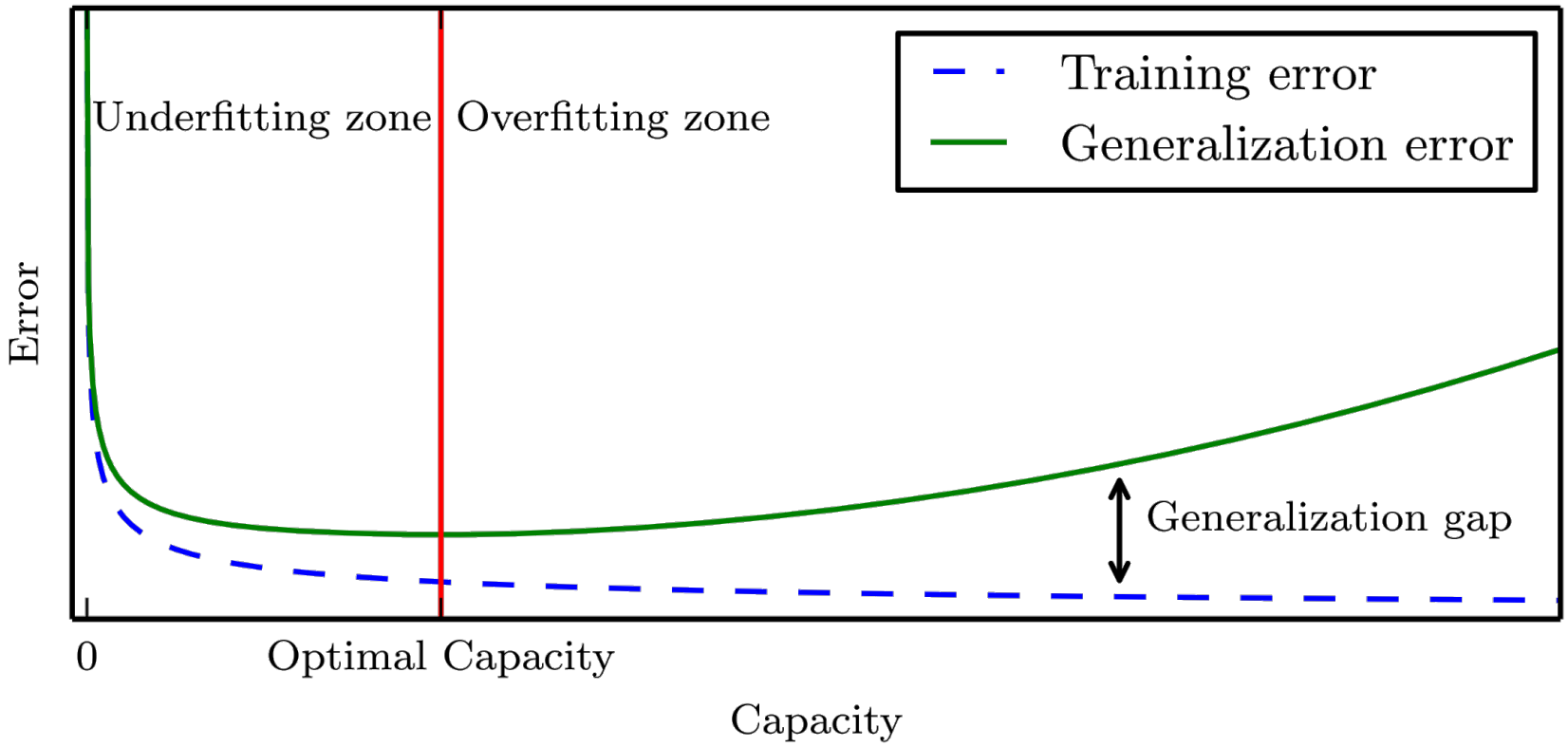
Hypothesis: More neurons $\rightarrow$ better approximation

Experiment: Train networks with 5, 10, 20, 50 neurons



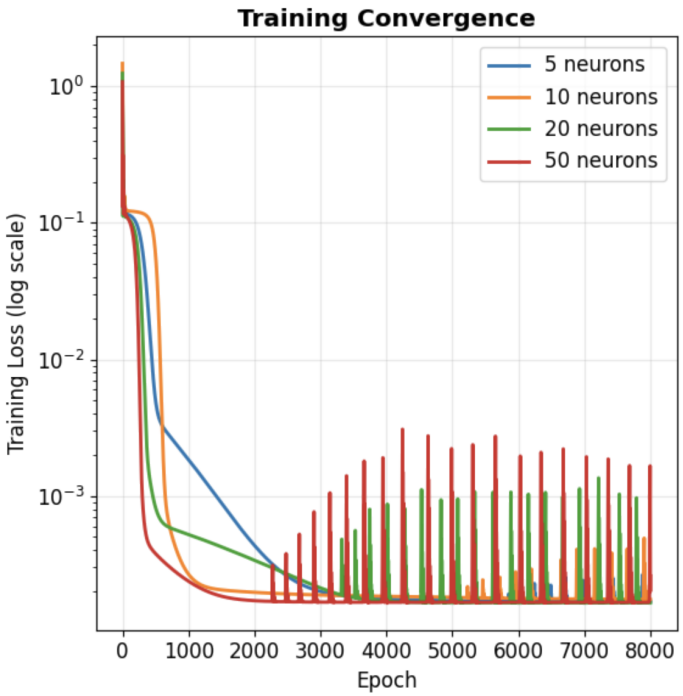
Approximation quality vs network width

Training/Validation



Training and validation convergence

Training Convergence Analysis



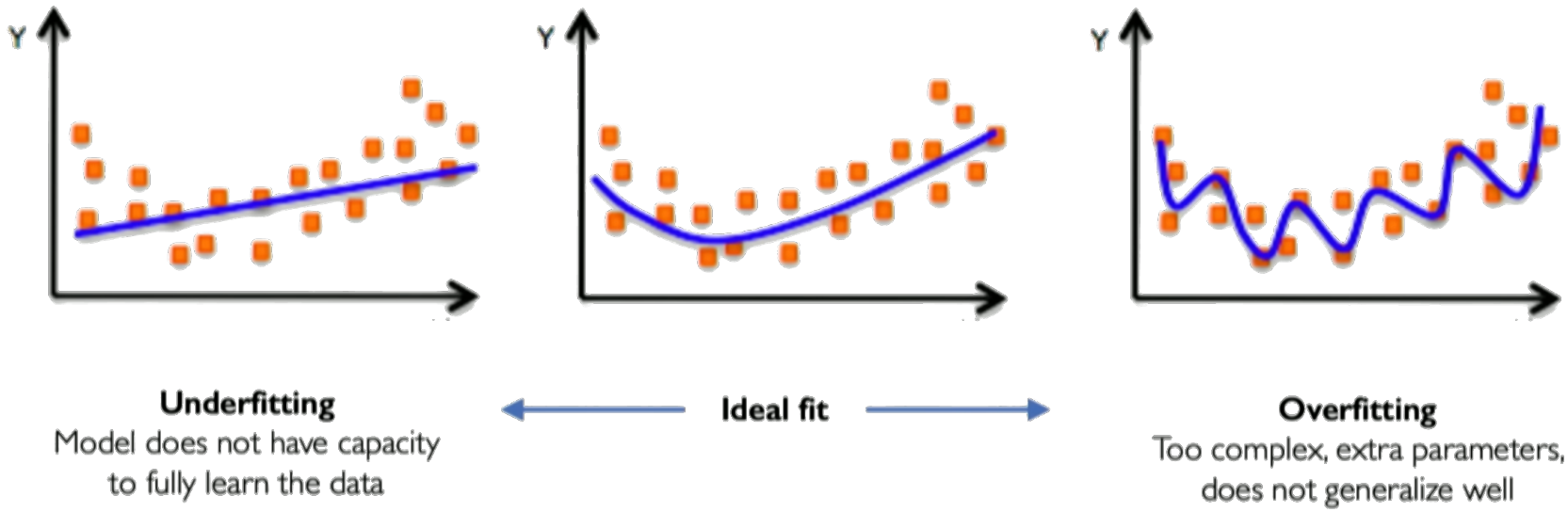
Training convergence and final loss vs width

Key Findings:

- 50 neurons: $\sim$10x better than 5 neurons
- Logarithmic improvement with width
- Convergence rate similar across widths

Overfitting and Model Capacity

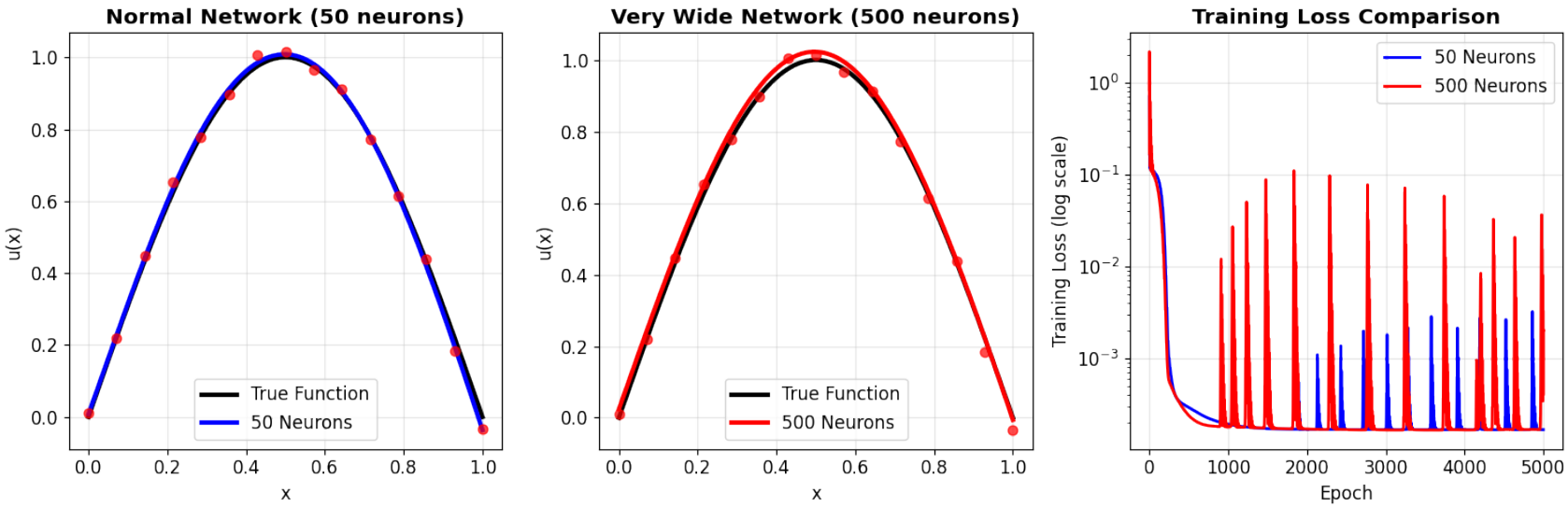
Problem: High-capacity networks can memorize training data.



Under and overfitting illustration

Detection: Monitor validation loss during training.
Solutions: More data, regularization, early stopping, simpler architectures.

Demonstrating Overfitting



Normal (50 neurons) vs very wide (500 neurons) network

Observation: Very wide networks may generalize worse despite lower training loss.

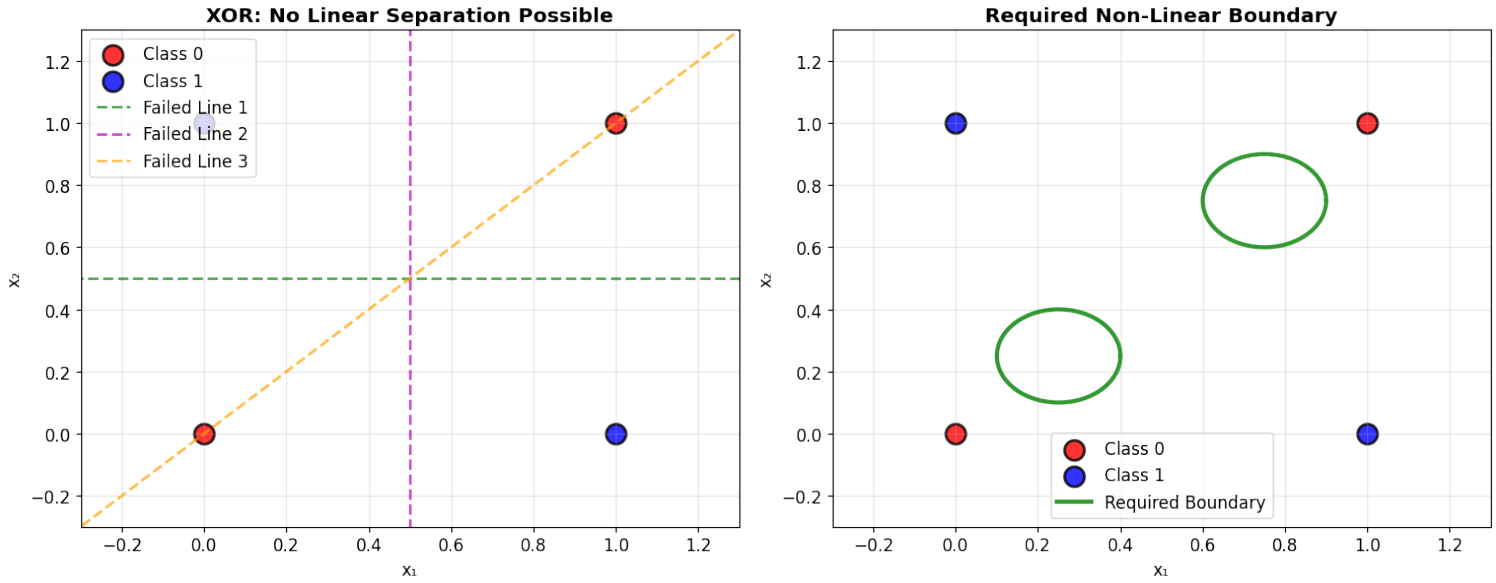
Outline

The XOR Problem: Historical Crisis

XOR Truth Table:

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

The Crisis: No single line can separate these classes!



XOR is not linearly separable

True Single-Layer vs Multi-Layer

Critical Distinction:

- **True Single-Layer:** Input $\rightarrow$ Output (NO hidden layers)
- **Multi-Layer:** Input $\rightarrow$ Hidden $\rightarrow$ Output (1+ hidden layers)

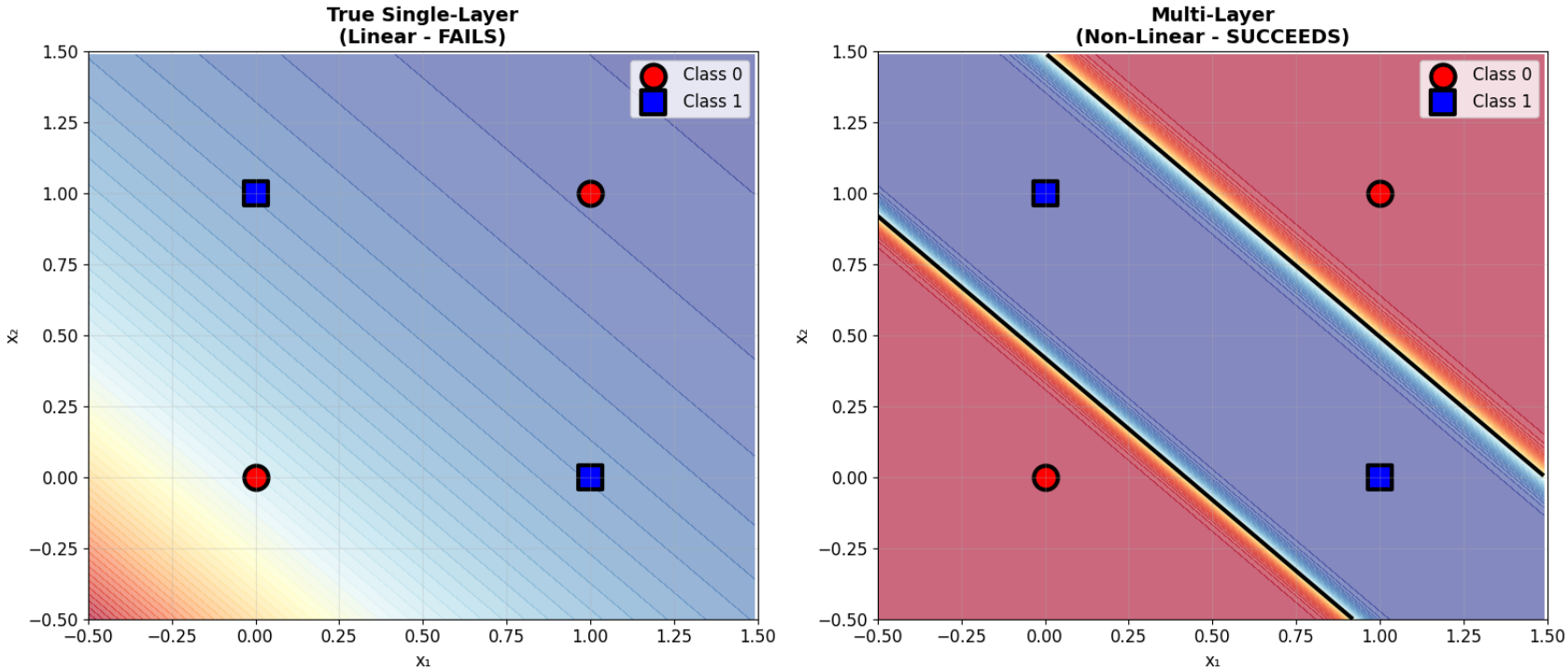
True Single-Layer (fails):

$$y = \sigma(w_1 x_1 + w_2 x_2 + b)$$

Multi-Layer (succeeds):

$$h_1 = \sigma(w_{11} x_1 + w_{12} x_2 + b_1)$$
$$h_2 = \sigma(w_{21} x_1 + w_{22} x_2 + b_2)$$
$$y = \sigma(v_1 h_1 + v_2 h_2 + b_3)$$

XOR Solution: Decision Boundaries



Linear (fails) vs Non-linear (succeeds) decision boundaries

Key Insight: Hidden layers enable curved boundaries through non-linear transformations.

Mathematical Explanation

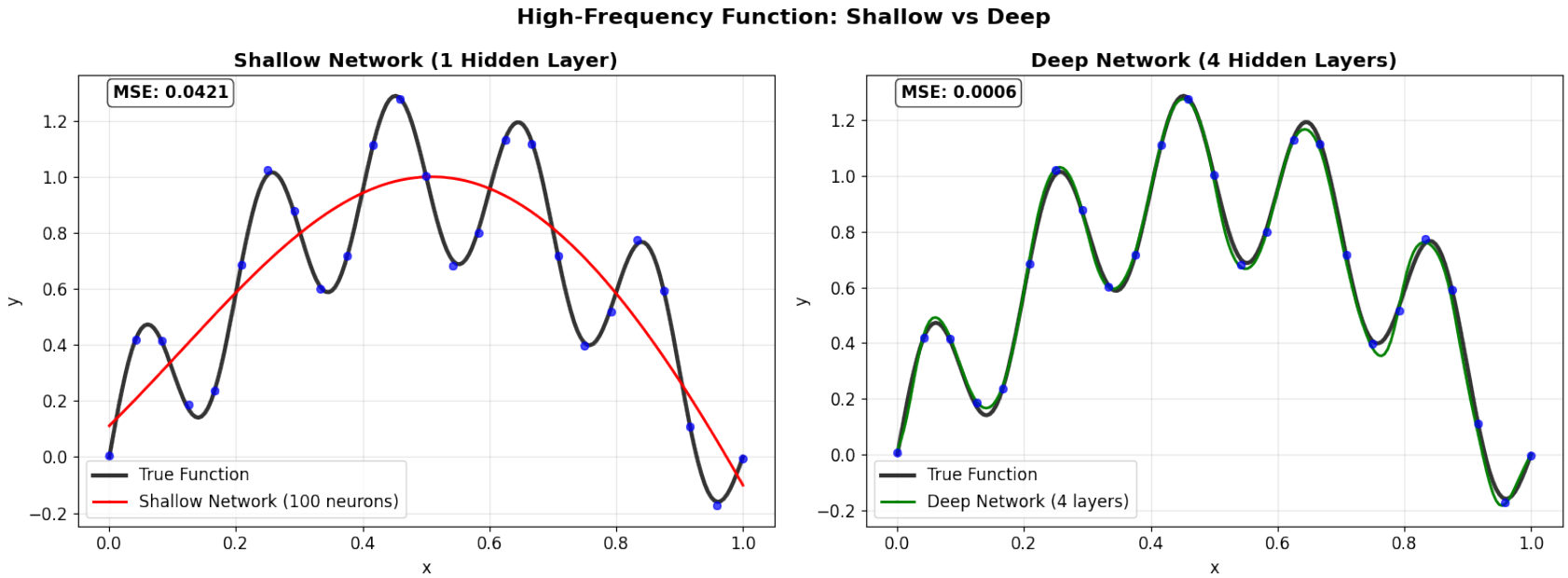
XOR Decomposition:

$$h_1 = \sigma(\text{weights}) \quad (\approx \text{OR gate})$$
$$h_2 = \sigma(\text{weights}) \quad (\approx \text{AND gate})$$
$$y = \sigma(v_1 h_1 + v_2 h_2 + b_3) \quad (\approx \text{OR AND NOT})$$

Result: XOR = (OR) AND (NOT AND) = compositional solution!
General Principle: Complex functions = composition of simple functions.

Beyond XOR: High-Frequency Functions

Test Case: $f(x) = \sin(\pi x) + 0.3\sin(10\pi x)$



Shallow (100 neurons) vs Deep (4 layers, 25 each) networks

Result: Deep networks achieve better performance with fewer parameters.

Historical Timeline: From Crisis to Revolution

Year	Event	Impact
1943	McCulloch-Pitts neuron	Foundation laid
1957	Rosenblatt's Perceptron	First learning success
1969	**Minsky & Papert: XOR**	**Showed limits**
1970s-80s	"AI Winter"	Funding dried up
1986	Backpropagation	Enabled multi-layer training
1989	Universal Approximation	Theoretical foundation
2006+	Deep Learning Revolution	Depth proves essential

Lesson: XOR taught us that depth is necessity, not luxury.

Four Key Insights on Depth

1. **Representation Efficiency**
   - Shallow: May need exponentially many neurons
   - Deep: Hierarchical composition is exponentially more efficient
2. **Feature Hierarchy**
   - Layer 1: Simple features (edges, patterns)
   - Layer 2: Feature combinations (corners, textures)
   - Layer 3+: Complex abstractions (objects, concepts)
3. **Geometric Transformation**
   - Each layer performs coordinate transformation
   - Deep networks "unfold" complex data manifolds
4. **Compositional Learning**
   - Complex functions = composition of simple functions
   - Build complexity incrementally

Summary and Conclusions

Key Takeaways:

- Neural networks learn continuous functions from sparse data
- Nonlinearity is essential for complex function approximation
- Universal Approximation Theorem provides theoretical foundation
- Width increases capacity but depth is more efficient
- Historical XOR problem revealed importance of hidden layers
- Deep networks enable hierarchical feature learning

Applications:

- Scientific computing and PDE solving
- Function approximation and regression
- Pattern recognition and classification
- Physics-informed neural networks (PINNs)

Questions?

Thank you!

Contact:
Krishna Kumar
krishnak@utexas.edu
University of Texas at Austin