Why Non-Linearity?

The Core Problem
Without activation functions:

$$\mathbf{a}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$$
$$\hat{\mathbf{y}} = \mathbf{W}^{(2)}\mathbf{a}^{(1)} + \mathbf{b}^{(2)}$$

Substituting:

$$\hat{\mathbf{y}} = \mathbf{W}^{(2)}(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) + \mathbf{b}^{(2)}$$
$$= (\mathbf{W}^{(2)}\mathbf{W}^{(1)})\mathbf{x} + (\mathbf{W}^{(2)}\mathbf{b}^{(1)} + \mathbf{b}^{(2)})$$
$$= \mathbf{W}'\mathbf{x} + \mathbf{b}'$$

Result: A single linear transformation!

The Solution
Non-linear activation functions:

$$\mathbf{a}^{(l)} = f(\mathbf{z}^{(l)})$$

where f is non-linear.

Why This Works:

- Non-linearity breaks the collapse
- Composition of non-linear functions
- Can approximate any function

Key Insight:
Non-linearity is what makes deep networks "deep" in a meaningful sense.

Non-linearity is essential for learning complex patterns

Linear Networks Collapse

Mathematical Proof
For any number of linear layers:

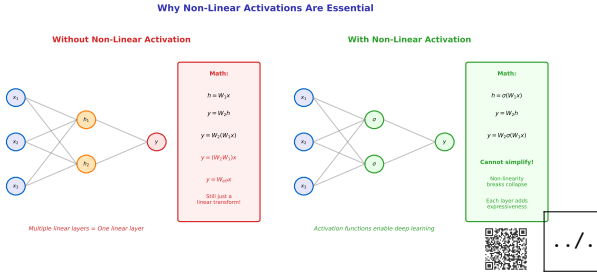
$$\mathbf{y} = \mathbf{W}^{(L)}\mathbf{W}^{(L-1)} \dots \mathbf{W}^{(1)}\mathbf{x}$$

Since matrix multiplication is associative:

$$= (\mathbf{W}^{(L)}\mathbf{W}^{(L-1)} \dots \mathbf{W}^{(1)})\mathbf{x}$$

$$= \mathbf{W}^{\text{eff}}\mathbf{x}$$

Conclusion:
100 linear layers = 1 linear layer.
No benefit from depth without non-linearity.



Stacked linear layers = single linear layer

The Sigmoid Function

Definition

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

Properties:

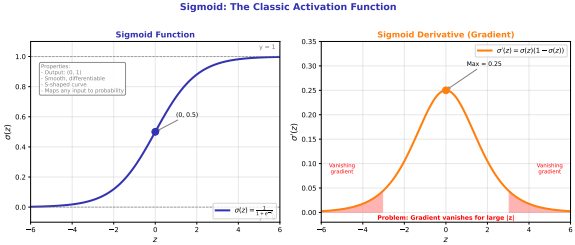
- Range: $(0, 1)$
- Smooth and differentiable
- $\sigma(0) = 0.5$
- Symmetric: $\sigma(-z) = 1 - \sigma(z)$

Derivative:

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

Use Cases:

- Binary classification (output)
- Probability interpretation
- Historical (hidden layers)



../.

sigmoid_functio

The classic activation: squashes to probability

Sigmoid: Properties and Problems

Advantages

+ Bounded output $(0, 1)$
+ Smooth gradient
+ Probability interpretation
+ Historically important

Disadvantages

- **Vanishing gradients**
  For $|z| > 4$: $\sigma'(z) \approx 0$
  Gradients become tiny
  Deep networks can't learn
- Not zero-centered
  All positive outputs
  Zig-zag weight updates
- Computationally expensive
  Requires exp function

The Vanishing Gradient Problem

When z is very positive or negative:

z	$\sigma'(z)$
0	0.25
2	0.10
4	0.018
6	0.0025

Gradients shrink exponentially through layers!

Result: Early layers learn very slowly in deep networks. This limited deep learning until ReLU.

Smooth and bounded, but gradients can vanish

The Tanh Function

Definition

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = 2\sigma(2z) - 1$$

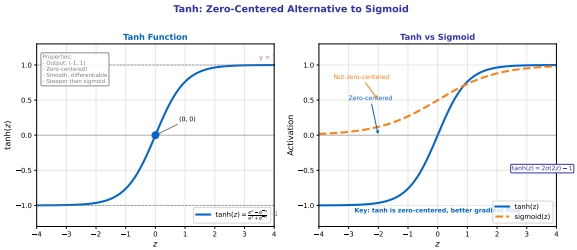
Properties:

- Range: $(-1, 1)$
- Zero-centered
- $\tanh(0) = 0$
- Odd function: $\tanh(-z) = -\tanh(z)$

Derivative:

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

Advantage over Sigmoid:
Zero-centered outputs lead to more stable gradient updates.



../.

tanh_functio

Zero-centered: range (-1, 1)

ReLU: Rectified Linear Unit

Definition

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

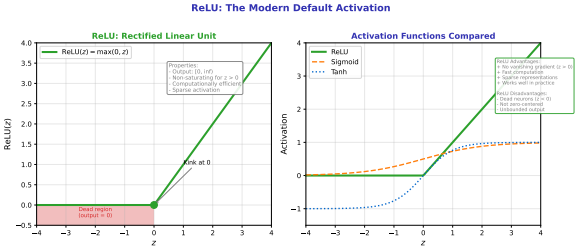
Properties:

- Range: $[0, \infty)$
- Not bounded above
- Not differentiable at $z = 0$
- Piecewise linear

Derivative:

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

The Modern Default for hidden layers.



../.

relu_functio

Simple but powerful: the modern default

Why ReLU Works So Well

Advantages

+ **No vanishing gradient**
  Gradient is 1 for $z > 0$
  Signal propagates through layers
+ **Computationally cheap**
  Just comparison and assignment
  No exponentials
  6x faster than sigmoid
+ **Sparse activation**
  Many neurons output 0
  Efficient representation
+ **Biological plausibility**
  Neurons can be "off"

Disadvantages

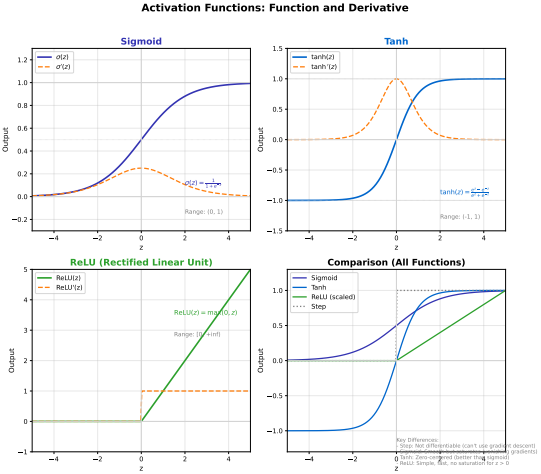
- **"Dying ReLU" problem**
  If $z < 0$ always: gradient $= 0$
  Neuron never updates
  Can "die" permanently
- Not zero-centered
- Unbounded (can explode)

Variants:

- Leaky ReLU: $\max(0.01z, z)$
- ELU: z if $z > 0$, $\alpha(e^z - 1)$ otherwise
- GELU: used in transformers

Cheap to compute, gradients don't vanish (for positive inputs)

Activation Functions: Comparison



../.

activation_compariso

Different functions for different problems

Discussion Question

"Which activation function would you use for: (a) predicting stock returns, (b) buy/sell classification? Why?"

Consider:

(a) Stock Returns (Regression)

- Output: continuous value
- Can be positive or negative
- Hidden: ReLU or tanh
- Output: **Linear (none)**
- Returns are unbounded

(b) Buy/Sell (Classification)

- Output: probability $\in (0, 1)$
- Two mutually exclusive classes
- Hidden: ReLU
- Output: **Sigmoid**
- Or softmax for multi-class

Think-Pair-Share: 3 minutes

Choosing the Right Activation

Hidden Layer Guidelines

Default: ReLU

- Works well in most cases
- Fast and stable

If dying ReLU: Leaky ReLU

- Small negative slope
- Prevents dead neurons

For RNNs: Tanh

- Bounded outputs help stability
- Zero-centered

Output Layer Guidelines

Task	Activation
Binary class	Sigmoid
Multi-class	Softmax
Regression	Linear
Bounded regression	Sigmoid/tanh
Positive only	ReLU

Finance Examples:

- Return prediction: Linear
- Direction prediction: Sigmoid
- Sector classification: Softmax
- Volatility: ReLU or Softplus

Output layer choice depends on your problem type

The Fundamental Question

How Powerful Are Neural Networks?
We've seen that MLPs can:

- Solve XOR (non-linear patterns)
- Combine features hierarchically
- Learn from data

But a Deeper Question:
Are there functions that MLPs fundamentally *cannot* represent?

Or can they approximate *anything*?

Why This Matters

If MLPs are limited:

- Need to check if problem is solvable
- Architecture constraints matter
- Some patterns impossible

If MLPs are universal:

- Architecture is not the bottleneck
- Challenges are elsewhere (data, training)
- Theoretical guarantee of capability

Spoiler: MLPs are universal approximators!

Just how powerful are neural networks?

Universal Approximation Theorem

The Theorem (Informal)
A feedforward network with:

- One hidden layer
- Sufficient hidden neurons
- Non-linear activation (e.g., sigmoid)

can approximate any continuous function on a compact domain to arbitrary accuracy.

Key Contributors:

- Cybenko (1989): sigmoid
- Hornik (1991): general activations
- Further extensions since

Formal Statement
Let $f : [0,1]^n \to \mathbb{R}$ be continuous.
For any $\epsilon > 0$, there exists an MLP $\hat{f}$ with:

$$|\hat{f}(\mathbf{x}) - f(\mathbf{x})| < \epsilon$$

for all $\mathbf{x} \in [0,1]^n$.

In Plain English:
No matter how complex the pattern, an MLP with enough hidden neurons can match it as closely as you want.

With enough hidden neurons, you can approximate any continuous function

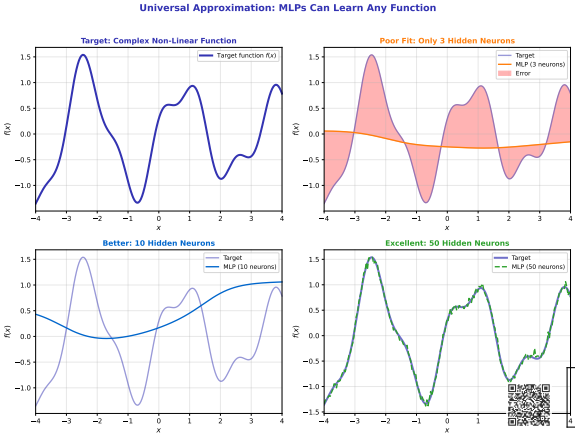
What Universal Approximation Means

The Good News

- No function is “too complex”
- MLPs are theoretically complete
- Architecture is not the limit
- One hidden layer is enough (in theory)

Visual Intuition:
Each hidden neuron contributes a “bump” or “step.”
With enough bumps, you can approximate any shape.

Think of it like approximating a curve with many small line segments.



universal_approximation_dem

Universal Approximation Theorem (Cybenko, 1989): A feedforward network with a single hidden layer containing a finite number of neurons can approximate any continuous function on compact subsets of $\mathbb{R}^n$

More neurons = better approximation

What It Doesn't Mean

Common Misconceptions

"Any network can learn anything"

No. Need enough neurons

- May need exponentially many

"Training will find the solution"

No. Theorem is about existence

- Says nothing about finding weights
- Optimization may fail

"One layer is always enough"

cally no.

- Deep networks often more efficient
- Fewer parameters for same accuracy

The Gap: Existence vs Construction

The theorem says:
"A good approximation exists."
It does NOT say:

- How many neurons you need
- How to find the right weights
- How much data is required
- How long training takes
- Whether it will generalize

Analogy:
"There exists a needle in this haystack" doesn't help you find it.

Existence of a solution does not mean we can find it

Theory vs Practice

Theoretical Guarantees
Universal approximation says:

- Given infinite neurons: perfect fit
- Given infinite data: find the function
- Given infinite compute: optimize

Practical Reality
We have:

- Finite neurons: limited capacity
- Finite data: must generalize
- Finite compute: approximate solutions

What Matters More in Practice

1. **Data quality and quantity**
   - More important than architecture
2. **Regularization**
   - Prevent overfitting
3. **Optimization**
   - Finding good weights
4. **Generalization**
   - Performance on new data

Module 3 will address these practical challenges.

Universal approximation is necessary but not sufficient

Implications for Finance

The Optimistic View
If markets have patterns, MLPs can learn them:

- Non-linear relationships? Possible.
- Complex interactions? Possible.
- Hidden factors? Possible.

Theoretical Capability:
"An MLP could, in principle, capture any market pattern."

The Realistic View

Challenges Remain:

- Signal-to-noise ratio is low
- Markets are non-stationary
- Past patterns may not repeat
- Data is limited (especially for crashes)
- Overfitting is easy

The EMH Counterargument:
If markets are efficient, there's nothing systematic to learn.

Module 4 will explore this tension.

In theory, yes. In practice, many challenges remain.

Why Loss Functions?

Learning Requires an Objective
To train a neural network, we need:

1. A way to measure errors
2. A number that decreases as we improve
3. A signal for weight updates

The Loss Function:
$\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$ measures how wrong our predictions are.

Goal of Training:
Find weights that minimize $\mathcal{L}$.

Finance Analogy

Profit & Loss (P&L):

- Measures trading performance
- Negative P&L = bad trades
- Optimize to maximize P&L

Loss Function:

- Measures prediction errors
- High loss = bad predictions
- Optimize to minimize loss

Note: "Loss" is the opposite of "profit" – we minimize loss!

To learn, we must measure mistakes

Mean Squared Error (MSE)

Definition

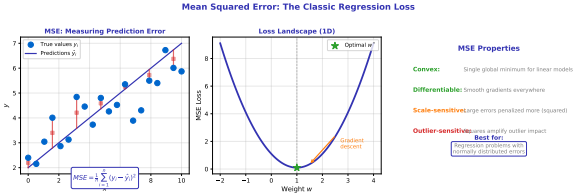
$$\mathcal{L}_{\text{MSE}} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Properties:

- Always non-negative
- Zero only if perfect predictions
- Penalizes large errors heavily
- Differentiable everywhere

Use Case:

- Regression problems
- Predicting continuous values
- Stock returns, prices, etc.



../.

mse_visualizatio

The standard loss for predicting continuous values

Cross-Entropy Loss

Binary Cross-Entropy

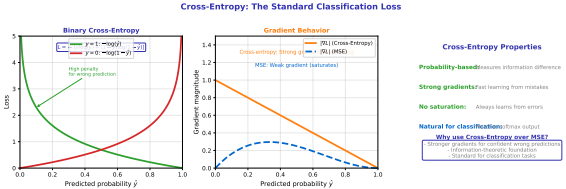
$$\mathcal{L}_{\text{BCE}} = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)]$$

Properties:

- For probability outputs
- Heavily penalizes confident wrong answers
- Connected to information theory

Use Case:

- Classification problems
- Buy/sell decisions
- Any yes/no prediction



../.

cross_entropy_visualizatio

The standard loss for classification

The Loss Landscape

Loss as a Function of Weights

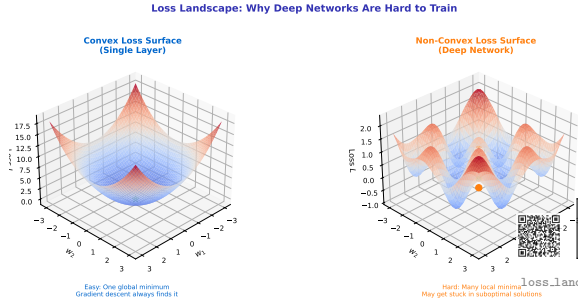
$$\mathcal{L}(\mathbf{W}, \mathbf{b})$$

For every choice of weights, there's a loss value.

The Landscape:

- High regions: bad weights
- Low regions: good weights
- Global minimum: best weights
- Local minima: traps

Training =
Finding the lowest point in this landscape.



Training = finding the lowest point in this landscape

Finance: Choosing Your Loss

Task-Specific Loss Functions

Task	Loss
Return prediction	MSE
Direction prediction	Cross-entropy
Volatility forecast	MSE
Multi-class sector	Categorical CE

Beyond Standard Losses:

- Sharpe ratio optimization
- Asymmetric losses (penalize losses more than gains)
- Custom finance metrics

Important Consideration

MSE vs Business Metric:
A model with low MSE may still lose money!

Example:

- Predict returns with 5% MSE
- But wrong on big moves
- Transaction costs eat profits
- Risk-adjusted return is poor

Lesson:
Statistical accuracy $\neq$ Trading profitability
Module 4 explores this gap.

Different problems, different loss functions

Module 2: Key Takeaways

What We Learned

1. **Historical Context**
   - AI Winter (1969-1982)
   - Backprop renaissance (1986)
   - Right idea + right time
2. **MLP Architecture**
   - Hidden layers find patterns
   - Matrix notation for computation
   - Parameter counting
3. **Activation Functions**
   - Non-linearity is essential
   - ReLU for hidden, task-specific for output
4. **Universal Approximation**
   - MLPs can learn any function
   - But existence $\neq$ construction
5. **Loss Functions**
   - MSE for regression
   - Cross-entropy for classification
   - Loss landscape visualization

The Big Picture:
We now have powerful architectures. But how do they *learn*?

From single perceptron to universal function approximator

Preview: Module 3

"We have the architecture. But how does it LEARN?"

The Missing Piece

We know:

- How to compute forward pass
- What loss functions measure
- That good weights exist

We don't know:

- How to find good weights
- How errors update weights
- How to avoid overfitting

Coming in Module 3:

- Gradient descent (intuition)
- Backpropagation (the magic)
- Training dynamics
- Overfitting and regularization
- Practical training tips

The Key: Backpropagation – the algorithm that made deep learning possible.

Mathematical details: See Appendix B (Backpropagation Derivation)

Next: The magic of backpropagation