Big Data in Finance

Week 7: Neural Networks

Dr Daniele Bianchi

Spring 2026

Today's Agenda

From What We Know to Neural Networks

The Single Neuron: Building Block

Building a Network: Multiple Neurons

How Neural Networks Learn

Regularization: Preventing Overfitting

Revisiting Market Timing

Out-of-Sample Performance

Neural Networks for Classification

When Do Neural Networks Help?

Summary and Next Steps

From What We Know to Neural Networks

Recap: Our Toolkit So Far

We've learned several approaches to prediction:

- **Linear methods** (Week 2): Ridge, Lasso, Elastic Net
  - Global model: same coefficients everywhere
  - Smooth predictions, easy to interpret
- **Tree-based methods** (Week 3): Random Forests, XGBoost
  - Local models: different rules in different regions
  - Capture interactions automatically
- **Classification** (Week 4): Logistic regression, tree ensembles
  - Predicting probabilities (e.g., default risk)
  - The sigmoid function "squashes" outputs to $[0, 1]$

Today's Question

Is there a method that combines the best of both worlds?

The Gap We Want to Fill

	Linear Models	Trees
Smooth predictions	✓	×
Captures non-linearity	×	✓
Finds interactions automatically	×	✓
Works well with many features	✓ (with regularization)	✓
Extrapolates beyond training data	✓	×

What if we want:

- Smooth predictions (like linear models)
- But also captures complex non-linear patterns (like trees)
- And learns which interactions matter from the data

Neural networks offer this flexibility—but with trade-offs we'll discuss.

A Financial Motivation

Return prediction: Complex patterns that simple models might miss

- **Non-linear effects:** Dividend yield might predict returns differently at extreme values vs. normal values
- **Interactions:** The effect of valuation ratios might depend on the level of interest rates
- **Regime dependence:** Predictability might differ in recessions vs. expansions

Credit risk: Complex borrower profiles

- Default risk depends on combinations of factors
- A borrower with high income *and* high debt is different from one with high income *and* low debt
- Trees capture this, but predictions are "blocky"

The Promise

Neural networks can learn **smooth, non-linear functions** from data

The Single Neuron: Building Block

You Already Know This! (Almost)

Recall logistic regression from Week 4:

$$P(\text{default}) = \sigma\left(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots\right)$$

where $\sigma(z) = \frac{1}{1+e^{-z}}$ is the **sigmoid function**

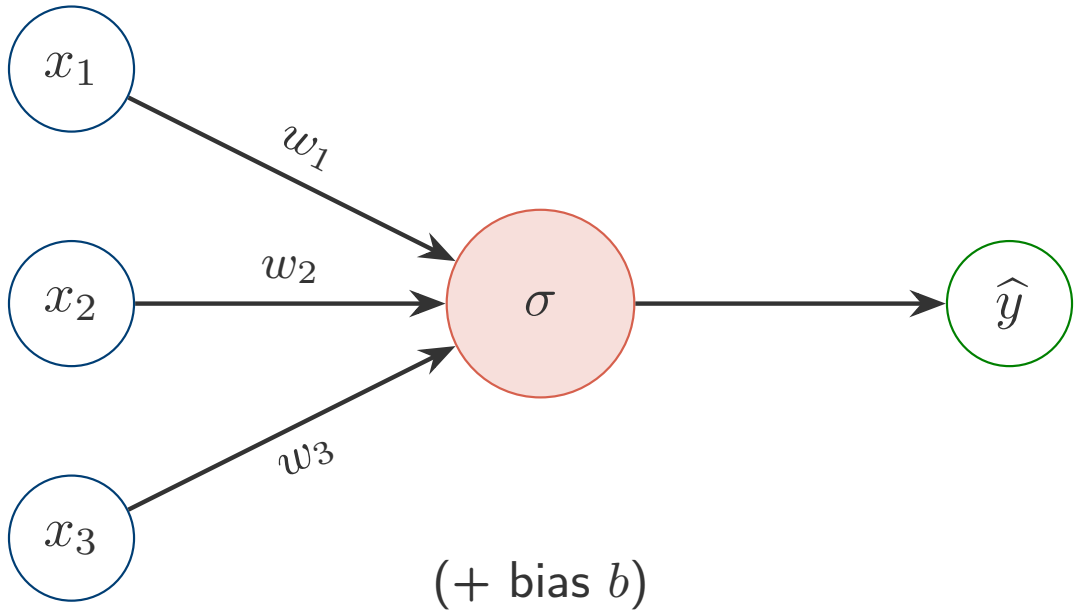
What logistic regression does:

1. Take a **weighted sum** of inputs: $z = \beta_0 + \beta_1 x_1 + \cdots$
2. Pass it through a **non-linear function** (sigmoid)
3. Output a value between 0 and 1

Key Insight

A single **neuron** does exactly this! Logistic regression *is* a one-neuron neural network.

The Single Neuron



A neuron computes:

$$\widehat{y} = \sigma \underbrace{(w_1 x_1 + w_2 x_2 + w_3 x_3 + b)}_{\text{weighted sum + bias}}$$

- **Weights** (w_1, w_2, w_3): How much each input matters (like β coefficients)
- **Bias** (b): Shifts the decision threshold (like intercept β_0)
- **Activation function** (σ): Introduces non-linearity

Why Do We Need the Non-Linear Function?

Without the activation function:

$$\text{Neuron output} = w_1 x_1 + w_2 x_2 + \cdots + b$$

This is just linear regression!

The problem: Stacking linear functions gives... another linear function

$$f(g(x)) = f(ax + b) = c(ax + b) + d = (ca)x + (cb + d)$$

Still linear! No matter how many layers, you get a linear model.

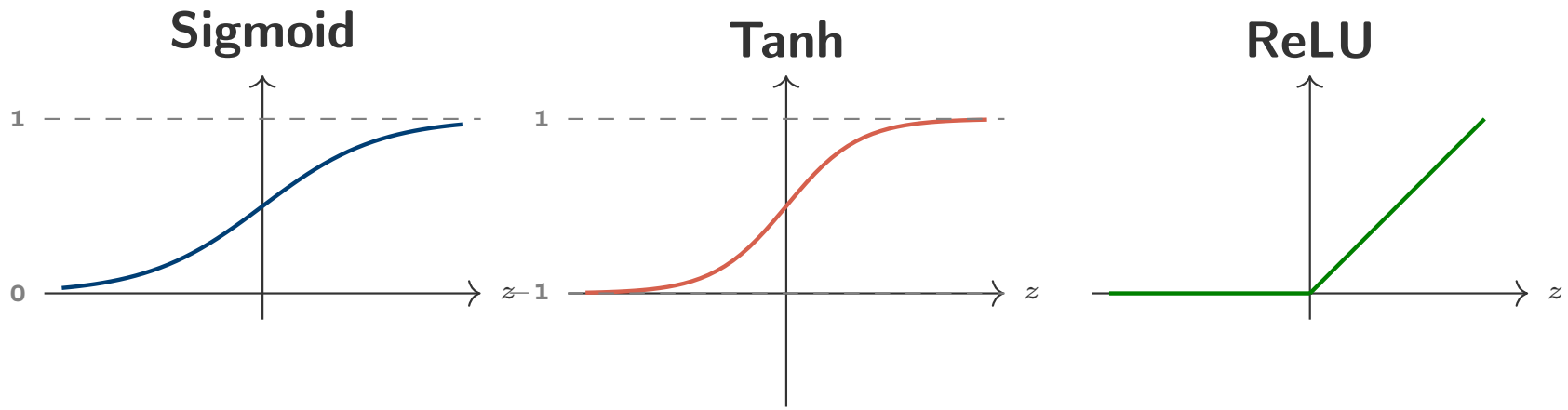
The solution: Non-linear activation functions "break" the linearity

- Allow the network to learn curved decision boundaries
- Enable approximation of complex functions

Key Point

The activation function is what makes neural networks powerful. Without it, we'd just have a fancy linear regression.

Common Activation Functions



Function	Formula	Notes
Sigmoid	$\frac{1}{1+e^{-z}}$	Output in $(0, 1)$; used in logistic regression
Tanh	$\frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$	Output in $(-1, 1)$; centered around 0
ReLU	$\max(0, z)$	Modern default; simple and effective

For this course: We'll primarily use **ReLU** (Rectified Linear Unit)—it's the most common choice in practice.

ReLU: The Workhorse of Modern Neural Networks

ReLU is surprisingly simple:

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

Why is this so popular?

- *Computationally simple:* Just check if input is positive
- *Avoids "vanishing gradients":* A technical problem we'll mention later
- *Sparsity:* Many neurons output zero, making networks efficient

Intuition: ReLU acts like a "gate"

- If the weighted input is positive: pass it through
- If the weighted input is negative: block it (output zero)

This allows the network to "turn on" different neurons for different inputs, creating complex patterns.

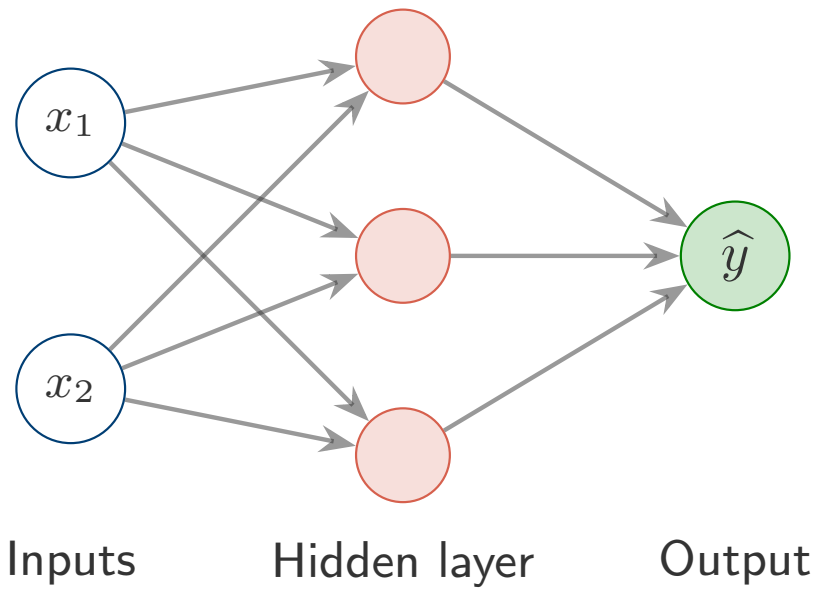
Building a Network: Multiple Neurons

One Neuron Isn't Enough

A single neuron with sigmoid = logistic regression

- Can only learn linear decision boundaries
- Limited expressive power

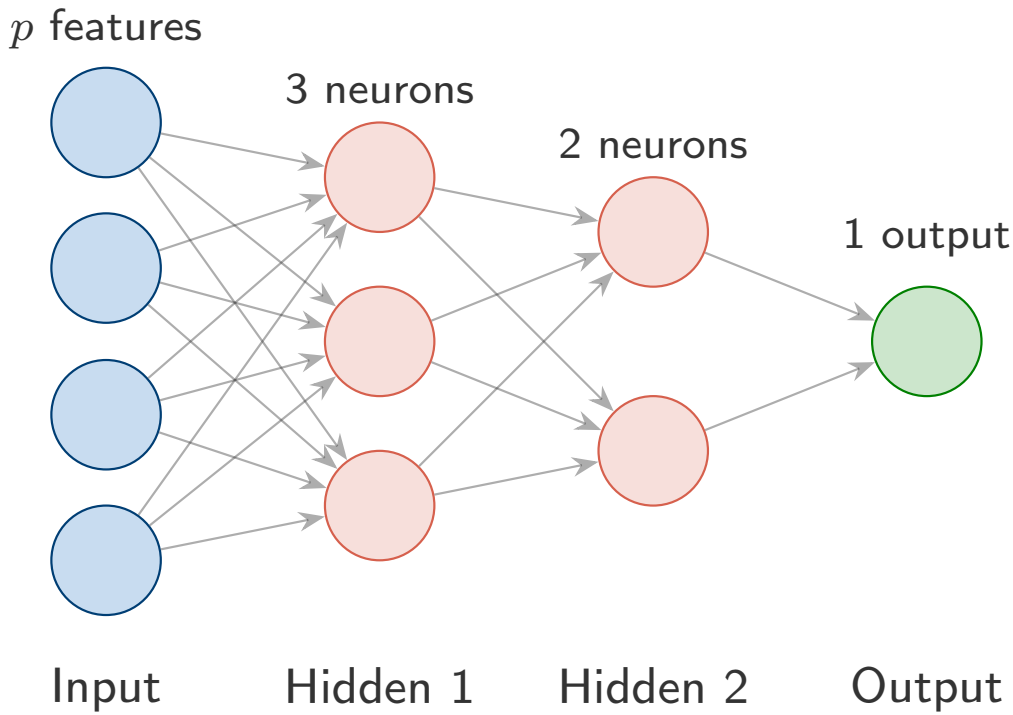
The solution: Combine multiple neurons



Each hidden neuron learns to detect a different pattern in the data.

The Multi-Layer Perceptron (MLP)

Architecture: Stack neurons into layers



Terminology:

- *Input layer:* Your features $(x_1, \ldots, x_p)$
- *Hidden layers:* Where the "learning" happens
- *Output layer:* The prediction
- *Fully connected:* Neurons between layers are all connected

How Information Flows (Forward Propagation)

Example: A network with one hidden layer of 3 neurons

Step 1: Input to hidden layer

$$h_1 = \text{ReLU}(w_{11}x_1 + w_{12}x_2 + b_1)$$
$$h_2 = \text{ReLU}(w_{21}x_1 + w_{22}x_2 + b_2)$$
$$h_3 = \text{ReLU}(w_{31}x_1 + w_{32}x_2 + b_3)$$

Step 2: Hidden layer to output

$$\widehat{y} = v_1 h_1 + v_2 h_2 + v_3 h_3 + c$$

What's happening:

- Each hidden neuron computes a **different combination** of inputs
- The output combines these learned "features"
- The network learns *what* combinations are useful

An Economic Intuition

Think of hidden neurons as "derived features":

Credit risk example:

- Input: income, debt, credit score
- Hidden neuron 1 might learn: "debt-to-income stress"
- Hidden neuron 2 might learn: "credit utilization pattern"
- Hidden neuron 3 might learn: "income stability indicator"

Return prediction example:

- Input: D/P, term spread, inflation
- Hidden neuron 1 might learn: "valuation regime"
- Hidden neuron 2 might learn: "monetary policy stance"
- Hidden neuron 3 might learn: "risk appetite indicator"

Key Insight

The network **automatically** constructs useful features from raw inputs. You don't have to manually engineer interaction terms or non-linear transformations.

Why Depth Matters

Adding more layers allows more complex functions:

- **1 hidden layer:** Can approximate any continuous function (in theory)
- **More layers:** Can represent the same function more **efficiently**

Intuition: Hierarchical feature learning

- Layer 1: Learn simple patterns (e.g., "is debt-to-income high?")
- Layer 2: Combine simple patterns (e.g., "high DTI *and* low income")
- Layer 3: Even more complex combinations

But for financial data:

- We typically have limited observations (decades, not millions)
- Very deep networks can overfit badly
- **1–2 hidden layers** often sufficient

More is not always better!

The Universal Approximation Theorem

A remarkable theoretical result:

Universal Approximation (informal)

A neural network with a *single* hidden layer containing enough neurons can approximate *any* continuous function to arbitrary accuracy.

What this means:

- Neural networks are extremely flexible
- In principle, they can learn any pattern in the data

What this does NOT mean:

- We can *find* the right weights (optimization is hard)
- We have *enough data* to learn the true function
- The network will *generalize* well to new data

"Can approximate" $\neq$ "Will approximate well in practice"

How Neural Networks Learn

The Learning Problem

We have many parameters to learn:

- Weights connecting every neuron to the next layer
- Biases for each neuron

Example: Network with 10 inputs, one hidden layer of 50 neurons, 1 output

- Input to hidden: $10 \times 50 = 500$ weights + 50 biases
- Hidden to output: $50 \times 1 = 50$ weights + 1 bias
- **Total: 601 parameters**

The goal: Find values for all these parameters that minimize prediction error

Just like before, we minimize a loss function:

- Regression: Mean Squared Error (MSE)
- Classification: Cross-entropy (related to log-likelihood in logistic regression)

No Closed-Form Solution

Unlike Ridge regression, we can't just solve an equation

Ridge	**Neural Network**
Linear in parameters	Non-linear in parameters
Closed-form: $(X'X + \lambda I)^{-1}X'y$	No closed-form solution
One global optimum	Many local optima

The solution: Iterative optimization

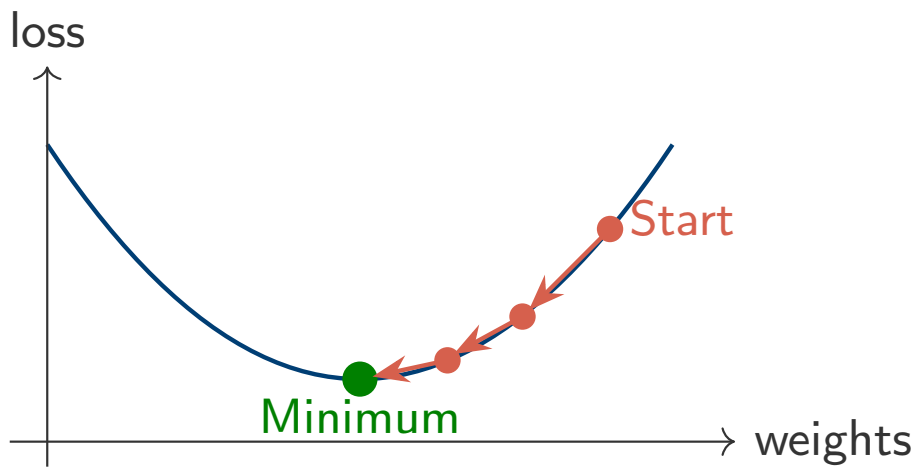
1. Start with random initial weights
2. Compute predictions and errors
3. Adjust weights to reduce error
4. Repeat until convergence

This is called **gradient descent**.

Gradient Descent: The Intuition

Imagine you're lost in mountains at night and want to reach the lowest point:

1. Feel the slope of the ground beneath your feet
2. Take a step in the downhill direction
3. Repeat



Gradient descent: "Gradient" = slope; "Descent" = go downhill

The Gradient Descent Update Rule

At each step, update each weight:

$$w^{\text{new}} = w^{\text{old}} - \eta \cdot \frac{\partial \text{Loss}}{\partial w}$$

Components:

- $\frac{\partial \text{Loss}}{\partial w}$: The gradient (slope) — tells us which direction is "downhill"
- η: The **learning rate** — how big a step to take
- Minus sign: We want to go *down*hill

The learning rate is crucial:

- Too large: Overshoot the minimum, might never converge
- Too small: Takes forever to converge
- Just right: Converges smoothly to a good solution

We'll see how to choose this in sklearn next week.

Backpropagation: Computing the Gradients

The challenge: How do we compute $\frac{\partial \text{Loss}}{\partial w}$ for all weights?

The answer: Backpropagation (short for "backward propagation of errors")

1. **Forward pass:** Compute predictions by passing inputs through the network
2. **Compute loss:** Compare predictions to true values
3. **Backward pass:** Use the chain rule to compute how each weight contributed to the error

Good News

You don't need to implement this! sklearn (and all neural network libraries) compute gradients automatically. Just understand the intuition.

Intuition: Errors at the output "propagate back" through the network, telling each weight how much it contributed to the mistake.

Stochastic Gradient Descent (SGD)

Problem: Computing gradients over the entire dataset is slow

Solution: Use a random subset ("mini-batch") at each step

Method	**Description**
Batch GD	Use all data points for each update (slow but stable)
Stochastic GD	Use one random point per update (fast but noisy)
Mini-batch GD	Use a small batch (e.g., 32 points) — best of both worlds

Why mini-batches work:

- Approximate the true gradient well enough
- Much faster per iteration
- Some noise can actually help escape local minima

sklearn's default solver ("adam") uses an advanced version of SGD.

Modern Optimizers: Adam

Plain gradient descent has limitations:

- Same learning rate for all parameters
- Can get stuck or oscillate

Adam (Adaptive Moment Estimation) is the modern standard:

- **Adapts learning rate** for each parameter individually
- Parameters with consistently large gradients: smaller effective learning rate
- Parameters with small gradients: larger effective learning rate
- Also uses "momentum": considers recent gradient history

For This Course

Use `solver='adam'` (sklearn's default). It's robust and works well in most cases. You don't need to understand the details.

Regularization: Preventing Overfitting

The Overfitting Problem

Neural networks have many parameters — perfect for overfitting!

Example:

- 20 input features
- One hidden layer with 100 neurons
- $20 \times 100 + 100 + 100 \times 1 + 1 = 2{,}201$ parameters
- With only a few hundred observations... trouble!

Symptoms of overfitting:

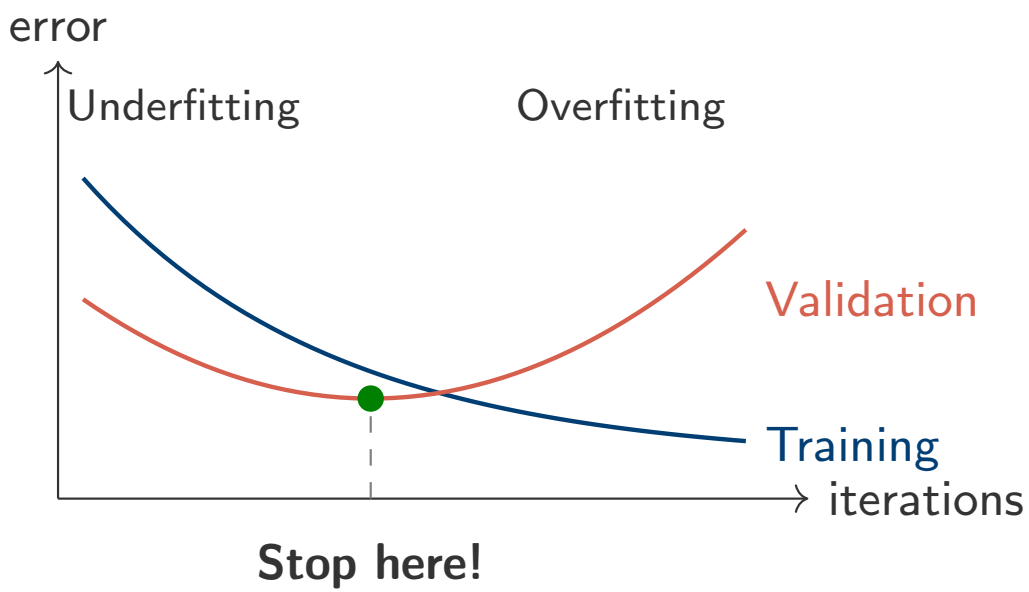
- Training loss is very low
- Validation/test loss is much higher
- Model memorizes training data instead of learning patterns

Especially problematic in finance:

- Limited historical data
- Low signal-to-noise ratio
- Patterns may change over time

Regularization Strategy 1: Early Stopping

Observation: Validation error often increases after a point



Early stopping: Stop training when validation error starts increasing

In sklearn: `early_stopping=True` monitors a validation set and stops automatically.

Regularization Strategy 2: Architecture Choices

Simpler networks = fewer parameters = less overfitting

Choices that affect complexity:

- **Number of hidden layers:** More layers = more complex
- **Neurons per layer:** More neurons = more parameters

Rules of thumb for financial applications:

- Start with 1 hidden layer
- Number of neurons: between p (number of features) and $2p$
- Add complexity only if validation performance improves

Practical Advice

In finance, simpler networks often work as well as complex ones. Don't assume "deep" is better — it rarely is with limited data.

Putting It All Together: Hyperparameters

Key choices when using neural networks:

Hyperparameter	**Guidance**
Hidden layers	Start with 1, rarely need more than 2 for finance
Neurons per layer	Start with $\sim$50–100, tune via CV
Activation	ReLU for hidden layers; linear/sigmoid for output
Learning rate	Start with default (0.001); Adam adapts it
Regularization (α)	Tune via cross-validation (e.g., 0.0001 to 0.1)
Early stopping	Almost always use it

Validation is essential: Use time-series cross-validation (no shuffling!) to select hyperparameters.

Feature Scaling: Critical for Neural Networks

Unlike trees, neural networks are sensitive to feature scales

Why?

- Gradient descent works better when features are on similar scales
- Large-scale features dominate the learning process
- Activation functions can saturate with extreme values

Solution: Standardize your features

$$x_j^{\text{scaled}} = \frac{x_j - \bar{x}_j}{s_j}$$

where $\bar{x}_j$ is the mean and s_j is the standard deviation

Important

- Compute mean and std on **training data only**
- Apply the same transformation to validation/test data
- sklearn's `StandardScaler` handles this

Revisiting Market Timing

Recap: What We Learned in Part I

Neural networks: the key ideas

- Neurons: weighted sum $\rightarrow$ non-linear activation $\rightarrow$ output
- Multi-layer perceptrons (MLPs): stack neurons into layers
- Learning: gradient descent minimizes a loss function
- Regularization: Early stopping, architecture choice

Key advantages over previous methods:

- Smooth predictions (unlike trees)
- Automatic non-linearities and interactions (unlike linear models)
- Universal approximation capability

Today's Question

Can neural networks improve on Ridge and Random Forest for **market timing**?

Benchmarks from Previous Weeks

What we found so far (Weeks 2–3):

Method	Sharpe Ratio	Ann. Return (%)
Buy & Hold	0.46	6.9
Ridge	0.64	10.3
Random Forest	0.47	7.0

Key findings:

- Market timing is **very hard**
- Ridge provides meaningful improvement over buy-and-hold
- Random Forest only marginally better than passive
- Statistical accuracy $\neq$ economic performance

Today's question: Can neural networks do better?

The Promise of Neural Networks

Potential advantages:

1. **Smooth non-linearity:** Unlike trees, MLPs produce smooth functions
2. **Flexible interactions:** Network learns which combinations matter
3. **Universal approximation:** Can represent any continuous function

Potential disadvantages:

- Many parameters $\rightarrow$ overfitting risk
- Requires careful tuning (architecture, regularization)
- Computationally more expensive
- Black box: hard to interpret

Empirical question: Do the advantages outweigh the disadvantages for market timing with limited data?

Out-of-Sample Performance

Implementation: Methods Compared

Linear methods (from Week 2):

- Ridge, Lasso, Elastic Net

Tree-based methods (from Week 3):

- Random Forest, Gradient Boosting (XGBoost)

Neural network (new):

- MLPRegressor with one hidden layer

Common setup:

- Data: GWZ (2024) — 25 predictors, monthly 1956–2021
- Walk-forward prediction (expanding window)
- Initial training: 120 months
- Hyperparameters: Time-series cross-validation

From Predictions to Portfolios

Recall from Weeks 2–3: Mean-variance timing weights

$$\omega_t = \frac{1}{\gamma} \cdot \frac{\widehat{r}_{t+1}}{\widehat{\sigma}_t^2}$$

- $\widehat{r}_{t+1}$: Predicted excess return (from each model)
- $\widehat{\sigma}_t^2$: Rolling variance estimate (60-month window)
- $\gamma = 5$: Risk aversion
- Weights constrained to $[-0.5, 1.5]$

Portfolio return:

$$r_{t+1}^{portfolio} = \omega_t \cdot r_{t+1}^{market} + (1 - \omega_t) \cdot r_t^f$$

Benchmark: Buy-and-hold ($\omega_t = 1$ always)

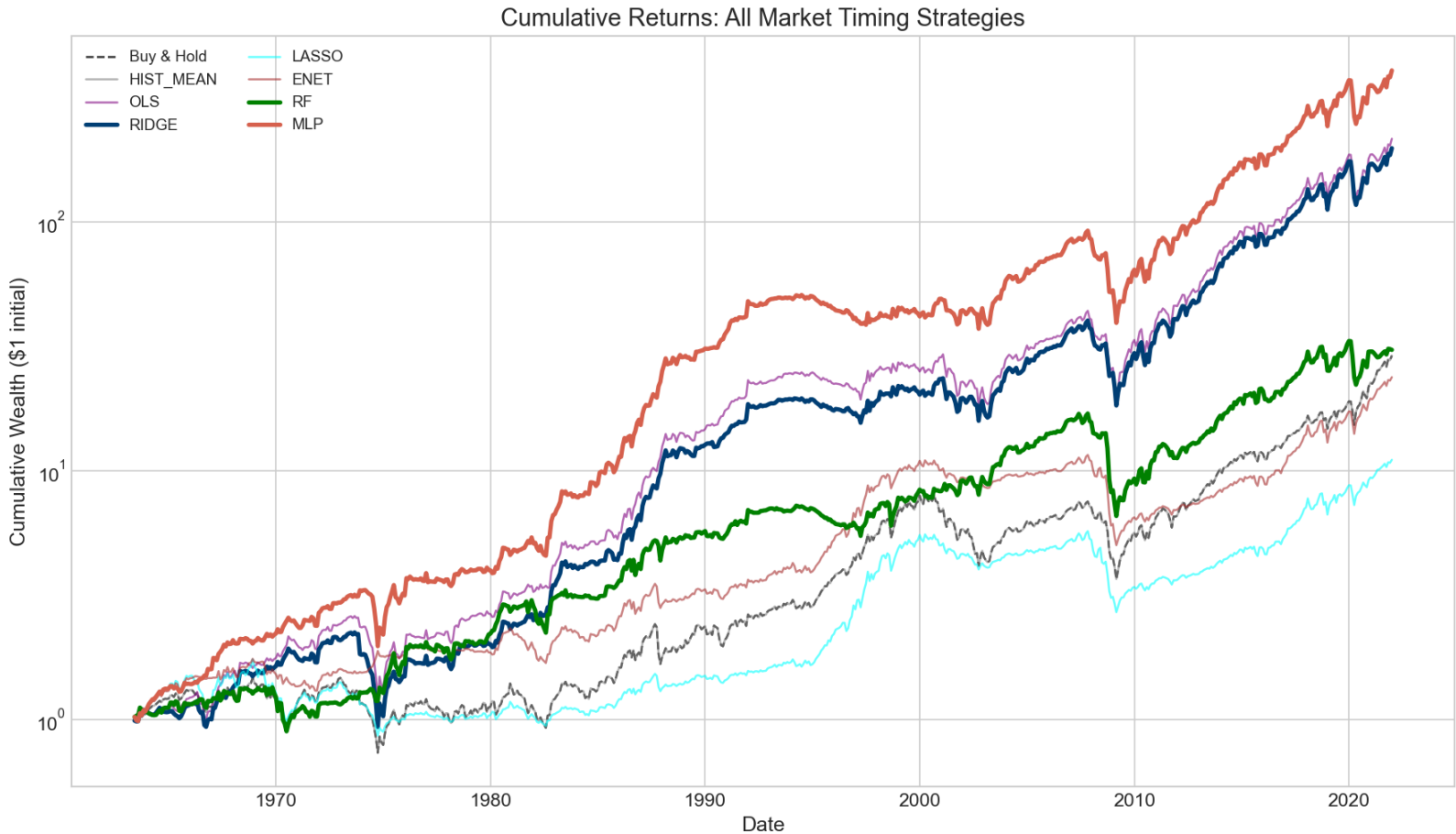
The Main Result: MLP Dominates Economically

Method	R^2_{OS} **(%)**	**Sharpe Ratio**	**Ann. Return (%)**
Buy & Hold	—	0.46	6.86
OLS	-13.85	0.65	10.50
Ridge	-11.94	0.64	10.33
Lasso	-0.03	0.41	4.81
Elastic Net	$+1.02$	0.53	6.08
Random Forest	-0.75	0.47	6.97
MLP	-0.68	**0.74**	**11.48**

Key puzzle: MLP has **slight negative** R^2_{OS} but achieves the **highest Sharpe ratio**!

Statistical accuracy $\neq$ *Economic value*

Cumulative Wealth (Gross)



Cumulative wealth from \$1 initial investment. Log scale. MLP (red) clearly dominates throughout the sample.

The R^2_{OS} vs. Sharpe Ratio Puzzle

How can MLP have negative R^2_{OS} but the best Sharpe ratio?

1. **R^2_{OS} measures average squared error**
   - Penalizes large errors heavily (squared!)
   - A few big mistakes hurt R^2 a lot
2. **Sharpe ratio depends on when you're right**
   - Being right when it matters most (big moves) is valuable
   - Small errors when market is flat don't hurt much
3. **Direction matters more than magnitude**
   - Predicting $+1\%$ when actual is $+5\%$ still makes money
   - Getting the **sign right** is what counts for portfolios

Lesson

Always evaluate models on their **intended use case**. For trading, economic metrics matter more than R^2.

Interpreting the Results

Key observations:

1. MLP achieves the **best performance** across all metrics
   - Highest Sharpe ratio (0.74)
   - Highest total return (40,381%)
   - Consistent outperformance throughout the sample

2. Ridge is a strong second
   - Sharpe of 0.64 — still excellent
   - Much simpler to implement and tune

3. Random Forest disappoints
   - Sharpe of 0.47 — barely beats buy-and-hold
   - Trees' piecewise-constant predictions may not suit this problem

4. Lasso underperforms
   - Too aggressive variable selection?

Turnover Analysis

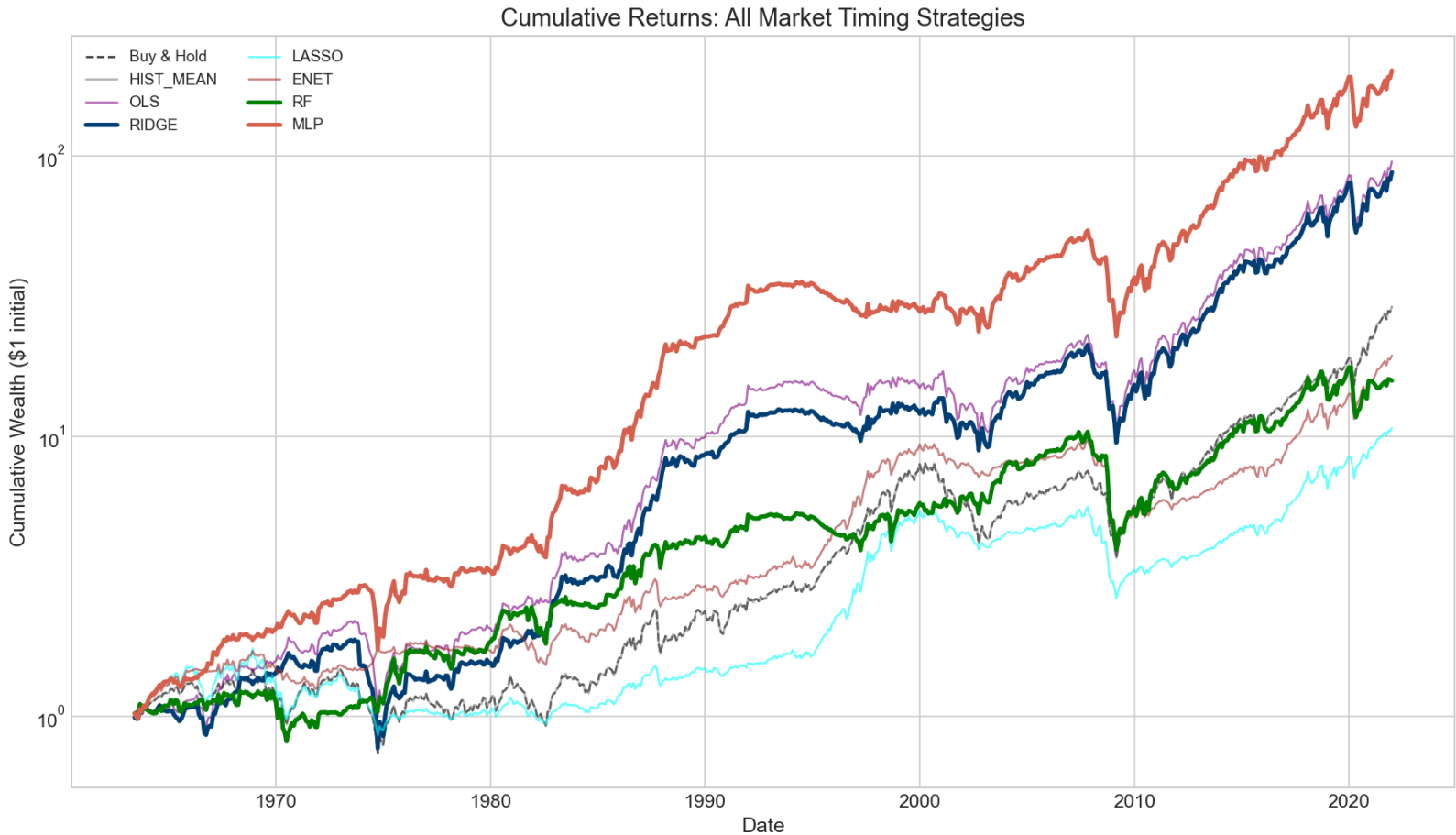
Average monthly weight change (%):

Strategy	Avg. Turnover (%/month)
Buy & Hold	0.0
Historical Mean	0.0
Lasso	1.8
Elastic Net	11.6
Random Forest	37.8
MLP	**40.1**
Ridge	46.6
OLS	46.6

Key observation: MLP has **lower turnover** than OLS/Ridge

- Smooth predictions $\rightarrow$ gradual weight adjustments
- Less sensitive to transaction costs than linear methods

Robustness: After Transaction Costs (50 bps)



Net of 50 bps transaction costs per rebalancing. MLP remains dominant.

Performance After Transaction Costs

Strategy	Turnover	Sharpe (Gross)	Sharpe (Net)	Δ
Buy & Hold	0.0	0.46	0.46	0.00
Lasso	1.8	0.41	0.40	-0.01
Elastic Net	11.6	0.53	0.50	-0.03
Random Forest	37.8	0.47	0.39	-0.08
MLP	40.1	**0.74**	**0.66**	-0.08
Ridge	46.6	0.64	0.56	-0.08
OLS	46.6	0.65	0.56	-0.09

Observations:

- MLP remains best even after 50 bps costs (Sharpe: 0.66)
- Higher turnover $\rightarrow$ larger Sharpe degradation
- Random Forest falls *below* buy-and-hold after costs!

Why Does MLP Outperform?

Potential explanations:

1. **Smooth non-linearity:**
   - MLP captures non-linear relationships smoothly
   - Trees produce "blocky" predictions $\rightarrow$ higher turnover
2. **Flexible interactions:**
   - Network learns which predictor combinations matter
   - More efficient than manual specification
3. **Regularization works well:**
   - L2 penalty + early stopping prevents overfitting
   - Similar principle to why Ridge beats OLS
4. **Lower turnover:**
   - Smooth predictions $\rightarrow$ gradual weight changes
   - Robust to transaction costs

A Caveat: Maximum Drawdowns

All timing strategies have significant drawdowns:

Strategy	Max Drawdown (%)
Buy & Hold	-54.3
Lasso	-52.7
Elastic Net	-56.6
OLS	-56.4
MLP	-57.4
Ridge	-58.3
Random Forest	-61.2

Key point: Higher returns come with similar or *higher* drawdowns

- Timing strategies lever up in good times
- When wrong, losses are magnified
- Real-world implementation requires risk management

Neural Networks for Classification

MLPClassifier: Credit Risk Application

Same architecture, different output:

	MLPRegressor	**MLPClassifier**
Output layer activation	Linear (identity)	Sigmoid / Softmax
Loss function	MSE	Cross-entropy
Output	Predicted value	Probability

sklearn interface is nearly identical:

- Same `hidden_layer_sizes`, `alpha`, `solver`
- Use `predict_proba()` to get probabilities
- `class_weight` **not available** — unlike Random Forest / Logistic

Handling Class Imbalance with MLPClassifier

Problem: `MLPClassifier` doesn't have `class_weight` parameter

Solutions:

1. **Resampling:**
   - Oversample minority class (SMOTE)
   - Undersample majority class
   - Use `imbalanced-learn` library
2. **Threshold tuning:**
   - Train with default threshold (0.5)
   - Adjust classification threshold based on cost trade-offs
   - Choose threshold that maximizes F1 or business objective
3. **Custom loss:** More advanced — requires leaving sklearn

For your projects: threshold tuning is often the simplest effective approach.

Expected Results for Credit Risk

Based on the literature and typical findings:

Method	AUC (typical range)
Logistic Regression	0.70–0.75
Random Forest	0.72–0.77
XGBoost	0.73–0.78
MLP	0.72–0.77

Key points:

- MLPClassifier typically **competitive but not dominant**
- With large datasets (millions of loans), MLPs can match boosting
- Interpretability remains a challenge for regulatory applications
- Probability calibration often needed (use `CalibratedClassifierCV`)

When Do Neural Networks Help?

Evidence from the Finance Literature

Gu, Kelly & Xiu (2020, *RFS*):

- Compared many ML methods for stock return prediction
- Neural networks: competitive but not always best
- Gains most pronounced with **many features** and **large samples**

Chen, Pelger & Zhu (2024, *Management Science*):

- Deep learning for asset pricing
- Benefits emerge with **cross-sectional data** (many stocks)
- Time-series prediction (market timing): modest improvements

Goyal, Welch & Zafirov (2024, *RFS*):

- Comprehensive comparison for equity premium prediction
- "No method consistently dominates"
- Simple methods often as good as complex ones

Conditions Favoring Neural Networks

Our results suggest MLPs help when:

- ✓ Relationships are **non-linear but smooth**
- ✓ Interactions between predictors matter
- ✓ Regularization is applied carefully
- ✓ Turnover costs are a concern (smooth predictions help)

Simpler methods may suffice when:

- × Interpretability is required (regulatory, audit)
- × Computational resources are very limited
- × Quick iteration/debugging is needed

Revised View

With proper regularization, MLPs *can* outperform in financial time-series — but the gains depend on careful implementation.

Computational Considerations

Neural networks are more expensive:

Method	Training Time	Tuning Complexity
Ridge	Very fast	1 hyperparameter
Random Forest	Moderate	2–3 hyperparameters
XGBoost	Moderate	4–5 hyperparameters
MLP	Slower	3–5 hyperparameters

Practical implications:

- Walk-forward with frequent refitting can be slow
- Hyperparameter search is more expensive (architecture choices)
- For production: consider computational budget

In our GWZ application: MLP takes 5–10× longer than Ridge to tune and fit.

Caveats and Considerations

Before concluding "MLPs always win":

1. **In-sample optimization:**
   - Results may partly reflect lucky hyperparameter choices
   - Different random seeds can give different results
2. **Structural breaks:**
   - Patterns learned in 1960s–1990s may not persist
   - MLP's flexibility could adapt or could overfit
3. **Implementation risk:**
   - More moving parts = more things that can go wrong
   - Ridge is harder to "break"
4. **Interpretability:**
   - "Why is the model bullish today?"
   - Regulators and risk managers need answers

Strong results deserve scrutiny — especially in finance!

Summary and Next Steps

Key Takeaways from Today

1. **MLP achieves the best performance for market timing**
   - Sharpe of 0.74 (gross), 0.66 (net) vs. 0.46 for buy-and-hold
   - Terminal wealth 14× higher than passive investing
2. **Smooth non-linearity matters**
   - MLP captures non-linear patterns with low turnover
   - Trees' piecewise-constant predictions create high turnover
3. **Regularization is essential**
   - Simple architecture + early stopping prevents overfitting
   - **Always scale features** with StandardScaler
4. **Results are robust to transaction costs**
   - MLP remains best after 50 bps costs
   - Random Forest hurt most by turnover

Readings

Readings:

- James et al. (2023), *ISLP*, Chapter 10 *[Recomm.]*
- Gu, Kelly & Xiu (2020), "Empirical Asset Pricing via Machine Learning," *Review of Financial Studies* *[Supp.]*
- Chen, Pelger & Zhu (2024), "Deep Learning in Asset Pricing", Management Science *[Supp.]*

Questions?