Big Data in Finance

Lecture 7: Introduction to Neural Networks

Dr Daniele Bianchi

Spring 2026

Contents

Overview

In previous lectures, we developed two distinct approaches to prediction. Linear methods—Ridge, Lasso, Elastic Net—produce smooth predictions but assume that the relationship between predictors and outcomes is linear. Tree-based methods—Random Forests, Gradient Boosting—capture non-linearities and interactions automatically but produce discontinuous predictions.

This lecture introduces **neural networks**, which combine the best of both worlds: smooth predictions *and* the ability to learn complex non-linear patterns from data. The key insight is surprisingly simple: by composing many simple non-linear functions, we can approximate arbitrarily complex relationships.

Sections 1–5 develop the foundations. We begin with the single neuron—which turns out to be nothing more than logistic regression viewed from a different angle. We then show how stacking neurons into layers creates multi-layer perceptrons (MLPs) capable of learning hierarchical representations of data. The learning algorithm, gradient descent with backpropagation, is conceptually straightforward: iteratively adjust weights to reduce prediction error.

Sections 6–8 turn to practice. Using the Goyal-Welch-Zafirov dataset, we implement MLPRegressor for market timing and discover a striking result: despite achieving *negative* out-of-sample R^2, the neural network delivers the highest Sharpe ratio among all methods tested. This puzzle—statistical accuracy versus economic value—reinforces a central theme of this course: evaluate models on their intended use case, not just statistical metrics.

1 From What We Know to Neural Networks

Our toolkit so far includes linear methods and tree-based methods. Each has strengths and weaknesses:

	Linear Models	**Trees**
Smooth predictions	✓	×
Captures non-linearity	×	✓
Finds interactions automatically	×	✓
Extrapolates beyond training data	✓	×

Linear models produce smooth predictions—small changes in inputs lead to small changes in outputs—but they cannot capture non-linear relationships without manual feature engineering (e.g., adding polynomial terms or interaction terms). Trees capture non-linearities automatically but produce piecewise-constant predictions that jump discontinuously at split points.

What if we want smooth predictions *and* automatic non-linearity? This is precisely what neural networks offer.

1.1 A Financial Motivation

Consider return prediction. The relationship between predictors and future returns is unlikely to be linear:

Non-linear effects: The dividend yield might predict returns differently at extreme values than at normal values. A D/P of 6% (historically high) may signal mean reversion, while a D/P of 2% may carry little information.

Interactions: The effect of valuation ratios might depend on the level of interest rates. High valuations may be sustainable when rates are low but dangerous when rates are high.

Regime dependence: Predictability patterns may differ in recessions versus expansions.

Trees can capture these patterns but produce discontinuous predictions. If the tree splits at D/P = 4%, predictions jump discretely as D/P crosses this threshold. Neural networks, by contrast, learn smooth non-linear functions that transition gradually.

2 The Single Neuron: Building Block

The fundamental building block of neural networks is the **neuron**. Remarkably, you already know what a neuron does—it is exactly what logistic regression computes.

Recall from Lecture 4 that logistic regression models default probability as:

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

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

What does this computation involve?

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

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

More generally, a neuron computes:

$$\text{output} = g\underbrace{(w_1 x_1 + w_2 x_2 + \cdots + w_p x_p + b)}_{\text{weighted sum + bias}}$$

where $g(\cdot)$ is the **activation function**. The weights $w_1, \ldots, w_p$ determine how much each input matters (analogous to regression coefficients β), and the bias b shifts the decision threshold (analogous to the intercept β_0).

2.1 Why Non-Linearity Matters

The activation function is crucial. Without it, a neuron computes:

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

which is just linear regression.

The problem is that stacking linear functions produces another linear function:

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

No matter how many layers of linear functions we stack, the result is linear. The activation function "breaks" this linearity, allowing the network to learn curved decision boundaries and approximate complex functions.

2.2 Common Activation Functions

Three activation functions dominate in practice:

Sigmoid: $\sigma(z) = \frac{1}{1+e^{-z}}$. Output in $(0, 1)$. You know this from logistic regression. Historically important but now mainly used in output layers for binary classification.

Tanh: $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$. Output in $(-1, 1)$. Similar to sigmoid but centered around zero, which can help with training.

ReLU (Rectified Linear Unit): $\text{ReLU}(z) = \max(0, z)$. The modern default for hidden layers. Surprisingly simple: if the input is positive, pass it through; if negative, output zero.

ReLU is popular because it is computationally simple, avoids the "vanishing gradient" problem (a technical issue with sigmoid/tanh in deep networks), and creates sparsity (many neurons output zero, making the network efficient).

For this course, we use ReLU in hidden layers and either linear activation (for regression) or sigmoid (for binary classification) in the output layer.

3 Building a Network: Multiple Neurons

A single neuron with sigmoid activation is just logistic regression—it can only learn linear decision boundaries. The power of neural networks comes from combining multiple neurons into layers.

3.1 The Multi-Layer Perceptron (MLP)

The **multi-layer perceptron** stacks neurons into layers:

Input layer: Your features $(x_1, \ldots, x_p)$. This is not really a "layer" of computation—it just holds the inputs.

Hidden layers: Where the learning happens. Each hidden neuron computes a weighted sum of its inputs, applies an activation function, and passes the result to the next layer. These layers are "hidden" because we don't directly observe their outputs.

Output layer: Produces the final prediction. For regression, this is typically a single neuron with linear activation. For binary classification, it's a single neuron with sigmoid activation.

In a **fully connected** (or "dense") network, every neuron in one layer is connected to every neuron in the next layer. Each connection has its own weight to be learned.

3.2 Forward Propagation

Consider a simple network with two inputs, one hidden layer of three neurons, and one output. Here's how information flows:

Step 1: Input to hidden layer. Each hidden neuron computes a weighted sum of inputs and applies ReLU:

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

Step 2: Hidden to output layer. The output neuron combines the hidden layer outputs:

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

Each hidden neuron learns to detect a different pattern in the inputs. The output layer then combines these learned "features" to make a prediction. Crucially, the network learns *what* combinations are useful—we don't have to specify them manually.

Think of hidden neurons as automatically constructed "derived features."

In credit risk: If inputs are income, debt, and credit score, hidden neurons might learn representations like "debt-to-income stress," "credit utilization pattern," or "income stability indicator." We don't tell the network to compute these—it discovers that they are useful for predicting default.

In return prediction: If inputs are D/P, term spread, and inflation, hidden neurons might learn "valuation regime," "monetary policy stance," or "risk appetite indicator."

This automatic feature learning is one of the main advantages of neural networks over linear methods, where you must manually specify interaction terms and non-linear transformations.

3.3 The Universal Approximation Theorem

A remarkable theoretical result: 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 (we usually don't), or the network will generalize well to new data (overfitting is a serious concern).

"Can approximate" $\neq$ "Will approximate well in practice." The universal approximation theorem is a statement about *capacity*, not about what happens with finite data and imperfect optimization.

3.4 Why Depth Matters (Sometimes)

Adding more layers allows more complex functions—but with diminishing returns, especially in finance.

Deeper networks can represent the same function more *efficiently* than shallow networks. The intuition is hierarchical feature learning: Layer 1 learns simple patterns ("is D/P high?"), Layer 2 combines these into more complex patterns ("high D/P *and* rising rates"), and so on.

However, for financial data with limited observations (decades, not millions of data points), deep networks can overfit badly. One or two hidden layers are typically sufficient. More is not always better—it rarely is with limited data.

4 How Neural Networks Learn

We have many parameters to learn: weights connecting every neuron to the next layer, plus biases for each neuron.

Example: A network with 10 inputs, one hidden layer of 50 neurons, and 1 output has:

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

With 25 predictors and 100 hidden neurons, we might have over 2,500 parameters. How do we find values for all of them?

Unlike Ridge regression, where we can solve directly for optimal coefficients using $(X'X + \lambda I)^{-1}X'y$, neural networks have no closed-form solution. The loss function is non-linear in the parameters due to the activation functions, and there are many local optima.

The solution is **iterative optimization**: start with random initial weights, compute predictions and errors, adjust weights to reduce error, and repeat until convergence. This is called **gradient descent**.

4.1 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 does exactly this in the space of network weights. The "gradient" tells us which direction is downhill (the direction of steepest decrease in the loss function), and we take a step in that direction.

The update rule for each weight is:

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

where $\frac{\partial \text{Loss}}{\partial w}$ is the gradient (slope) and η is the **learning rate**—how big a step to take. The minus sign ensures we go *downhill*.

The learning rate is crucial: too large and we overshoot the minimum; too small and training takes forever. Modern optimizers like Adam adapt the learning rate automatically.

How do we compute $\frac{\partial \text{Loss}}{\partial w}$ for all weights efficiently? The answer is **backpropagation** (backward propagation of errors).

The algorithm has three steps:

1. **Forward pass:** Compute predictions by passing inputs through the network
2. **Compute loss:** Compare predictions to true values (e.g., using MSE)
3. **Backward pass:** Use the chain rule of calculus to compute how each weight contributed to the error

The key insight is that errors at the output "propagate back" through the network. If the output is too high, weights that contributed positively to the output should decrease; weights that contributed negatively should increase.

You don't need to implement backpropagation—sklearn (and all neural network libraries) compute gradients automatically. Just understand the intuition: the algorithm efficiently assigns "blame" to each weight for the prediction error.

Computing gradients over the entire dataset is slow. **Stochastic gradient descent (SGD)** uses a random subset ("mini-batch") at each step. This is faster and, surprisingly, the noise can actually help escape local minima.

Adam (Adaptive Moment Estimation) is the modern standard optimizer. It adapts the learning rate for each parameter individually and uses "momentum" to smooth out noisy gradient estimates. In sklearn, use `solver='adam'`—it's robust and works well in most cases.

5 Regularization: Preventing Overfitting

Neural networks have many parameters—perfect for overfitting. A network with 20 inputs and 100 hidden neurons has over 2,000 parameters. With only a few hundred observations (typical in financial time series), the network can easily memorize training data instead of learning generalizable patterns.

Symptoms of overfitting: training loss is very low, but validation/test loss is much higher. The model fits noise in the training data that doesn't persist out of sample.

This is especially problematic in finance: limited historical data, low signal-to-noise ratio, and patterns that may change over time.

5.1 Strategy 1: Weight Decay (L2 Regularization)

Just like Ridge regression, we can add an L2 penalty to the loss function:

$$\text{Loss} = \text{MSE} + \alpha \sum_j w_j^2$$

where α controls the strength of regularization. This penalizes large weights, forcing the network to find simpler solutions.

In sklearn's MLPRegressor, this is controlled by the `alpha` parameter. Typical values range from 0.0001 to 0.1, selected via cross-validation.

5.2 Strategy 2: Early Stopping

During training, validation error typically decreases at first, reaches a minimum, then increases as the network overfits. **Early stopping** monitors validation error and stops training when it starts increasing.

This is one of the most effective regularization techniques for neural networks. In sklearn, set `early_stopping=True` and the algorithm automatically monitors a validation set.

5.3 Strategy 3: Architecture Choices

Simpler networks have fewer parameters and less capacity to overfit. Rules of thumb for financial applications:

- Start with one hidden layer
- Number of neurons: between p (number of features) and $2p$, or around 50–100
- Add complexity only if validation performance improves

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

5.4 Feature Scaling: Critical for Neural Networks

Unlike trees, neural networks are sensitive to feature scales. Gradient descent works better when features are on similar scales; otherwise, large-scale features dominate the learning process.

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.

Critical: Compute mean and standard deviation on *training data only*, then apply the same transformation to validation and test data. sklearn's `StandardScaler` handles this, and you should wrap it in a `Pipeline` with the neural network.

6 Implementation: MLPRegressor for Market Timing

We now apply neural networks to the market timing problem from Lectures 2–3. The Goyal-Welch-Zafirov dataset contains 25 macroeconomic predictors of monthly excess market returns from 1956–2021.

We compare six methods: OLS, Ridge, Lasso, Elastic Net, Random Forest, and MLP, using walk-forward prediction with time-series cross-validation for hyperparameter selection.

6.1 Statistical vs. Economic Performance

The results reveal a striking pattern:

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**

MLP achieves the highest Sharpe ratio (0.74) despite having *negative* out-of-sample R^2 (−0.68%). This is a 60% improvement over buy-and-hold (0.46).

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

How can MLP have negative R^2_{OS} but the best Sharpe ratio? The answer lies in what these metrics measure:

R^2_{OS} measures average squared error. It penalizes large errors heavily (they're squared). A few big mistakes hurt R^2 a lot, even if most predictions are reasonable.

Sharpe ratio depends on *when* you're right. Being right when it matters most—during big market moves—is valuable. Small errors when the market is flat don't hurt much.

Direction matters more than magnitude for portfolios. Predicting +1% when the actual return is +5% still makes money. Getting the sign right is what counts for trading strategies.

The lesson: always evaluate models on their intended use case. For trading, economic metrics matter more than R^2.

6.3 Terminal Wealth

The cumulative performance differences are dramatic. \$1 invested in 1963 grows to:

- Buy & Hold: \$29
- Ridge: \$197

- MLP: \$405

MLP's terminal wealth is 14× higher than buy-and-hold and 2× higher than Ridge.

6.4 Why Does MLP Outperform?

Several factors contribute to MLP's success:

Smooth non-linearity: MLP captures non-linear relationships smoothly, while trees produce "blocky" predictions that jump at split points. This matters for trading because smooth predictions lead to gradual portfolio adjustments.

Flexible interactions: The network learns which predictor combinations matter without manual specification.

Regularization works well: L2 penalty plus early stopping prevents overfitting, similar to why Ridge beats OLS.

Lower turnover: MLP has average monthly turnover of 40.1%, compared to 46.6% for Ridge/OLS. Smooth predictions mean gradual weight changes, reducing transaction costs.

6.5 Transaction Costs

We test robustness to transaction costs of 50 basis points per trade:

Strategy	**Turnover (%/month)**	**Sharpe (Gross)**	**Sharpe (Net)**
Buy & Hold	0.0	0.46	0.46
Lasso	1.8	0.41	0.40
Elastic Net	11.6	0.53	0.50
Random Forest	37.8	0.47	0.39
MLP	40.1	**0.74**	**0.66**
Ridge	46.6	0.64	0.56
OLS	46.6	0.65	0.56

MLP remains best even after costs (Sharpe: 0.66 vs. 0.56 for Ridge). Notably, Random Forest falls *below* buy-and-hold after costs—its piecewise-constant predictions create high turnover that erodes returns.

6.6 A Caveat: Drawdowns

All timing strategies have significant maximum drawdowns, comparable to or larger than buy-and-hold:

Strategy	**Max Drawdown (%)**
Buy & Hold	−54.3
MLP	−57.4
Ridge	−58.3
Random Forest	−61.2

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.

7 Neural Networks for Classification

The same architecture applies to classification problems like credit risk. The key differences are in the output layer:

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

sklearn's `MLPClassifier` has nearly the same interface as `MLPRegressor`. Use `predict_proba()` to get probability estimates for credit scoring.

7.1 Handling Class Imbalance

Unlike Random Forest and Logistic Regression, `MLPClassifier` does not have a `class_weight` parameter. Solutions include:

Resampling: Oversample the minority class (e.g., SMOTE) or undersample the majority class.

Threshold tuning: Train with default threshold (0.5), then adjust based on cost trade-offs. This is often the simplest effective approach.

8 When Do Neural Networks Help?

The main advantage of MLPs for classification is smooth probability estimates, which can be valuable for risk-based pricing. The main disadvantage is lack of interpretability, which is problematic for regulatory applications.

The academic literature provides guidance on when neural networks are most valuable.

Gu, Kelly & Xiu (2020, Review of Financial Studies) compared many ML methods for stock return prediction. Neural networks were competitive but not always best. Gains were most pronounced with many features and large samples (cross-sectional data with thousands of stocks).

Chen, Pelger & Zhu (2024, Management Science) found that deep learning benefits emerge primarily with cross-sectional data (many stocks). For time-series prediction (market timing), improvements are more modest.

Goyal, Welch & Zafirov (2024, Review of Financial Studies) concluded that "no method consistently dominates" for equity premium prediction. Simple methods often perform as well as complex ones.

8.1 Conditions Favoring Neural Networks

Based on our results and the literature, MLPs tend to help when:

- Relationships are non-linear but smooth
- Interactions between predictors matter
- Regularization is applied carefully
- Turnover costs are a concern (smooth predictions help)
- Large datasets are available (cross-sectional or many observations)

8.2 Conditions Favoring Simpler Methods

Simpler methods may suffice when:

- Interpretability is required (regulatory, audit)
- Computational resources are limited
- Quick iteration and debugging is needed
- Data is very limited (typical in macro finance)

8.3 Computational Considerations

Neural networks are more expensive than simpler methods:

Method	Training Time	Hyperparameters
Ridge	Very fast	1
Random Forest	Moderate	2–3
XGBoost	Moderate	4–5
MLP	Slower	3–5

In our GWZ application, MLP takes 5–10× longer than Ridge to tune and fit. For production systems with frequent refitting, this can matter.

9 Summary and Key Takeaways

This lecture has covered neural networks and their application to financial prediction. The key points:

Neural networks compose simple non-linear functions. A neuron computes a weighted sum, applies a non-linear activation, and outputs a value. Stacking neurons into layers creates powerful function approximators.

You already knew the basics. Logistic regression is a one-neuron neural network. The sigmoid function, weighted sums, bias terms—these are all familiar from Lecture 4.

Gradient descent learns the weights. Unlike Ridge, there's no closed-form solution. We iteratively adjust weights to reduce prediction error. Backpropagation efficiently computes gradients.

Regularization is essential. With many parameters and limited data, overfitting is a serious concern. Use L2 regularization (`alpha`), early stopping, and simple architectures. Always scale features.

MLP achieved the best performance for market timing. Sharpe ratio of 0.74 (gross), 0.66 (net of costs), compared to 0.46 for buy-and-hold. Terminal wealth 14× higher than passive investing.

Statistical accuracy ≠ economic value. MLP had negative R^2_{OS} but the best Sharpe ratio. Evaluate models on their intended use case.

Smooth non-linearity matters. MLP captures non-linear patterns with gradual predictions, leading to lower turnover than trees. This robustness to transaction costs is a practical advantage.

Strong results deserve scrutiny. Implementation risk, structural breaks, and interpretability concerns all warrant caution. Ridge remains a strong, simpler alternative.

Readings

Required:

- James, G., Witten, D., Hastie, T., Tibshirani, R., & Taylor, J. (2023). *An Introduction to Statistical Learning with Applications in Python*, Chapter 10.

Supplementary:

- Gu, S., Kelly, B., & Xiu, D. (2020). "Empirical Asset Pricing via Machine Learning." *The Review of Financial Studies*, 33(5), 2223–2273. Comprehensive comparison of ML methods for return prediction.
- Chen, L., Pelger, M., & Zhu, J. (2024). "Deep Learning in Asset Pricing." *Management Science*, 70(2), 714–750. Application of deep learning to cross-sectional asset pricing.
- Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press. The definitive textbook on neural networks (Chapters 6–7 for foundations).