Stock Market Forecasting with Recurrent Neural Networks and Long Short-Term Memory

Hunter Wright

2025-03-10

Introduction

Time series prediction, also known as *forecasting*, is the process of building predictive models using historical time series data. Applications of forecasting are used across many different disciplines, like meteorology, finance, and macroeconomics, and are used to make important decisions for governments and businesses around the world. This wiki focuses on stock market prediction, a notoriously unsuccessful practice in which analysts try to predict the price of stocks using predictive modeling. In machine learning, analysts use **Recurrent Neural Networks** (RNN) and **Long Short-Term Memory** (LSTM) to predict stock prices, with mixed results.

History

Recurrent Neural Networks Santiago Ramón y Cajal

Recurrent neural networks originated from theories in neuroscience and research in statistical mechanics in the early 1900s. In 1901, Spanish neuroscientist Santiago Ramón y Cajal discovered "recurrent semicircles" mad from neurons inside the human brain, which led to theories about feedback loops in the brain by the 1940s. In 1960, Frank Rosenblatt published a paper on "close-loop cross-coupled perceptrons." These 3 layer machine learning models contained recurrent connections in the middle layer, and are considered to be the predecessor to modern RNN.

Long Short-Term Memory Modern recurrent neural networks face issues known as the **vanishing gradient problem** and the **exploding gradient problem**. Both of these problems relate to how an RNN treats data from different time steps, by either severely under or over estimating the weight applied to past calculations. Both of these interrupt RNN's learning process, rendering it useless. Hochreiter & Schmidhuber solved this problem by developing the first LSTM in 1995 and Gers, Schmidhuber, and Cummins added the "forget gate" in 1999, creating the LSTM most commonly used today.

Neural Networks

A Basic Neural Network

Before learning about RNNs and LSTMs, it's important to establish what a regular neural network is. Neural networks are models consisting of a number of interconnected nodes, or neurons, that process data and learn patterns in a manner similar to a human brain. They receive inputs, pass them through layers that apply mathematical functions, and generate outputs.

Building Blocks

Neurons

Neurons are basic units that receive inputs. Each neuron sums its inputs, applies a bias, and then passes the result to an *activation function*. The activation function is a mathematical formula that helps transform a neuron's input into a usable output. The activation function also introduces non-linearity into the model, which is crucial in improving a model's performance in estimating complex relationships.

Connections

Links between neurons that carry information. In a regular neural net, information typically moves in only one direction, forward. This is known as *forward propagation*. During this process, weights and biases are applied to the information.

Weights and Biases

Weights and biases are parameters determining strength and influence of connections. Each connection has a weight, and each neuron has an associated bias.

Propagation Functions

Mechanisms that help process and transfer data across layers of neurons.

Learning Rule

Adjusts weights and biases over time to improve accuracy, known as *backpropagation*.

Architecture

Every neural net is composed of three main layers:

1. **Input Layer**: Where the network receives input data
2. **Hidden Layers**: Perform the primary computations
3. **Output Layer**: Where processed information is transformed into a final decision/prediction

In a regular neural network, data moves through these layers in order, which is called forward propagation. After moving inputs through every layer, the neural net learns from its mistakes using backpropagation. Backpropagation is a process where the neural net compares its predicted output to the actual output, calculates its error, and works backward to find out which weights caused the error, and adjusts them. This is similar to checking your work on a math problem, finding your mistake, and adjusting the process you used to correct the error.

Recurrent Neural Networks

A Recurrent Neural Network

Recurrent neural networks add an additional step to the movement of inputs through layers. While a regular neural net moves inputs in a streamlined process with forward propagation, recurrent neural nets feed every piece of new information back into the system. In the context of stock market prediction, this is like trying to predict the price of a stock using not just the current price, but every past price as well. Recurrent neural nets "remember" the outputs of past steps, and feed them into every new step within the network. This process is extremely important for time series data, as regular neural nets lack the "memory" required to process sequential data.

Building Blocks

Recurrent Neurons

Neurons that contain information from previous inputs in a hidden state. They can "remember" these inputs by taking input of their previous outputs.

Connections:

On top of regular feed-forward connections, RNNS have recurrent (feedback) connections. These connections feed a neurons output back into itself, allowing the model to use past information.

Weights and Biases

The same weights and biases are used at each time step. This is known as temporal weight sharing, and is essential for sequential data processing.

Left: A recurrent neuron. Right: The unfolded neuron showing multiple time steps.

RNN Unfolding

Unlike regular forward propagation in a neural net, RNN extends its propagation function to include the flow of time. Because the same weights are used at every time step, during training the RNN is "unfolded" into a series of copies, each corresponding to a different time step. This is important for understanding how an RNN improves its predictions during training.

Learning Rule

RNNs use a variant of backpropagation called backpropagation through time (BPTT), which unrolls the network across time steps so that gradients can be computed with respect to both current inputs and past states.

Architecture

RNNs are similar in structure to other types of neural nets, and differ mainly in how information flows from neuron to neuron. While other neural nets have specific weights at each connection, RNNs maintain the same weights across time steps, which is where their "memory" comes from.

Computations

As inputs pass through neurons in the hidden layer, they are transformed using a propagation function that might look like:

$$h = \sigma(U \cdot X_i + W \cdot h_{t-1} + B)$$

In this formula, h is the hidden state, σ is the activation function, X is the input, U and W are weights, and B is bias. h_{t-1} represents the previous hidden state, containing all previous inputs and transformations, and is what sets RNNs apart from regular neural nets. Applying h_{t-1} to the input X is how RNNs remember previous inputs and apply that memory at each new transformation, shown by $h_t = f(h_{t-1}, x_t)$. The activation function σ is given by:

$$h_t = \tanh(W_{hh} \cdot h_{t-1} + W_{zh} \cdot X_t)$$

In this formula, W_{hh} and W_{zh} are *weight matrices*. A weight matrix holds all of the learnable parameters that combine and scale inputs as they move from one neuron to the next. W_{hh} controls how the previous hidden state influences the current hidden state, and W_{zh} determines how the input is factored into the new

hidden state. When an RNN is ready to produce an output, it applies one more activation function to the latest hidden state, which looks like this:

$$Y = O(W_{hy} \cdot h + b_y)$$

Here, Y is the output, O is the activation function, W_{hy} is the output weight, and b_y is the output bias. This is a crucial step in the RNN process, responsible for transforming the complex multidimensional hidden state into a usable output. The weight matrix W_{hy} and activation function O work in tandem to complete this task, while the bias term adjusts for any systematic discrepancies between the hidden state and output. It's important to note that the output function is applied at every time step, which is important for propagating error signals, and helping the model improve itself over time.

Backpropagation Through Time (BPTT)

Neural nets conduct a process known as backpropagation to adjust the parameters of their weights and biases and reduce error. In other words, neural nets go back and look through their calculations, "learn" where the errors they made are, and adjust their calculations to improve their predictions. Recurrent neural nets go through a similar process, but because they conduct this action over multiple time steps, the process adds a step to backpropagation, and is known as backpropagation through time, or BPTT. Before explaining how BPTT is performed, two terms need to be defined: *gradients* and the *loss function*. A gradient represents how a small change in each parameter affects the *loss*, or error between the predicted outputs and actual values during training. The loss function is a mathematical formula used to calculate loss. During BPTT, the network first "unfolds" across time steps. As mentioned earlier, recurrent neural nets use the same weights at every time step, and unfolding creates a vector of copies which are then used to calculate the loss at each step. The chain rule is then applied to the entire sequence. Gradients are computed at each time step and combined to update the parameters, which are the same for every step. This is all done to minimize loss and improve the model's predictions.

The Vanishing Gradient Problem During BPTT, all of the gradients are summed together to update the parameters. This means that the magnitude of the gradients is directly responsible for how the model decides to update its parameters and minimize loss. During backpropagation, the chain rule multiplies derivatives of each output at every step to calculate gradients. Over time, if these derivatives are less than 1, they can exponentially shrink gradient calculations to zero. As gradients shrink to zero, the model applies less and less of a change to its parameters, effectively stifling its ability to learn. This same process can also happen the other way around, where gradients increase exponentially towards infinity, causing the model to become erratic, and ultimately having the same effect. This problem mainly affects RNNs long term memory, as the further back a neuron is, the more calculations are applied to it when determining its gradient. Long Short Term Memory (LSTM) models were invented to mitigate the vanishing gradient problem.

Long Short-Term Memory

Long Short-Term Memory models are upgraded recurrent neural networks. They are better equipped at dealing with long sequences of data which improves their recognition of long term dependencies and trends. The underlying building blocks of an LSTM are the same as an RNN, though more complex with the addition of more sophisticated features like gating mechanisms.

Architecture

The Memory Cell

The biggest difference between RNNs and LSTMs is the memory cell. In an RNN, the basic unit is a recurrent neuron, which essentially takes an input, applies weights and biases, and produces an output. Recurrent

neurons update their hidden state uniformly across time steps, which can lead to the vanishing gradient problem. LSTMs address this issue with an enhanced version of a recurrent neuron called the memory cell. Memory cells use gates to control what information passes through the cell and what information is discarded. By discarding irrelevant information, LSTMs can maintain "cleaner" memories without diluting important information after many time steps. This allows gradients to focus more on relevant information and improves accuracy.

The Cell State

While normal RNNs only maintain a hidden state, LSTMs also maintain a cell state that runs along side the hidden state. The cell state is the LSTM's long term memory, regulated by the gating mechanisms in every memory cell. It is what sets an LSTMs abilities apart from an RNN, which doesn't have the ability to maintain a long term regulated memory.

The Input Gate The input gate determines how much new information should be written to the cell state at each memory cell. It computes a gate value, which is typically a sigmoid activation, that includes both the input value and previous hidden state. It is given by:

$$i_t = \sigma\big(W_i[h_{t-1}, x_t] + b_i\big)$$

and

$$\hat{C}_t = \tanh\big(W_c[h_{t-1}, x_t] + b_c\big)$$

The first equation represents the input gate. It applies the sigmoid activation function σ to a linear combination of the previous hidden state and the input. This outputs a value between 0 and 1, which determines how much of the new information should be written to the cell state. The second equation represents the new information that could be added to the cell state. It is passed through a tanh function to constrain its value to between -1 and 1.

The Forget Gate The forget gate determines what information from the previous cell should be discarded, and what should be kept. It is different from the input gate in that it isn't deciding how much new information should be added to the updated cell, just what information can be added from the previous cell. In other words, the input gate determines what proportion of new input information is added, and the forget gate determines what proportion of information from the previous cell should be retained. The forget gate equation is given by

$$f_t = \sigma\big(W_f[h_{t-1}, x_t] + b_f\big)$$

Both of these processes work concurrently with each other, and their outputs are combined in the cell state update equation:

$$C_t = f_t \odot C_{t-1} + i_t \odot \hat{C}_t$$

Where:

- C_t: The new cell state at time step t
- f_t: The forget gate's output, a vector of values between 0 and 1
- C_{t-1}: The previous cell state a time step $t-1$
- i_t: The input gate's output, a vector of values between 0 and 1
- $\hat{C}_t$: The candidate cell state, or new information to be added to the cell state

Note: The $\odot$ symbol represents multiplying the corresponding elements of two vectors of equal length, known as element-wise (Hadamard) multiplication.

The Output Gate After the cell updates its state, the output gate determines what part of this information is exposed as the hidden state for the next time step or final output. It processes the previous hidden state and the new input through a sigmoid function to generate a filter. The cell state is then passed through a tanh activation function to scale its values between -1 and 1, which is multiplied element-wise by the output gate's filter, creating the new hidden state. This process is given by the equation:

$$o_t = \sigma\big(W_o[h_{t-1}, x_t] + b_o\big)$$

Useful Analogy Imagine the LSTM is a bakery, and each time step is a day. Every day, the bakers look into the storage and pull out all of the pastries from the previous day that they didn't sell. This includes both pastries they baked yesterday and pastries they baked in earlier days. They then decide which pastries to display and which to throw out. This is the forget gate. They also bake new pastries, and they decide which to display and which to throw out. This is the input gate. At the cash register, the cashier decides which pastries to select when a customer buys one, and then gives them that pastry. This is the output gate.

Applications

Recurrent neural networks are used in a variety of fields where sequential data or the ability to "remember" is important for analysis. They include the following:

- Natural Language Processing
  - Text-to-Speech
  - Text Generation
  - Language Modeling
  - Speech Recognition
  - Speech-to-Text
  - Machine Translation
- Classification
- Music and Video Processing
- Anomaly Detection
- Time Series Forecasting
  - Weather Forecasting
  - Stock Market Prediction

add why its used, math, example