Analyzing Recurrent Neural Networks for Time Series Forecasting

Tran Viet Hung

April 2024

Contents

1 Introduction 2

2 Background 2

3 Recurrent Neural Network 3
3.1 What is a Recurrent Neural Network? 3
3.2 How Does Recurrent Neural Networks Work? 4
3.2.1 How Recurrent Neural Networks Remember 5
3.2.2 Training Recurrent Networks 6
3.2.3 Types of Recurrent Neural Networks 8
3.3 Summary 9

4 Bidirectional Recurrent Neural Networks 10

5 Long Short-Term Memory (LSTM) 11
5.1 The Logic Behind LSTM 12
5.2 Forget Gate 13
5.3 Input Gate 14
5.4 Output Gate 16
5.4.1 Overview 17

6 Using Recurrent Neural Networks (RNN) and LSTM Models to Predict Stock Prices 18
6.1 Data Preparation 18
6.2 Complementary Method 19
6.3 Result 20
6.3.1 RNN Model Results 20
6.3.2 LSTM Model Results 21
6.3.3 General Observations 23

1 Introduction

Neural networks have been a prominent topic in computer science, with various models developed to address real-world problems. The achievements of neural networks go beyond image processing and speech recognition tasks and extend into many fields such as time series prediction, language translation, and even content creation. Among neural network models, Recurrent Neural Networks (RNNs) stand out for their ability to efficiently handle sequential data, where information from previous time steps influences predictions at subsequent time steps.

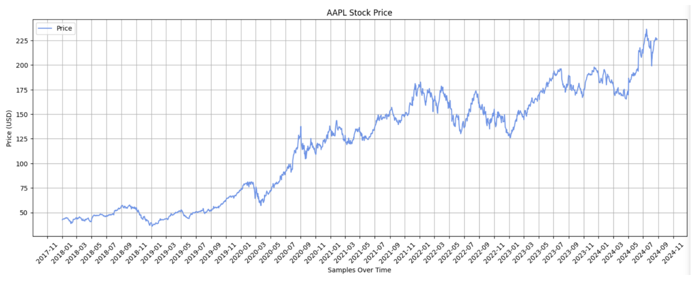
RNNs are specifically designed to work with sequential data, such as text, audio, and financial data, by maintaining a "memory state" across time steps. This allows the model to flexibly store and update information, making it more accurate and efficient in processing time-structured data compared to traditional neural networks.

In this report, I will focus on analyzing the structure and principles of RNNs, as well as their practical applications in time series tasks, such as financial forecasting, speech recognition, and natural language processing. I will also consider the challenges related to training RNNs and the improvements made to address these issues, including the use of variants like Long Short-Term Memory (LSTM).

2 Background

Before exploring RNNs, we will briefly discuss what time series data is. Time series data consists of a set of data points collected at regular intervals, where each value is not just a standalone piece of data but is also closely tied to a specific time point. This time element is what makes time series unique, allowing them to be useful tools for tracking, analyzing trends, and identifying patterns over time.

To help illustrate time series, consider the chart below, showing Apple's stock prices from 2017 to the end of 2024.



This chart illustrates the fluctuation of Apple's stock prices over time. Each data point on the time axis corresponds to a specific stock price at a particular moment. As we can see, stock prices change over time with clear cycles of increases and decreases, along with a long-term upward trend during this period.

Time series data like the one in this chart can be used to predict future price trends based on patterns that have occurred in the past. With the ability to store and process information from previous time steps, Recurrent Neural Networks (RNNs) are ideal for analyzing and predicting such time series fluctuations.

3 Recurrent Neural Network

3.1 What is a Recurrent Neural Network?

As we have learned, time series data has a crucial characteristic: temporal order, where each data point depends on previous values. This feature requires analytical models that can store information from the past to make accurate future predictions.

However, traditional neural network models such as Artificial Neural Networks (ANNs) or Convolutional Neural Networks (CNNs) are not effective at solving this problem because they cannot retain states from previous steps. That's why Recurrent Neural Networks (RNNs) were developed.

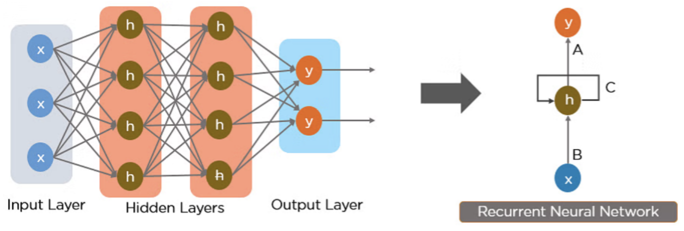


Fig: Simple Recurrent Neural Network

The figure above illustrates the basic structure of a Recurrent Neural Network. During processing, each time step is represented through hidden layers, and information from previous steps is fed back as input for subsequent steps. This helps maintain the state of the data from the past and ensures that the model can process sequential data efficiently.

To train RNNs effectively, we use the Backpropagation Through Time (BPTT) algorithm. After information is passed through time steps in the forward direction, the BPTT algorithm propagates errors backward through each previous time step. This process helps adjust the network's weights based on errors at each step, allowing the model to learn from historical data and improve prediction accuracy. This algorithm was introduced in 1990 and is the main method for efficiently training RNNs.

3.2 How Does Recurrent Neural Networks Work?

In Recurrent Neural Networks (RNNs), information from previous time steps is fed back and combined with the current data to calculate the next time step. This allows the network to "remember" past information, enabling it to process sequential data such as text or time series. The feedback mechanism is why it's called "recurrent," as information continuously loops and feeds back over multiple time steps.

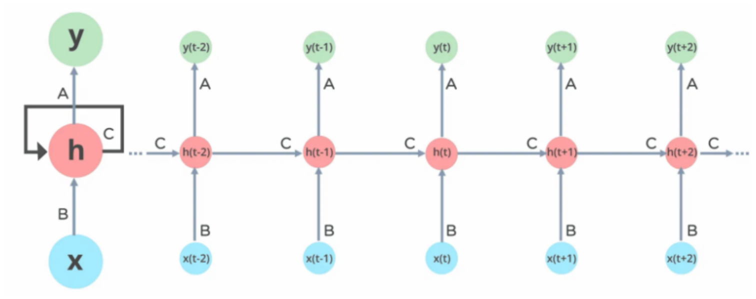


Fig: Working of Recurrent Neural Network

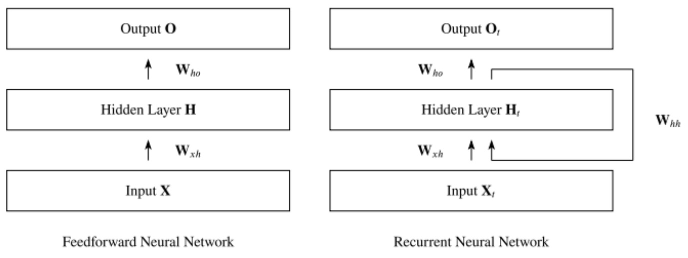
The figure above illustrates how an RNN works. The input layer x receives sequential data and passes it through time steps. At each time step, data from the input layer is processed and passed to the hidden layer h.

The hidden layer h plays a critical role in retaining and processing information from previous time steps. Unlike traditional neural networks where hidden layers operate independently, RNNs maintain a "memory" by passing information from the previous hidden layer to the current hidden layer. This is done by using the same weights and activation functions for each hidden layer, creating repetition of time steps without needing separate hidden layers.

An RNN processes information by repeating a single hidden layer over multiple time steps, rather than creating independent hidden layers. This allows RNNs to effectively use past information to improve its predictions for future time steps. The output y of the network at each time step depends not only on the input at that time step but also on the states retained from previous time steps.

3.2.1 How Recurrent Neural Networks Remember

In Recurrent Neural Networks (RNNs), information from previous time steps is fed back and combined with the current data to calculate the next time step. This process allows the RNN to retain information across time steps, or in other words, helps the network "remember" past data in sequential order. The formulas below detail how this process works.



The hidden state H_t at time step t in an RNN is computed by the formula:

$$H_t = \phi_h \left(X_t W_{xh} + H_{t-1} W_{hh} + b_h \right)$$

Where:

- W_{xh} is the weight matrix between the input X_t and the hidden state H_t,
- W_{hh} is the weight matrix between the previous hidden state H_{t-1} and the current hidden state H_t,

- b_h is the bias vector for the hidden state,
- ϕ_h is the activation function, such as the sigmoid or tanh function.

It is important to note that **these weights and biases do not change** throughout the process across time steps. This ensures that the RNN can handle sequences of any length without increasing the number of parameters to be trained.

This process allows the RNN to "remember" information from the previous time step and combine it with new input to compute the current state.

The output $\hat{O_t}$ at time step t is calculated as follows:

$$\hat{O_t} = \text{softmax}(H_t W_{ho} + b_o)$$

Where:

- W_{ho} is the weight matrix between the hidden state H_t and the output $\hat{O_t}$,
- b_o is the bias vector for the output,
- The softmax function is used to normalize the output into probabilities.

Just like the weight matrices and biases in the hidden state computation, the parameters W_{ho} and b_o also **do not change** throughout the time steps. This ensures that the network can handle long sequences without adding new parameters at each step.

These formulas describe how an RNN passes information through time steps while using the hidden state to compute the output at each step. As a result, RNNs can process continuous and sequential data efficiently, especially in tasks involving time series, text, or audio.

3.2.2 Training Recurrent Networks

Training Recurrent Neural Networks (RNNs) is a critical process aimed at optimizing the model's parameters, such as weights and biases. This process requires the model to learn how to minimize a loss function, which reflects the discrepancy between predicted values and actual values. One of the main algorithms used to train RNNs is **Backpropagation Through Time (BPTT)**.

Overview of Backpropagation Through Time (BPTT) BPTT is an extended version of the traditional backpropagation algorithm, adapted to work with sequential data like time series. Instead of propagating errors back only through layers, BPTT propagates errors back through time steps.

When using BPTT, the error at each time step is propagated backward from the end of the sequence to the previous time steps. This allows the model to "remember" and adjust its weights based on accumulated errors across multiple time steps.

Loss Function To evaluate the quality of the model, a loss function is used to measure the difference between predicted and actual values. In time series regression tasks, *Mean Squared Error (MSE)* is one of the most common loss functions. The MSE measures the total squared error between the predicted and actual values at each time step, then adjusts the model's weights based on this loss value.

$$L_t = \frac{1}{n}\sum_{i=1}^{n}(y_t - \hat{y_t})^2$$

Where:

- y_t is the actual value at time step t,
- $\hat{y_t}$ is the RNN's predicted value at time step t,
- n is the total number of samples in the training set.

The MSE is used to minimize the error between the predicted and actual values by adjusting the weights through the BPTT algorithm.

Backpropagation Through Time (BPTT) The backpropagation through time process in BPTT allows the model to adjust its weights at each time step. The error from the final output is propagated back to previous time steps, helping the model optimize predictions based on sequences of past events. Instead of computing gradients only through layers, BPTT computes gradients across time steps, allowing the model to learn from sequential data.

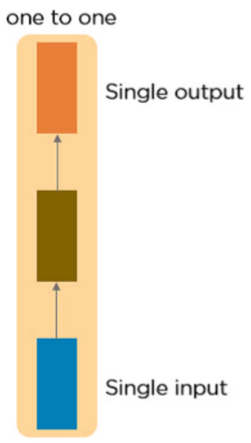
Challenges in Training: Vanishing Gradient and Exploding Gradient During the training of RNNs using BPTT, two common gradient-related issues can occur: *vanishing gradient* and *exploding gradient.*

- **Vanishing Gradient** occurs when the gradient becomes very small during backpropagation through time steps. Particularly with activation functions like tanh or σ, the gradient can decrease rapidly when passed through many time steps. This makes it difficult for the model to learn information from distant time steps, as the gradient is too small to adjust the weights effectively. As a result, the model cannot "remember" well what happened in the distant past.
- **Exploding Gradient** occurs when the gradient becomes too large during backpropagation, especially when weights are too large. This can lead to the weights being over-adjusted, causing the model to fail to converge and potentially leading to "exploding" errors. A common method to resolve this issue is using *gradient clipping*, which limits the value of the gradient to prevent it from becoming too large and ruining the training process.

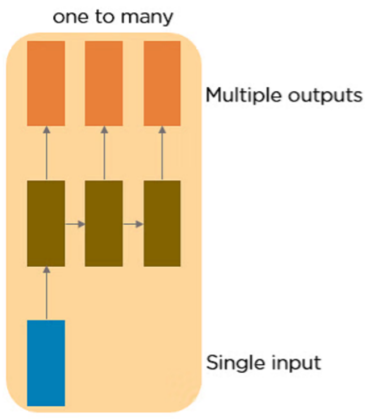
3.2.3 Types of Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are not limited to a single structure but can be customized to handle different types of tasks. Based on the relationship between input and output, RNNs can be divided into four main types. Each type serves a specific purpose in sequential data processing.

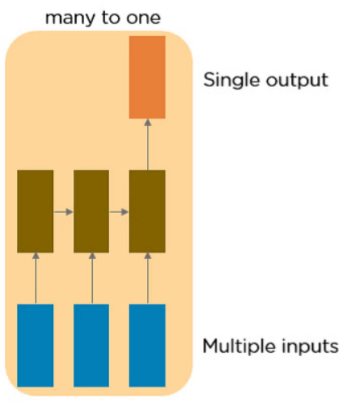
One to One: This is the simplest RNN, often used for non-sequential tasks.



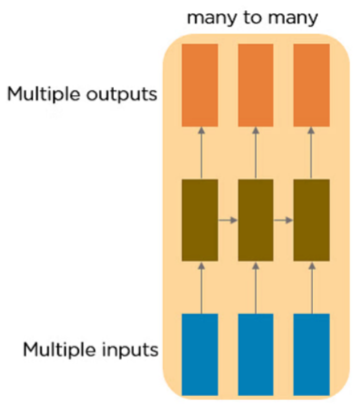
One to Many: This type is suitable for tasks where a single input generates multiple outputs, such as generating a sequence of text from a keyword.



Many to One: This model is used when multiple sequential inputs lead to a single output, typical in text classification tasks.



Many to Many: This type of RNN is suited for tasks where both input and output are sequential, such as in machine translation.



3.3 Summary

Recurrent Neural Networks (RNNs) are a powerful and flexible model for processing sequential time-series data. Thanks to their recurrent structure, RNNs can retain information from the past, making them highly suitable for tasks involving sequential data, such as time series prediction, machine translation, and text classification. The mathematical formulas related to RNNs are also straightforward and intuitive, allowing the model to be easily deployed in many practical applications.

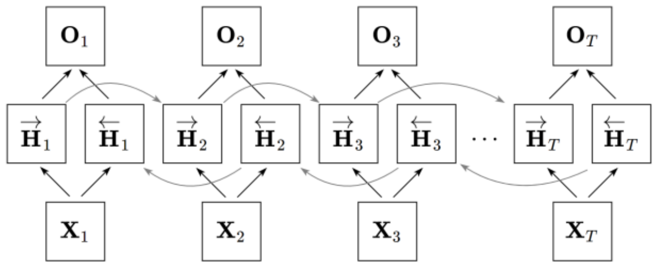
However, RNNs are not without limitations. One of the biggest issues RNNs face is the *vanishing gradient* problem, which makes it difficult for the model to learn and retain information from distant time steps. This limits the model's ability to learn long-term dependencies in sequential data. Additionally, RNNs can suffer from the *exploding gradient* problem, making the training process unstable and sometimes leading to errors when the gradient becomes too large.

Moreover, another significant limitation of RNNs is their computational speed. Due to the sequential nature of the network, RNNs process each time step one by one, making the training process slow, especially when working with long sequences of data. Furthermore, the memory capacity of traditional RNNs is limited when dealing with very long sequences of data.

To address these limitations, variants of RNNs such as *Long Short-Term Memory (LSTM)* and *Gated Recurrent Unit (GRU)* have been developed. These models improve RNNs' long-term memory capabilities and reduce the vanishing gradient problem while optimizing computational efficiency. Despite the challenges, RNNs and their variants still play a crucial role in solving time series and sequential data problems.

4 Bidirectional Recurrent Neural Networks

To enhance the time series processing capabilities of RNNs, the **Bidirectional Recurrent Neural Networks (BRNNs)** model was introduced. BRNNs can access information from both directions of the sequence: forward (from past to present) and backward (from future to past). This gives the model a more comprehensive view of the context surrounding each time step, thus improving prediction quality, especially in applications requiring accurate predictions at each time step, such as natural language processing and machine translation.



$\mathbf{O}_1$
$\mathbf{O}_2$
$\mathbf{O}_3$
$\mathbf{O}_T$
$\overrightarrow{\mathbf{H}}_1$
$\overleftarrow{\mathbf{H}}_1$
$\overrightarrow{\mathbf{H}}_2$
$\overleftarrow{\mathbf{H}}_2$
$\overrightarrow{\mathbf{H}}_3$
$\overleftarrow{\mathbf{H}}_3$
$\cdots$
$\overrightarrow{\mathbf{H}}_T$
$\overleftarrow{\mathbf{H}}_T$
$\mathbf{X}_1$
$\mathbf{X}_2$
$\mathbf{X}_3$
$\mathbf{X}_T$

The figure above illustrates the architecture of a BRNN. Each time step of the input sequence X_t is fed into both a forward RNN and a backward RNN. These networks create the forward hidden state $\overrightarrow{H_t}$ and the backward hidden state $\overleftarrow{H_t}$, respectively. These states are then combined to produce the final output O_t.

Mathematically, the calculations of BRNN can be described as follows:

$$\overrightarrow{H_t} = \phi\left(X_t W_{xh}^{(f)} + \overrightarrow{H_{t-1}} W_{hh}^{(f)} + b_h^{(f)}\right)$$

$$\overleftarrow{H_t} = \phi\left(X_t W_{xh}^{(b)} + \overleftarrow{H_{t+1}} W_{hh}^{(b)} + b_h^{(b)}\right)$$

Where $W_{xh}^{(f)}, W_{xh}^{(b)}$ are the forward and backward weight matrices for the input, and $W_{hh}^{(f)}, W_{hh}^{(b)}$ are the weight matrices for the forward and backward hidden states. $b_h^{(f)}, b_h^{(b)}$ are the corresponding biases. The function ϕ is an activation function, such as the tanh or ReLU function.

The final output of the BRNN at each time step O_t is computed by combining the forward hidden state $\overrightarrow{H_t}$ and the backward hidden state $\overleftarrow{H_t}$:

$$O_t = \phi\left([\overrightarrow{H_t}, \overleftarrow{H_t}] W_o + b_o\right)$$

Here, $[\overrightarrow{H_t}, \overleftarrow{H_t}]$ represents the concatenation of the forward and backward states, W_o is the output weight matrix, and b_o is the output bias.

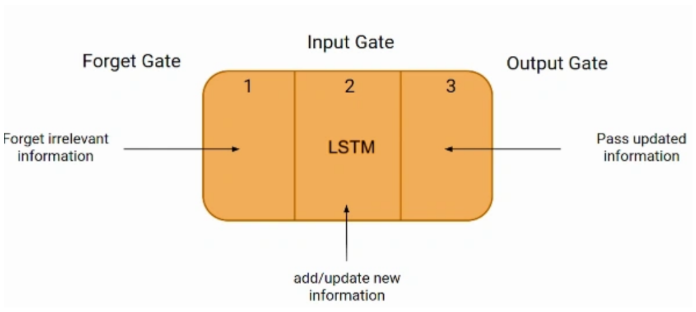
Thanks to this mechanism, BRNNs can look ahead and back in the sequence, helping the model to make more accurate predictions when working with tasks requiring surrounding context. BRNNs are particularly useful in tasks such as natural language processing, speech recognition, and many other related applications.

5 Long Short-Term Memory (LSTM)

In 1997, to address the vanishing gradient problem, Hochreiter and Schmidhuber introduced the Long Short-Term Memory (LSTM) model. LSTM is a variant of the traditional Recurrent Neural Network (RNN), with a key difference in using memory cells instead of regular nodes in the hidden layer.

These memory cells can store information across multiple time steps thanks to mechanisms controlling the flow of information through gates, including input gates, output gates, and forget gates. Specifically, a node in the memory cell has a self-loop with a weight of 1, ensuring that the gradient can propagate across many time steps without vanishing or exploding. This allows LSTM to perform better on tasks requiring long-term memory than traditional RNNs.

The following diagram shows the basic architecture of an LSTM cell, including three main gates: the *Forget Gate*, the *Input Gate*, and the *Output Gate*.

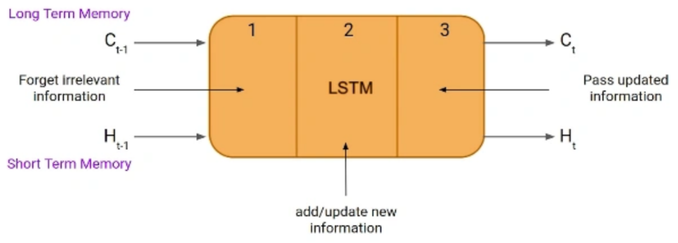


- **Forget Gate**: Determines which information to forget from the memory cell.
- **Input Gate**: Controls which information will be added to the memory cell.
- **Output Gate**: Determines which information from the memory cell will be outputted.

Thanks to these gating mechanisms, LSTMs can maintain stable gradients during backpropagation through time, allowing the model to learn and retain information from longer sequences.

5.1 The Logic Behind LSTM

In LSTMs, the hidden state is called *short-term memory*, and the cell state is called *long-term memory*. The diagram below illustrates how LSTMs retain and process information through different gates:



Interestingly, the cell state carries information across all time steps. This allows the LSTM to "remember" important information over a long period, something that traditional Recurrent Neural Networks (RNNs) struggle to maintain.

Example of LSTM Operation Let's consider an example to understand how LSTM works. Suppose we have two sentences:

- The first sentence is: "Bob is a nice person."

- The second sentence is: "Dan, on the other hand, is evil."

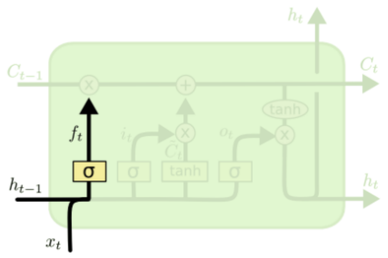
Clearly, in the first sentence, the subject we are talking about is Bob, but once we hit the period (.), we start talking about Dan in the second sentence. The LSTM needs to realize that after the period, the subject has changed, and it is no longer talking about Bob. Instead, the current subject is Dan.

This is where the *Forget Gate* of the LSTM comes into play. It allows the network to "forget" old information (about Bob) and update the new subject (about Dan). Specifically, each gate in the LSTM architecture has a specific role in processing information:

- **Forget Gate**: Discards information that is no longer relevant.
- **Input Gate**: Decides which information to add to the memory cell.
- **Output Gate**: Controls which information from the memory cell will be outputted for predictions at the current time step.

Thanks to this mechanism, LSTM can effectively handle complex and long-term sequential data, such as in machine translation, text classification, or time series prediction.

5.2 Forget Gate



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

The forget gate is one of the essential components of the LSTM architecture, helping the model decide which information to retain and which to discard from the memory cell. This process is critical for LSTMs to learn how to remove irrelevant or unnecessary information when processing long-term sequences.

At each time step t, the forget gate receives input consisting of the previous hidden state h_{t-1} and the current input x_t, then applies a sigmoid activation function (σ) to produce the forget vector f_t. The mathematical formula for this process is as follows:

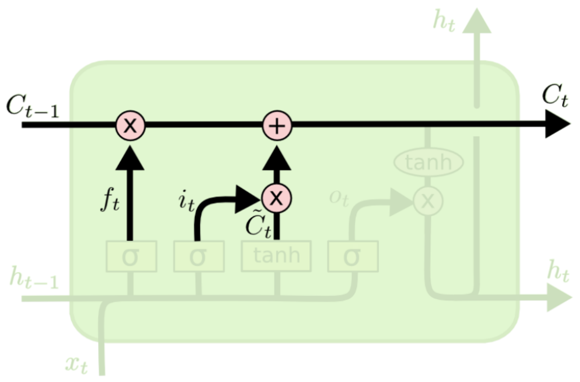
$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

Where:

- f_t is the vector that determines which information to forget, with values ranging between 0 and 1.
- W_f is the weight matrix applied to h_{t-1} and x_t.
- b_f is the bias term.
- The sigmoid activation function (σ) converts the output into probabilities, helping to control the extent of information retained or discarded.

The closer f_t is to 1, the more the corresponding information from the previous time step will be retained. Conversely, a value close to 0 will result in the corresponding information being discarded from the memory cell. This enables the LSTM to automatically learn how to filter important information during training, allowing the model to efficiently process long-term sequences and discard irrelevant information.

5.3 Input Gate



The input gate in LSTM decides which information will be added to the memory cell at the current time step. This is a critical phase in updating new information to the long-term memory C_t.

At each time step t, the input gate considers two sources of information: the previous hidden state h_{t-1} and the current input x_t. These two sources are combined through weights and biases, then passed through a sigmoid activation function (σ) to produce the input gate vector i_t. The formula for the input gate is:

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

Where:

- i_t is the input vector that determines the extent to which new information will be added to the memory cell.

- W_i is the weight matrix applied to the hidden state h_{t-1} and current input x_t.
- b_i is the bias term.
- The sigmoid activation function (σ) outputs probabilities between 0 and 1.

After computing i_t, a new value $\tilde{C_t}$ is generated from x_t and h_{t-1}, which serves as the candidate for updating the memory cell. The formula for $\tilde{C_t}$ is:

$$\tilde{C_t} = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

Where:

- $\tilde{C_t}$ is the candidate cell state value.
- W_C is the weight matrix for the hidden state h_{t-1} and input x_t.
- b_C is the bias term related to the candidate cell state calculation.
- The tanh function helps regulate the candidate cell state value to lie between -1 and 1.

Next, the final cell state value at the current time step C_t will be a combination of the old information (processed through the forget gate f_t) and the newly updated information from the input gate. The overall formula is:

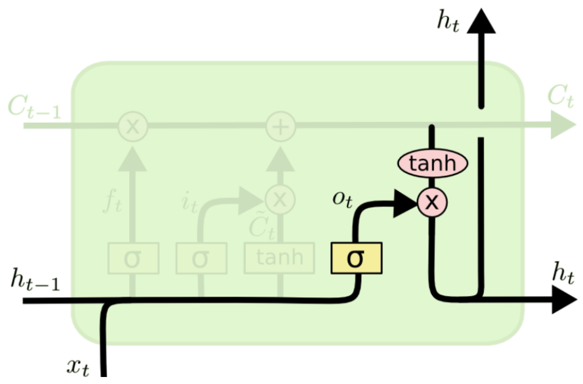
$$C_t = f_t * C_{t-1} + i_t * \tilde{C_t}$$

Where:

- C_t is the updated memory cell state.
- $f_t * C_{t-1}$ represents the old information filtered by the forget gate.
- $i_t * \tilde{C_t}$ represents the new information added to the memory cell through the input gate.

Thanks to the input gate, the LSTM can add new information to the memory cell in a controlled manner, allowing the model to learn and retain important information at each time step.

5.4 Output Gate



The output gate is the final component in the LSTM architecture, determining the new hidden state h_t at the current time step. This gate is responsible for controlling which information from the memory cell will be outputted for predictions or further calculations.

At each time step t, the output gate combines the previous hidden state h_{t-1} and the current input x_t, then applies a sigmoid activation function (σ) to calculate the output gate vector o_t. The formula for o_t is:

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

Where:

- o_t is the output vector at time step t, determining which information from the memory cell will be outputted.
- W_o is the weight matrix applied to h_{t-1} and x_t.
- b_o is the bias term related to the output gate calculation.
- The sigmoid activation function (σ) produces values between 0 and 1, determining the extent of information from the memory cell to be transferred to the hidden state h_t.

After calculating o_t, it is multiplied by the tanh function of the memory cell state C_t, to produce the new hidden state h_t at the current time step. The formula for the hidden state h_t is:

$$h_t = o_t * \tanh(C_t)$$

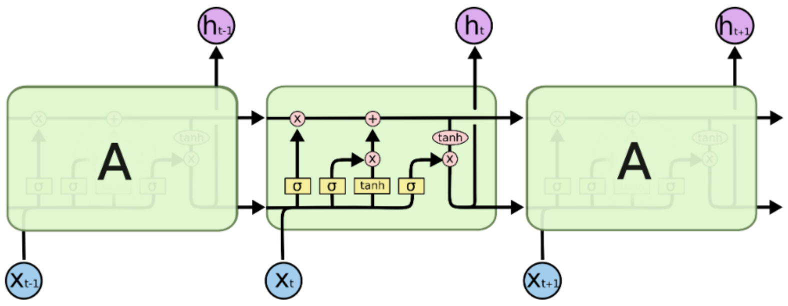
Where:

- h_t is the new hidden state, representing the information to be outputted at time step t.

- C_t is the updated memory cell state from previous gates (forget gate and input gate).
- The tanh function converts the memory cell state C_t into a value between -1 and 1, then multiplies it by o_t to produce the final hidden state h_t.

The output gate ensures that only relevant information from the memory cell will be outputted for use in subsequent predictions or for the next time step in the sequence. Thanks to this gate, LSTMs can flexibly select information to output, avoiding the storage or use of unnecessary information.

5.4.1 Overview



LSTM (Long Short-Term Memory) is not just a simple improvement of traditional Recurrent Neural Networks (RNNs), but a breakthrough in the field of time series and sequential data processing. By effectively addressing the **vanishing gradient** problem, LSTM opens up new possibilities for learning and retaining information in long sequences, which earlier models could not efficiently handle.

One of the most remarkable features of LSTM is its sophisticated yet intelligent operation mechanism, built through three main gates: **Forget Gate**, **Input Gate**, and **Output Gate**. These three gates work together in a delicate manner to precisely control the amount of information to be retained, discarded, or outputted at each time step. This mechanism allows LSTM to handle complex and long-term sequential data, where information from earlier time steps is crucial for future predictions.

The **Forget Gate** allows the model to discard irrelevant information, keeping only the important details, helping the system avoid "overloading." The **Input Gate** determines which information will be added to the memory, ensuring that the model can continuously learn and update new information. Finally, the **Output Gate** regulates the output of information from the memory at each time step, allowing LSTM to make the most accurate predictions.

The diagram above provides an overview of LSTM, showing how the model receives input X_t and the previous hidden state h_{t-1}, then processes the information through the gates to produce the new hidden state h_t for the next time step. Thanks to this complex mechanism, LSTM can maintain and exploit long-term relationships between data points in the sequence, something traditional RNNs cannot achieve.

In summary, LSTM is a powerful tool for tasks requiring long-term memory and information processing. Its ability to "selectively forget" and "selectively remember" has made it the foundation for many important applications such as machine translation, text classification, and time series prediction. LSTM not only overcomes the limitations of previous models but also opens up infinite potential for handling complex and extended sequential data.

6 Using Recurrent Neural Networks (RNN) and LSTM Models to Predict Stock Prices

6.1 Data Preparation

The stock data of Apple Inc. (AAPL) was collected from January 1, 2018, to August 27, 2024. This data includes important information on stock prices per trading day, and it was divided into three main stages for the purpose of training, validating, and testing the stock price prediction model. The features of the data include the opening price (*Open*), highest price during the day (*High*), lowest price during the day (*Low*), closing price (*Close*), adjusted closing price (*Adj Close*), and trading volume during the day (*Volume*). The model's target is to predict the opening price (*Open*).

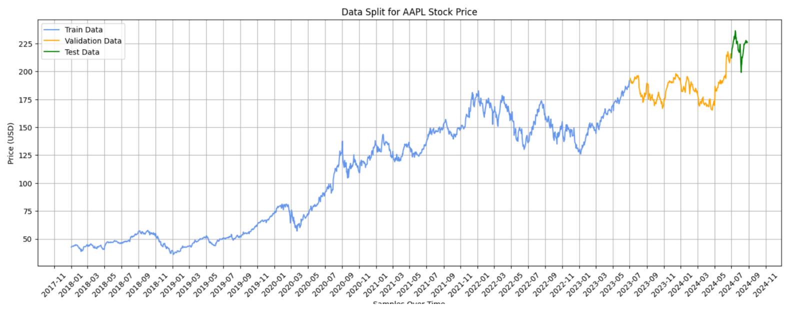
The data was divided into the following sets:

- **Training Set (Train Data)**: Contains data from January 1, 2018, to June 30, 2023. This set is used to train the prediction model.
- **Validation Set (Validation Data)**: Contains data from July 1, 2023, to June 30, 2024, used to evaluate model performance during training, to adjust model parameters.
- **Test Set (Test Data)**: Contains data from July 1, 2024, to August 27, 2024. This set is used to evaluate the final model performance after training.

The data is normalized using the **MinMaxScaler** method with a value range of 0 to 1. Specifically, all features (*Open*, *High*, *Low*, *Close*, *Adj Close*, *Volume*) are normalized to ensure the model works efficiently and avoids data imbalance.

The prediction model is trained and tested with a batch size of 32 (*batch size* = 32), meaning the model updates weights after every 32 data samples. Additionally, the sequence length used for the model is 32 (*sequence size* = 32), meaning the model learns from 32 consecutive time steps to predict the next time step.

A visual chart has been created to show the data split across different time stages. This chart illustrates Apple's stock opening prices (AAPL) over each stage with three distinct colors:



- **Light blue (cornflowerblue)**: Represents the training data.
- **Orange (orange)**: Represents the validation data.
- **Green (green)**: Represents the test data.

This chart allows easy observation of stock price fluctuations over time and the logical data split across time stages. The time markers on the horizontal axis are formatted by month and are evenly spaced to clearly display the trading cycle over each stage, thus supporting the training and testing of the prediction model effectively.

6.2 Complementary Method

During the training of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) models, several supplementary techniques were applied to improve the model's performance while preventing common issues like overfitting and optimizing the training process. The following complementary methods were used:

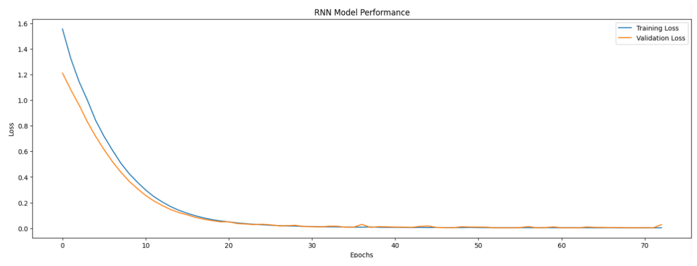
- **Early Stopping**: The *Early Stopping* technique helps monitor the training process and automatically stops when the model no longer improves on the validation set. Specifically, the model is set with the parameter `monitor='val_loss'` to track the loss on the validation set. With the parameter `patience=10`, if after 10 consecutive epochs the loss does not improve, the training process will stop. Additionally, the parameter `restore_best_weights=True` ensures that the model will restore the best weights from earlier epochs with the highest performance. This prevents the model from continuing to train after reaching peak performance, thus preventing overfitting.

- **Dropout**: The *Dropout* technique is applied with a rate of 0.2, meaning that 20% of the neurons will be "turned off" randomly during each training step. This helps the model avoid becoming too reliant on specific neurons and helps prevent overfitting. Dropout is known as an effective method for improving the generalization ability of the model, especially when working with deep neural networks.
- **Regularization**: To prevent the model from learning too much detail from the training data and to minimize overfitting, a regularization term (*kernel regularization*) with $L2$ normalization was applied. This regularization term is set to 0.01, to ensure that the model's weights do not become too large during training. The $L2$ method helps the model maintain smaller weights, thereby avoiding the model becoming too complex and reliant on features that are not truly important in the data.

Combining these complementary methods helps improve the model's prediction accuracy while minimizing the risk of encountering issues such as overfitting or difficulties in optimizing the training process. These techniques all play a crucial role in helping the model achieve better and more stable results in time series prediction tasks.

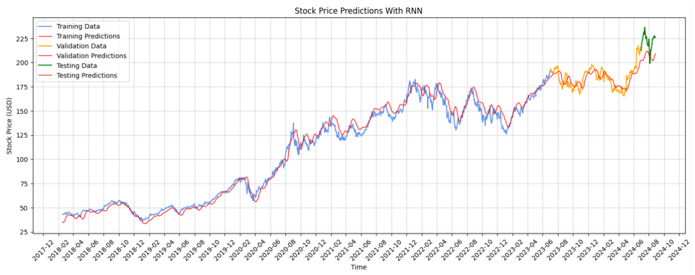
6.3 Result

6.3.1 RNN Model Results



Observation of Loss Function: The chart above illustrates the gradual reduction of the loss function during the training and validation phases of the RNN model. Both the training loss and validation loss curves show a rapid decrease in the early stages and gradually converge to a very low level after about 30 epochs. This indicates that the model learns very well from the training data without overfitting. This is also evident from the validation loss continuing to decrease without a sudden increase after epochs, demonstrating that the model has a strong generalization ability on new data.

The *early stopping* parameter was also applied to stop the training process when the model no longer showed significant improvement in loss, helping save training time and avoid unnecessary model updates.

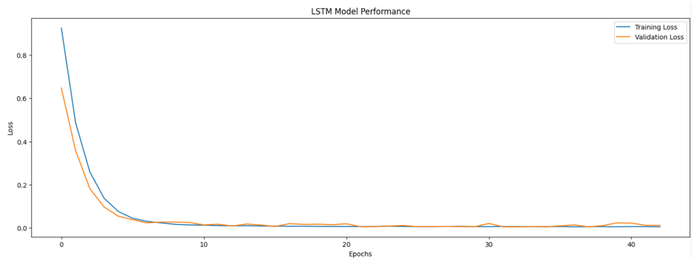


Observation of Prediction Results: The chart above shows a comparison between the actual data and the predicted results from the RNN model. The predicted results for the training, validation, and test sets all have a high level of alignment with the actual data.

- **For the training set**: The prediction curve (Training Predictions) is very well aligned with the actual data, indicating that the model has learned the main trends from the training data.
- **For the validation set**: The predicted results are also very close to the validation data, showing that the model not only learns well from the training data but also has the ability to make accurate predictions on new data.
- **For the test set**: Although the test period is shorter, the predicted results maintain high accuracy, especially in capturing large price fluctuations.

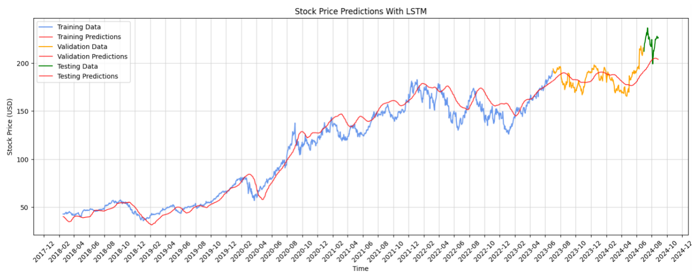
Overall, the RNN model shows strong prediction performance across all data stages (training, validation, and test), with small error margins and high similarity between actual and predicted data. This indicates that the model was well trained and has strong generalization capabilities on new data.

6.3.2 LSTM Model Results



Observation of LSTM Loss Function: The chart above shows the reduction of the loss function for both the training and validation sets of the LSTM model. Similar to the RNN model, the loss function decreases rapidly in the first epochs and reaches a very low level after about 10 to 20 epochs. However, the difference between the two curves (Training Loss and Validation Loss) is very small, indicating that the LSTM model learns well from the data without overfitting. The almost simultaneous reduction of both loss functions also shows that the model maintains good generalization performance on the validation set.

Compared to the RNN, LSTM has a faster convergence rate and achieves a lower loss function, demonstrating the LSTM's ability to remember long-term information and handle time series better.



Comparison Between RNN and LSTM Both RNN and LSTM models show good prediction performance on Apple's stock price data (AAPL); however, they also have their strengths and weaknesses:

Strengths of RNN:

- RNN models learn quickly and easily capture short-term sequences.
- RNNs have a simpler structure compared to LSTMs, which helps reduce computational workload and training time in tasks that do not require long-term memory.

Weaknesses of RNN:

- RNNs struggle to retain information across many time steps, as shown by weaker predictions for long sequences.
- The model is more susceptible to the *vanishing gradient* problem, leading to poor performance when working with long sequences of data.

Strengths of LSTM:

- LSTM excels at handling long sequences of data thanks to its memory cell structure and control gates such as the forget gate, input gate, and output gate.

- LSTM's long-term memory capabilities allow the model to capture longer trends in the data, as reflected in more accurate predictions on the test set.

Weaknesses of LSTM:

- LSTM has a more complex structure than RNN, leading to longer training times and requiring more computational resources.
- Although LSTM generally outperforms RNN in long-sequence tasks, RNN can be a better choice for short sequences or tasks that do not require long-term memory.

6.3.3 General Observations

From the results obtained, it is clear that both RNN and LSTM can effectively predict stock price trends, but LSTM demonstrates superior performance when handling long-term and more complex sequences of data. While RNNs learn quickly and are simpler, they struggle with retaining information over many time steps.

In time series tasks, LSTM is generally the better choice due to its ability to remember and process long-term information, while RNNs may be more suitable for tasks that require speed and short-sequence processing.