# CHAPTER 9: Recurrent Neural Networks

## Overview

In this chapter, you will learn how to handle real sequential data. You will extend your knowledge of **artificial neural network** (**ANN**) models and **recurrent neural network** (**RNN**) architecture for training sequential data. You will also learn how to build an RNN model with an LSTM layer for natural language processing.

By the end of this chapter, you will have gained hands-on experience of applying multiple LSTM layers to build RNNs for stock price predictions.

## Introduction

Sequential data refers to datasets in which each data point is dependent on the previous ones. Think of it like a sentence, which is composed of a sequence of words that are related to each other. A verb will be linked to a subject and an adverb will be related to a verb. Another example is a stock price, where the price on a particular day is related to the price of the previous days. Traditional neural networks are not fit for processing this kind of data. There is a specific type of architecture that can ingest sequences of data. This chapter will introduce you to such models—known as **recurrent neural networks** (**RNNs**).

An RNN model is a specific type of deep learning architecture in which the output of the model feeds back into the input. Models of this kind have their own challenges (known as vanishing and exploding gradients) that will be addressed later in the chapter.

In many ways, an RNN is a representation of how a brain might work. RNNs use memory to help them learn. But how can they do this if information only flows in one direction? To understand this, you'll need to first review sequential data. This is a type of data that requires a working memory to process data effectively. Until now, you have only explored non-sequential models, such as a perceptron or CNN. In this chapter, you will look at sequential models such as RNN, LSTM, or GRU.

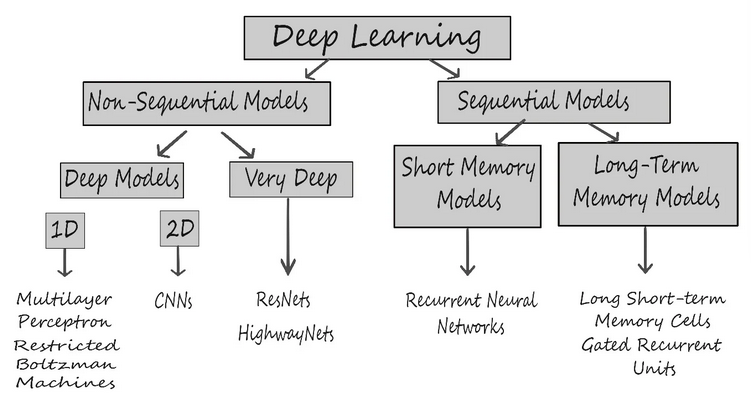


Figure 9.1: Sequential versus non-sequential models

## Sequential Data

Sequential data is information that happens in a sequence and is related to past and future data. An example of sequential data is time series data; as you perceive it, time only travels in one direction.

Suppose you have a ball (as in *Figure 9.2*), and you want to predict where this ball will travel next. If you have no prior information about the direction from which the ball was thrown, you will simply have to guess. However, if in addition to the ball's current location, you also had information about its previous location, the problem would be much simpler. To be able to predict the ball's next location, you need the previous location information in a sequential (or ordered) form to make a prediction about future events.



Figure 9.2: Direction of the ball

RNNs function in a way that allows the sequence of the information to retain value with the help of internal memory.

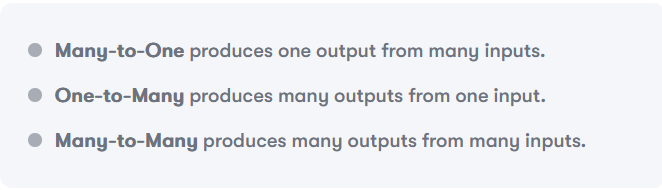
You'll look at some examples of sequential data in the following section.

## Examples of Sequential Data

Sequential data is a specific type of data where the order of each piece of information is important, and they all depend on each other.

One example of sequential data is financial data, such as stock prices. If you want to predict future data values for a given stock, you need to use previous values in time. In fact, you will work on stock prediction in *Exercise 9.01*, *Training an ANN for Sequential Data – APPLE Stock Prediction*.

Audio and text can also be considered sequential data. Audio can be split up into a sequence of sound waves, and text can be split up into sequences of either characters or words. The sound waves or sequences of characters or words should be processed in order to convey the desired result. Beyond these two examples that you encounter every day, there are many more examples in which sequential processing may be useful, from analyzing medical signals such as EEGs, projecting stock prices, and inferring and understanding genomic sequences. There are three categories of sequential data:



Diagram

Description automatically generated

Figure 9.3: Categories of sequential data

Consider another example. Suppose you have a language model with a sentence, or a phrase and you are trying to predict the word that comes next, as in the following figure:

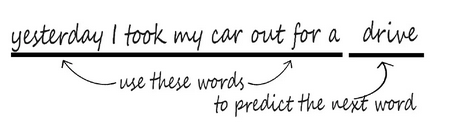


Figure 9.4: Sentence example

Say you're given the words yesterday I took my car out for a…, and you want to try to predict the next word, drive. One way you could do this is by building a deep neural network such as a feed-forward neural network. However, you would immediately run into a problem. A feed-forward network can only take a fixed-length input vector as its input; you have to specify the size of that input right from the start.

Because of this, your model needs a way to be able to handle variable-length inputs. One way you can do this is by using a fixed window. That means that you force your input vector to be just a certain length. For example, you can split the sentence into groups of two consecutive words (also called a bi-gram) and predict the next one. This means that no matter where you're trying to make that next prediction, your model will only be taking in the previous two words as its input. You need to consider how you can numerically represent this data. One way you can do this is by taking a fixed-length vector and allocating some space in that vector for the first word and some space in that vector for the second word. In those spaces, encode the identity of each word. However, this is problematic.

Why? Because you're using only a portion of the information available (that is, two consecutive words only). You have access to a limited window of data that doesn't give enough context to accurately predict what will be the next word. That means you cannot effectively model long-term dependencies. This is important in sentences like the one in Figure 9.5 where you clearly need information from much earlier in the sentence to be able to accurately predict the next word.

Figure 9.5: Sentence example


Figure 9.5: Sentence example

If you were only looking at the past two or three words, you wouldn't be able to make this next prediction, which you know is Italian. So, this means that you really need a way to integrate the information in the sentence from start to finish.

To do this, you could use a set of counts as a fixed-length vector and use the entire sentence. This method is known as bag of words.

You have a fixed-length vector regardless of the identity of the sentence, but what differs is adding the counts over this vocabulary. You can feed this into your model as an input to generate a prediction.

However, there's another big problem with this. Using just the counts means that you lose all sequential information and all information about the prior history.

Consider Figure 9.6. So, these two sentences, which have completely opposite semantic meanings would have the exact same representations in this bag of words format. This is because they have the exact same list of words, just in a different order. So, obviously, this isn't going to work. Another idea could be simply to extend the fixed window.

Figure 9.6: Bag of words example


Figure 9.6: Bag of words example

Now, consider Figure 9.7. You can represent your sentence in this way, feed the sentence into your model, and generate your prediction. The problem is that if you were to feed this vector into a feed-forward neural network, each of these inputs, yesterday I took my car, would have a separate weight connecting it to the network. So, if you were to repeatedly see the word yesterday at the beginning of the sentence, the network may be able to learn that yesterday represents a time or a setting. However, if yesterday were to suddenly appear later in that fixed-length vector, at the end of a sentence, the network may have difficulty understanding the meaning of yesterday. This is because the parameters that are at the end of a vector may never have seen the term yesterday before, and the parameters from the beginning of the sentence weren't shared across the entire sequence.

Figure 9.7: Sentence example


Figure 9.7: Sentence example

So, you need to be able to handle variable-length input and long-term dependencies, track sequential order, and have parameters that can be shared across the entirety of your sequence. Specifically, you need to develop models that can do the following:

* Handle variable-length input sequences.
* Track long-term dependencies in the data.
* Maintain information about the sequence's order.
* Share parameters across the entirety of the sequence.

How can you do this with a model where information only flows in one direction? You need a different kind of neural network. You need a recursive model. You will practice processing sequential data in the following exercise.

## Example 9.01: Training an ANN for Sequential Data – Apple Stock Prediction

In this exercise, you will build a simple ANN model to predict the Apple stock price. But unlike examples from previous chapters, this time the input data is sequential. So, you need to manually do some processing to create a dataset that will contain the price of the stock for a given day as the target variable and the price for the previous 9 years and 9 months as features. You are required to split the data into training and testing sets before and after the date 2022-07-01.

Note

You can find the AAPL.csv dataset [here](https://raw.githubusercontent.com/fenago/tf/main/Chapter9-Recurrent%20Neural%20Networks/datasets/AAPL.csv)

1. Open a new Jupyter or Colab notebook.
2. Import the libraries needed. Use numpy for computation, matplotlib for plotting

*import numpy as np*

*import matplotlib.pyplot as plt*

*import pandas as pd*

*from sklearn.preprocessing import StandardScaler, MinMaxScaler*

1. Use the read\_csv() function to read in the CSV file and store your dataset in a pandas DataFrame, data, for manipulation:

*import io*

*path = "https://raw.githubusercontent.com/fenago/tf/main/Chapter9-Recurrent%20Neural%20Networks/datasets/AAPL.csv"*

*data = pd.read\_csv(path)*

1. Call the head() function on your data to take a look at the first five rows of your DataFrame:

*data.head()*

You should get the following output:

Text

Description automatically generated

Figure 9.5: First five rows of output

The preceding tables shows the raw data. You can see that each row represents a day where you have information about the stock price when the market opened and closed, the highest price, the lowest price, and the adjusted close price of the stock (taking into account dividend or stock split, for instance).

1. Now, split the training data. Use all data that is older than 2022-07-01 using the Date column for your training data. Save it as data\_training. Save this in a separate file by using the copy() method:

*data\_training = data[data['Date']<'2022-07-01'].copy()*

1. Now, split the test data. Use all data that is more recent than or equal to 2022-07-01 using the Date column. Save it as data\_test. Save this in a separate file by using the copy() method:

*data\_test = data[data['Date']>='2022-07-01'].copy()*

1. Use drop( ) to remove your Date and Adj Close columns in your DataFrame. Remember that you used the Date column to split your training and test sets, so the date information is not needed. Use axis = 1 to specify that you also want to drop labels from your columns. To make sure it worked, call the head() function to take a look at the first five rows of the DataFrame:

*training\_data = data\_training.drop(['Date', 'Adj Close'], axis=1)*

*training\_data.head()*

You should get the following output:

A picture containing table

Description automatically generated

Figure 9.9: New training data

This is the output you should get after removing those two columns

1. Create a scaler from MinMaxScaler to scale training\_data to numbers between zero and one. Use the fit\_transform function to fit the model to the data and then transform the data according to the fitted model:

*scaler = MinMaxScaler()*

*training\_data = scaler.fit\_transform(training\_data)*

*training\_data*

You should get the following output:

Text, letter

Description automatically generated

1. Split your data into X\_train and y\_train datasets:

*X\_train = []*

*y\_train = []*

1. Check the shape of training\_data:

*training\_data.shape[0]*

You should get the following output:



You can see there are 2748 observations in the training set.

1. Create a training dataset that has the previous 60 days' stock prices so that you can predict the closing stock price for day 61. Here, X\_train will have two columns. The first column will store the values from 0 to 59, and the second will store values from 1 to 60. In the first column of y\_train, store the 61st value at index 60, and in the second column, store the 62nd value at index 61. Use a for loop to create data in 60 time steps:

*for i in range(60, training\_data.shape[0]):*

*X\_train.append(training\_data[i-60:i])*

*y\_train.append(training\_data[i, 0])*

1. Convert X\_train and y\_train into NumPy arrays:

*X\_train, y\_train = np.array(X\_train), np.array(y\_train)*

1. Call the shape( ) function on X\_train and y\_train:

*X\_train.shape, y\_train.shape*

You should get the following output:

Text

Description automatically generated

The preceding snippet shows that the prepared training set contains 2688 observations with 60 days of data for five features you kept (Open, Low, High, Close, and Volume)

1. Transform the data into a 2D matrix with the shape of the sample (the number of samples and the number of features in each sample). Stack the features for all 60 days on top of each other to get an output size of (2688, 300). Use the following code for this purpose:

*X\_old\_shape = X\_train.shape*

*X\_train = X\_train.reshape(X\_old\_shape[0], \*

*X\_old\_shape[1]\*X\_old\_shape[2])*

*X\_train.shape*

You should get the following output:

Text, whiteboard

Description automatically generated

1. Now, build an ANN, You will need some additional libraries for this. Use Sequential to initialize the neural net, Input to add an input layer, Dense to add a dense layer, and Dropout to help prevent overfitting:

*from tensorflow.keras import Sequential*

*from tensorflow.keras.layers import Input, Dense, Dropout*

1. Initialize the neural network by calling regressor\_ann = Sequential().

*regressor\_ann = Sequential()*

1. Add an input layer with shape as 300:

*regressor\_ann.add(Input(shape = (300,)))*

1. Then, add the first dense layer. Set it to 512 units, which will be your dimensionality for the output space. Use a ReLU activation function. Finally, add a dropout layer that will remove 20% of the units during training to prevent overfitting:

*regressor\_ann.add(Dense(units = 512, activation = 'relu'))*

*regressor\_ann.add(Dropout(0.2))*

1. Add another dense layer with 128 units, ReLU as the activation function, and a dropout of 0.3:

*regressor\_ann.add(Dense(units = 128, activation = 'relu'))*

*regressor\_ann.add(Dropout(0.3))*

1. Add another dense layer with 64 units, ReLU as the activation function, and a dropout of 0.4:

*regressor\_ann.add(Dense(units = 64, activation = 'relu'))*

*regressor\_ann.add(Dropout(0.4))*

1. Again, add another dense layer with 128 units, ReLU as the activation function, and a dropout of 0.3:

*regressor\_ann.add(Dense(units = 64, activation = 'relu'))*

*regressor\_ann.add(Dropout(0.4))*

1. Add a final dense layer with one unit:

*regressor\_ann.add(Dense(units = 1))*

1. Check the summary of the model:

*regressor\_ann.summary()*

You will get valuable information about your model layers and parameters.

Table

Description automatically generated

Figure 9.11: Model Summary

1. Use the compile() method to configure your model for training. Choose Adam as your optimizer and mean squared error to measure your loss function:

*regressor\_ann.compile(optimizer='adam', \*

*loss = 'mean\_squared\_error')*

1. Finally, fit your model and set it to run on 10 epochs. Set your batch size to 32:

*regressor\_ann.fit(X\_train, y\_train, epochs=10, batch\_size=32)*

You should get the following output:

Table

Description automatically generated

Figure 9.12: Training the model

1. Test and predict the stock price and prepare the dataset. Check your data by calling the head() method:

*data\_test.head()*

You should get the following output:

Table

Description automatically generated

Figure 9.13: First five rows of a DataFrame

1. Use the tail(60) method to create a past\_60\_days variable, which consists of the last 60 days of data in the training set. Add the past\_60\_days variable to the test data with the append() function. Assign True to ignore\_index:

*past\_60\_days = data\_training.tail(60)*

*df = past\_60\_days.append(data\_test, ignore\_index = True)*

1. Now, prepare your test data for predictions by repeating what you did for the training data in *steps 8* to *15*:

*df = df.drop(['Date', 'Adj Close'], axis = 1)*

*inputs = scaler.transform(df)*

*X\_test = []*

*y\_test = []*

*for i in range(60, inputs.shape[0]):*

*X\_test.append(inputs[i-60:i])*

*y\_test.append(inputs[i, 0])*

*X\_test, y\_test = np.array(X\_test), np.array(y\_test)*

*X\_old\_shape = X\_test.shape*

*X\_test = X\_test.reshape(X\_old\_shape[0], \*

*X\_old\_shape[1] \* X\_old\_shape[2])*

*X\_test.shape, y\_test.shape*

You should get the following output:

Text

Description automatically generated with low confidence

1. Test some predictions for your stock prices by calling the predict() method on X\_test:

*y\_pred = regressor\_ann.predict(X\_test)*

1. Before looking at the results, reverse the scaling you did earlier so that the number you get as output will be at the correct scale using the StandardScaler utility class that you imported with scaler.scale\_:

*scaler.scale\_*

You should get the following output:

Text

Description automatically generated

Figure 9.14: Using StandardScaler

1. Use the first value in the preceding array to set your scale in preparation for the multiplication of y\_pred and y\_test. Recall that you are converting your data back from your earlier scale, in which you converted all values to between zero and one:

*scale = 1/3.70274364e-03*

*scale*

You should get the following output:

Text

Description automatically generated

1. Multiply y\_pred and y\_test by scale to convert your data back to the proper values:

*y\_pred = y\_pred\*scale*

*y\_test = y\_test\*scale*

1. Review the real Apple stock price and your predictions:

*plt.figure(figsize=(14,5))*

*plt.plot(y\_test, color = 'black', label = "Real Apple Stock Price")*

*plt.plot(y\_pred, color = 'gray',\*

*label = 'Predicted Apple Stock Price')*

*plt.title('Apple Stock Price Prediction')*

*plt.xlabel('time')*

*plt.ylabel('Apple Stock Price')*

*plt.legend()*

*plt.show()*

You should get the following output:

Chart, line chart

Description automatically generated

Figure 9.15: Real Apple stock price versus your predictions

In the preceding graph, you can see that your trained model is able to capture some of the trends of the Apple stock price. Observe that the predictions are quite different from the real values. It is evident from this result that ANNs are not suited for sequential data.

In this exercise, you saw the inability of simple ANNs to deal with sequential data. In the next section, you will learn about recurrent neural networks, which are designed to learn from the temporal dimensionality of sequential data. Then, in *Exercise 9.02*, *Building an RNN with LSTM Layer Apple Stock Prediction*, you will perform predictions on the same Apple stock price dataset using RNNs and compare your results.

## Recurrent Neural Networks

The first formulation of a recurrent-like neural network was created by John Hopfield in 1982. He had two motivations for doing so:

* Sequential processing of data
* Modeling of neuronal connectivity

Essentially, an RNN processes input data at each time step and stores information in its memory that will be used for the next step. Information is first transformed into vectors that can be processed by machines. The RNN then processes the vector sequence one at a time. As it processes each vector, it passes the previous hidden state. The hidden state retains information from the previous step, acting as a type of memory. It does this by combining the input and the previous hidden state with a tanh function that compresses the values between -1 and 1.

Essentially, this is how the RNN functions. RNNs don't need a lot of computation and work well with short sequences.

Diagram

Description automatically generated

Figure 9.16: RNN data flow

Now turn your attention to applying neural networks to problems that involve sequential processing of data. You've already learned a bit about why these sorts of tasks require a fundamentally different type of network architecture from what you've seen so far.

## RNN Architecture

This section will go through the key principles behind RNNs, how they are fundamentally different from what you've learned so far, and how RNN computation actually works.

But before you do that, take one step back and consider the standard feed-forward neural network that was discussed previously.

In feed-forward neural networks, data propagates in one direction only, that is, from input to output.

Therefore, you need a different kind of network architecture to handle sequential data. RNNs are particularly well-suited to handling cases in which you have a sequence of inputs rather than a single input. These are great for problems in which a sequence of data is being propagated to give a single output.

For example, imagine that you are training a model that takes a sequence of words as input and outputs an emotion associated with that sequence. Similarly, consider cases in which, instead of returning a single output, you could have a sequence of inputs and propagate them through your network, where each time step in the sequence generates an output.

Simply put, RNNs are networks that offer a mechanism to persist previously processed data over time and use it to make future predictions.

Figure 9.17: RNN computation


Figure 9.17: RNN computation

In the preceding diagram, at some time step denoted by t, the RNN takes in Xt as the input, and at that time step, it computes a prediction value, Yt, which is the output of the network.

In addition to that output, it saved an internal state, called update, Ht. This internal state from time step t can then be used to complement the input of the next time step t+1. So, basically, it provides information about the previous step to the next one. This mechanism is called recurrent because information is being passed from one time step to the next within the network.

What's really happening here? This is done by using a simple recurrence relation to process the sequential data. RNNs maintain internal state, Ht, and combine it with the next input data, Xt+1, to make a prediction, Yt+1, and store the new internal state, Ht+1. The key idea is that the state update is a combination of the previous state time step as well as the current input that the network is receiving.

It's important to note that, in this computation, it's the same function f of W and the same set of parameters that are used at every time step, and it's those sets of parameters that you learn during the course of training. To get a better sense of how these networks work, step through the RNN algorithm:

1. You begin by initializing your RNN and the hidden state of that network. You can denote a sentence for which you are interested in predicting the next word. The RNN computation simply consists of them looping through the words in this sentence.
2. At each time step, you feed both the current word that you're considering, as well as the previous hidden state of your RNN into the network. This can then generate a prediction for the next word in the sequence and use this information to update its hidden state.
3. Finally, after you've looped through all the words in the sentence, your prediction for that missing word is simply the RNN's output at that final time step.

As you can see in the following diagram, this RNN computation includes both the internal state update and the formal output vector.

Figure 9.18: RNN data flow


Figure 9.18: RNN data flow

Given the input vector, Xt, the RNN applies a function to update its hidden state. This function is simply a standard neural net operation. It consists of multiplication by a weight matrix and the application of a non-linearity activation function. The key difference is that, in this case, you're feeding in both the input vector, Xt, and the previous state as inputs to this function, Ht-1.

Next, you apply a non-linearity activation function such as tanh to the previous step. You have these two weight matrices, and finally, your output, yt, at a given time step is then a modified, transformed version of this internal state.

After you've looped through all the words in the sentence, your prediction for that missing word is simply the RNN's output at that final time step, after all the words have been fed through the model. So, as mentioned, RNN computation includes both internal state updates and formal output vectors.

Another way you can represent RNNs is by unrolling their modules over time. You can think of RNNs as having multiple copies of the same network, where each passes a message on to its descendant.

Figure 9.19: Computational graph with time


Figure 9.19: Computational graph with time

In this representation, you can make your weight matrices explicit, beginning with the weights that transform the input to the H weights that are used to transform the previous hidden state to the current hidden state, and finally the hidden state to the output.

It's important to note that you use the same weight matrices at every time step. From these outputs, you can compute a loss at each time step. The computation of the loss will then complete your forward propagation through the network. Finally, to define the total loss, you simply sum the losses from all of the individual time steps. Since your loss is dependent on each time step, this means that, in training the network, you will have to also involve time as a component.

Now that you've got a bit of a sense of how these RNNs are constructed and how they function, you can walk through a simple example of how to implement an RNN from scratch in TensorFlow.

The following snippet uses a simple RNN from keras.models.Sequential. You specify the number of units as 1 and set the first input dimension to None as an RNN can process any number of time steps. A simple RNN uses tanh activation by default:

*model = keras.models.Sequential([*

*keras.layers.SimpleRNN\*

*(1, input\_shape=[None, 1])*

*])*

The preceding code creates a single layer with a single neuron.

That was easy enough. Now you need to stack some additional recurrent layers. The code is similar, but there is a key difference here. You will notice return\_sequences=True on all but the last layer. This is to ensure that the output is a 3D array. As you can see, the first two layers each have 20 units:

*model = keras.models.Sequential\*

*([Keras.layers.SimpleRNN\*

*(20, return\_sequences=True, input\_shape=[None, 1]), \*

*Keras.layers.SimpleRNN(20, return\_sequences=True), \*

*Keras.layers.SimpleRNN(1)])*

The RNN is defined as a layer, and you can build it by inheriting it from the layer class. You can also initialize your weight matrices and the hidden state of your RNN cell to zero.

The key step here is defining the call function, which describes how you make a forward pass through the network given an input X. And, to break down this call function, you would first update the hidden state according to the equation discussed previously.

Take the previous hidden state and the input X, multiply them by the relevant weight matrices, add them together, and then pass them through a non-linearity, like a hyperbolic tangent (tanh).

Then, the output is simply a transformed version of the hidden state, and at each time step, you return both the current output and the updated hidden state.

TensorFlow has made it easy by having a built-in dense layer. The same applies to RNNs. TensorFlow has implemented these types of RNN cells with the simple RNN layer. But this type of layer has some limitations, such as vanishing gradients. You will look at this problem in the next section before exploring different types of recurrent layers.

## Vanishing Gradient Problem

If you take a closer look at how gradients flow in this chain of repeating modules, you can see that between each time step you need to perform matrix multiplication. That means that the computation of the gradient—that is, the derivative of the loss with respect to the parameters, tracing all the way back to your initial state—requires many repeated multiplications of this weight matrix, as well as repeated use of the derivative of your activation function.

You can have one of two scenarios that could be particularly problematic: the exploding gradient problem or the vanishing gradient problem.

The exploding gradients problem is when gradients become continuously larger and larger due to the matrix multiplication operation, and you can't optimize them anymore. One way you may be able to mitigate this is by performing what's called gradient clipping. This amounts to scaling back large gradients so that their values are smaller and closer to 1.

You can also have the opposite problem where your gradients are too small. This is what is known as the vanishing gradient problem. This is when gradients become increasingly smaller (close to 0) as you make these repeated multiplications, and you can no longer train the network. This is a very real problem when it comes to training RNNs.

For example, consider a scenario in which you keep multiplying a number by some number that's in between zero and one. As you keep doing this repeatedly, that number is constantly shrinking until, eventually, it vanishes and becomes 0. When this happens to gradients, it's hard to propagate errors further back into the past because the gradients are becoming smaller and smaller.

Consider the earlier example from the language model where you were trying to predict the next word. If you're trying to predict the last word in the following phrase, it's relatively clear what the next word is going to be. There's not that much of a gap between the key relevant information, such as the word "fish," and the place where the prediction is needed.

Figure 9.20: Word prediction 


Figure 9.20: Word prediction

However, there are other cases where more context is necessary, like in the following example. Information from early in the sentence, She lived in Spain, suggests that the next word of the sentence after she speaks fluent is most likely the name of a language, Spanish.

Figure 9.21: Sentence example


Figure 9.21: Sentence example

But you need the context of Spain, which is located at a much earlier position in this sentence, to be able to fill in the relevant gaps and identify which language is correct. As this gap between words that are semantically important grows, RNNs become increasingly unable to connect the dots and link these relevant pieces of information together. That is due to the vanishing gradient problem.

How can you alleviate this? The first trick is simple. You can choose either tanh or sigmoid as your activation function. Both of these functions have derivatives that are less than 1.

Another simple trick you can use is to initialize the weights for the parameters of your network. It turns out that initializing the weights to the identity matrix helps prevent them shrinking to zero too rapidly during back-propagation.

But the final and most robust solution is to use a slightly more complex recurrent unit that can track long-term dependencies in the data more effectively. It can do this by controlling what information is passed through and what information is used to update its internal state. Specifically, this is the concept of a gated cell, like in the LSTM layer, which is the focus of the next section.

## Long Short-Term Memory Network

LSTMs are well-suited to learning long-term dependencies and overcoming the vanishing gradient problem. They are very performant models for sequential data, and they're widely used by the deep learning community.

LSTMs have a chain-like structure. In an LSTM, the repeating unit contains different interacting layers. The key point is that these layers interact to selectively control the flow of information within the cell.

The key building block of the LSTM is a structure called a gate, which functions to enable the LSTM to selectively add or remove information from its cell state. Gates consist of a neural net layer like a sigmoid.

Figure 9.22: LSTM architecture


Figure 9.22: LSTM architecture

Take a moment to think about what a gate like this would do in an LSTM. In this case, the sigmoid function would force its input to be between 0 and 1. You can think of this mechanism as capturing how much of the information that's passed through the gate should be retained. It's between zero and one. This effectively gates the flow of information.

LSTMs process information through four simple steps:

1. The first step in the LSTM is to decide what information is going to be thrown away from the cell state, to forget irrelevant history. This is a function of both the prior internal state, Ht-1, and the input, Xt, because some of that information may not be important.
2. Next, the LSTM decides what part of the new information is relevant and uses this to store this information in its cell state.
3. Then, it takes both the relevant parts of the prior information, as well as the current input, and uses this to selectively update its cell state.
4. Finally, it returns an output, and this is known as the output gate, which controls what information encoded in the cell state is sent to the network.

Figure 9.23: LSTM processing steps


Figure 9.23: LSTM processing steps

The key takeaway here for LSTMs is the sequence of how they regulate information flow and storage. Once again, LSTMs operate as follows:

* Forgetting irrelevant history
* Storing what's new and what's important
* Using its internal memory to update the internal state
* Generating an output

An important property of LSTMs is that all these different gating and update mechanisms work to create an internal cell state, C, which allows the uninterrupted flow of gradients through time. You can think of it as sort of a highway of cell states where gradients can flow uninterrupted. This enables you to alleviate and mitigate the vanishing gradient problem that's seen with standard RNNs.

LSTMs are able to maintain this separate cell state independently of what is output, and they use gates to control the flow of information by forgetting irrelevant history, storing relevant new information, selectively updating their cell state, and then returning a filtered version as the output.

The key point in terms of training and LSTMs is that maintaining the separate independent cell state allows the efficient training of an LSTM to backpropagate through time, which is discussed later.

Now that you've gone through the fundamental workings of RNNs, the backpropagation through time algorithm, and a bit about the LSTM architecture, you can put some of these concepts to work in the following example.

Consider the following LSTM model:

*regressor = Sequential()*

*regressor.add(LSTM(units= 50, activation = 'relu', \*

*return\_sequences = True, \*

*input\_shape = (X\_train.shape[1], 5)))*

*regressor.add(Dropout(0.2))*

*regressor.add(LSTM(units= 60, activation = 'relu', \*

*return\_sequences = True))*

*regressor.add(Dropout(0.3))*

*regressor.add(LSTM(units= 80, activation = 'relu', \*

*return\_sequences = True))*

*regressor.add(Dropout(0.4))*

*regressor.add(LSTM(units= 120, activation = 'relu'))*

*regressor.add(Dropout(0.5))*

*regressor.add(Dense(units = 1))*

First, you have initialized a neural network by calling regressor = Sequential(). Again, it's important to note that in the last line you omit return\_sequences = True because it is the final output:

*regressor = Sequential()*

Then, the LSTM layer is added. In the first instance, set the LSTM layer to 50 units. Use a relu activation function and specify the shape of the training set. Finally, the dropout layer is added with regressor.add(Dropout(0.2). The 0.2 means that 20% of the layers will be removed. Set return\_sequences = True, which allows the return of the last output.

Similarly, add three more LSTM layers and one dense layer to the LSTM model.

Now that you are familiar with the basic concepts surrounding working with sequential data, it's time to complete the following exercise using some real data.

## Exercise 9.02: Building an RNN with an LSTM Layer – Apple Stock Prediction

In this exercise, you will be working on the same dataset as for *Exercise 9.01*, *Training an ANN for Sequential Data – Apple Stock Prediction*. You will still try to predict the Apple stock price based on the data of the previous 60 days. But this time, you will be training an LSTM model. You will need to split the data into training and testing sets before and after the date 2022-07-01.

You will need to prepare the dataset like in Exercise 9.01, Training an ANN for Sequential Data – Apple Stock Prediction (steps 1 to 15) before applying the following code:

1. Start building the LSTM. You will need some additional libraries for this. Use Sequential to initialize the neural net, Dense to add a dense layer, LSTM to add an LSTM layer, and Dropout to help prevent overfitting:

*from tensorflow.keras import Sequential*

*from tensorflow.keras.layers import Dense, LSTM, Dropout*

1. Initialize the neural network by calling regressor = Sequential(). Add four LSTM layers with 50, 60, 80, and 120 units each. Use a ReLU activation function and assign True to return\_sequences for all but the last LSTM layer. Provide the shape of your training set to the first LSTM layer. Finally, add dropout layers with 20%, 30%, 40%, and 50% dropouts:

*regressor = Sequential()*

*regressor.add(LSTM(units= 50, activation = 'relu',\*

*return\_sequences = True,\*

*input\_shape = (X\_train.shape[1], 5)))*

*regressor.add(Dropout(0.2))*

*regressor.add(LSTM(units= 60, activation = 'relu', \*

*return\_sequences = True))*

*regressor.add(Dropout(0.3))*

*regressor.add(LSTM(units= 80, activation = 'relu', \*

*return\_sequences = True))*

*regressor.add(Dropout(0.4))*

*regressor.add(LSTM(units= 120, activation = 'relu'))*

*regressor.add(Dropout(0.5))*

*regressor.add(Dense(units = 1))*

1. Check the summary of the model using the summary() method:

*regressor.summary()*

You should get the following output:

Table

Description automatically generated

Figure 9.24: Model Summary

As you can see from the preceding figure, the summary provides valuable information about all model layers and parameters. This is a good way to make sure that your layers are in the order you wish and that they have the proper output shapes and parameters.

1. Use the compile() method to configure your model for training. Choose Adam as your optimizer and mean squared error to measure your loss function:

*regressor.compile(optimizer='adam', loss = 'mean\_squared\_error')*

1. Fit your model and set it to run on 10 epochs. Set your batch size equal to 32:

*regressor.fit(X\_train, y\_train, epochs=10, batch\_size=32)*

You should get the following output:

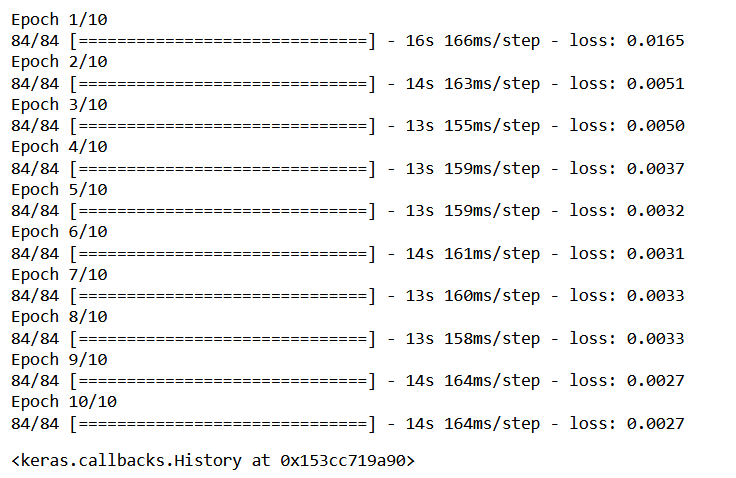


Figure 9.25: Training the model

1. Test and predict the stock and prepare the dataset. Check your data by calling the head( ) function:

*data\_test.head()*

You should get the following output:

Table

Description automatically generated

Figure 9.26: First five rows of the DataFrame

1. Call the tail(60) method to look at the last 60 days of data. You will use this information in the next step:

*data\_training.tail(60)*

You should get the following output:



Figure 9.27: rows of the DataFrame

1. Use the tail(60) method to create the past\_60\_days variable:

*past\_60\_days = data\_training.tail(60)*

1. Add the past\_60\_days variable to your test data with the append() function. Set True to ignore\_index. Drop the Date and Adj Close columns as you will not need that information:

*df = past\_60\_days.append(data\_test, ignore\_index = True)*

*df = df.drop(['Date', 'Adj Close'], axis = 1)*

1. Check the DataFrame to make sure that you successfully dropped Date and Adj Close by using the head() function:

*df.head()*

You should get the following output:

Table

Description automatically generated

Figure 9.28: Checking the first five rows of the DataFrame

1. Use scaler.transform from StandardScaler to perform standardization on inputs:

*inputs = scaler.transform(df)*

*inputs*

You should get the following output:

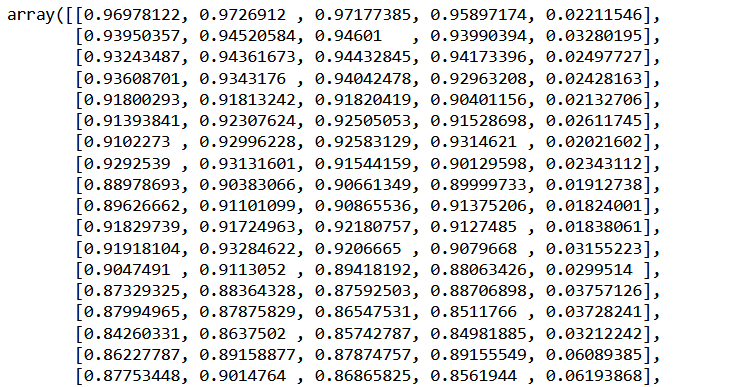


Figure 9.29: DataFrame standardization

From the preceding results, you can see that after standardization, all the values are close to 0 now.

1. Split your data into X\_test and y\_test datasets. Create a test dataset that has the previous 60 days' stock prices, so that you can test the closing stock price for the 61st day. Here, X\_test will have two columns. The first column will store the values from 0 to 59. The second column will store values from 1 to 60. In the first column of y\_test, store the 61st value at index 60, and in the second column, store the 62nd value at index 61. Use a for loop to create data in 60 time steps:

*X\_test = []*

*y\_test = []*

*for i in range(60, inputs.shape[0]):*

*X\_test.append(inputs[i-60:i])*

*y\_test.append(inputs[i, 0])*

1. Convert X\_test and y\_test into NumPy arrays:

*X\_test, y\_test = np.array(X\_test), np.array(y\_test)*

*X\_test.shape, y\_test.shape*

You should get the following output:



The preceding result shows that there are 20 observations and for each of them you have the last 60 days' data for the following five features: Open, High, Low, Close, and Volume. The target variable, on the other hand, contains 391 values.

1. Test some predictions for stock prices by calling regressor.predict(X\_test):

*y\_pred = regressor.predict(X\_test)*

1. Before looking at the results, reverse the scaling you did earlier so that the number you get as output will be at the correct scale using the StandardScaler utility class that you imported with scaler.scale\_:

*scaler.scale\_*

You should get the following output:

A picture containing text

Description automatically generated

Figure 9.30: Using StandardScaler

1. Use the first value in the preceding array to set your scale in preparation for the multiplication of y\_pred and y\_test. Recall that you are converting your data back from the scale you did earlier when converting all values to between zero and one:

*scale = 1/3.70274364e-03*

*scale*

You should get the following output:

A picture containing text

Description automatically generated

1. Multiply y\_pred and y\_test by scale to convert your data back to the proper values:

*y\_pred = y\_pred\*scale*

*y\_test = y\_test\*scale*

1. Use y\_pred to view predictions for Apple stock:

*y\_pred*

You should get the following output:

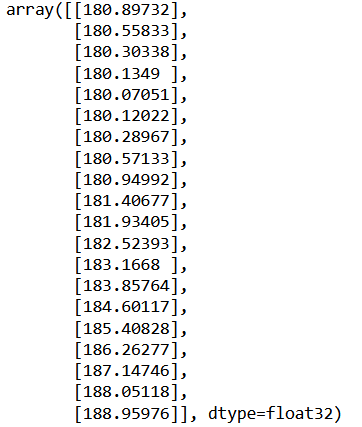


Figure 9.31: Checking prediction

The preceding results show the predicted APPLE stock price for the future dates.

1. Plot the real Apple stock price and your predictions:

*plt.figure(figsize=(14,5))*

*plt.plot(y\_test, color = 'black', label = "Real Apple Stock Price")*

*plt.plot(y\_pred, color = 'gray',\*

*label = 'Predicted Apple Stock Price')*

*plt.title(‘Apple Stock Price Prediction')*

*plt.xlabel('time')*

*plt.ylabel(Apple Stock Price')*

*plt.legend()*

*plt.show()*

You should get the following output:

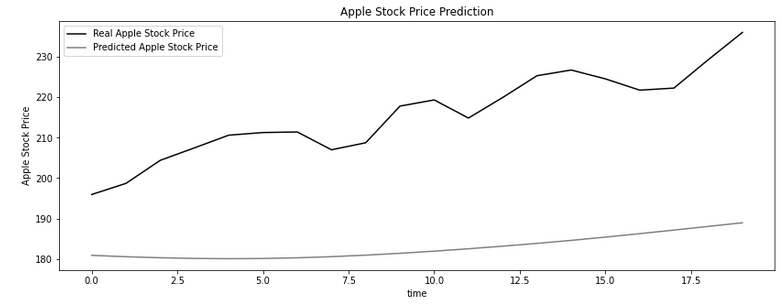


Figure 9.32: Apple stock price visualization

As you can see from the gray line in *Figure 9.32*, your prediction model is not much accurate, when compared to the actual stock price, which is shown by the black line. That might be due to the number of epochs we have trained are less. We can increase the number of epochs and check the performance of the model.

In this exercise, you built an RNN with an LSTM layer for Apple stock prediction and completed the training, testing, and prediction steps.

Now, test the knowledge you've gained so far in this chapter in the following activity.

## Activity 9.01: Building an RNN with Multiple LSTM Layers to Predict Appliances Energy Consumption.

The “**Appliances Energy Consumption”** dataset contains information related to energy consumption of appliances.

You are tasked with adapting an RNN model with additional LSTM layers to predict household energy consumption. You will be building an RNN model with three LSTM layers.

Note

You can find the dataset [here](https://raw.githubusercontent.com/fenago/tf/main/Chapter9-Recurrent%20Neural%20Networks/datasets/energydata_complete.csv).

Follow these steps will help you to complete this activity:

1. Load the data.
2. Convert the datatype of ‘date’ column from object to date and time.
3. Drop the unwanted columns and keep only Appliances, lights, T\_outs, RH\_1 and Visibility.
4. Standardize the selected columns.
5. Reshape the data for a given day to include the previous 60 days values.
6. Split the data into training and testing sets with, respectively, data before and after the index 19000, which corresponds to the last month of data.
7. Define and train an RNN model composed of three different layers of LSTM with 20, 40 and 80 units, followed by 50% dropout and ReLU as the activation function.
8. Make predictions on the testing set with the trained model.
9. Compare the predictions against the actual values on the entire dataset.

You should get the following output:

Chart, line chart

Description automatically generated

Figure 9.1: Expected output of the activity

Note

The solution to this activity can be found via this [link](https://github.com/fenago/tf/blob/main/Chapter9-Recurrent%20Neural%20Networks/Activity%209.01-%20Building%20an%20RNN%20with%20Multiple%20LSTM%20Layers%20to%20Predict%20Appliances%20Energy%20Consumption.ipynb)

In the next section, you will learn how to apply RNNs to text.

## Natural Language Processing

**Natural Language Processing** (**NLP**) is a quickly growing field that is both challenging and rewarding. NLP takes valuable data that has traditionally been very difficult for machines to make sense of and turns it into information that can be used. This data can take the form of sentences, words, characters, text, and audio, to name a few. Why is this such a difficult task for machines? To answer that question, consider the following examples.

Recall the two sentences: *it is what it is* and *is it what it is*. These two sentences, though they have completely opposite semantic meanings, would have the exact same representations in this bag of words format. This is because they have the exact same words, just in a different order. So, you know that you need to use a sequential model to process this, but what else? There are several tools and techniques that have been developed to solve these problems. But before you get to that, you need to learn how to preprocess sequential data.

## Data Preprocessing

As a quick review, preprocessing generally entails all the steps needed to train your model. Some common steps include data cleaning, data transformation, and data reduction. For natural language processing, more specifically, the steps could be all, some, or none of the following:

* Tokenization
* Padding
* Lowercase conversion
* Removing stop words
* Removing punctuation
* Stemming

The following sections provide a more in-depth description of the steps that you will be using. For now, here's an overview of each step:

* **Dataset cleaning** encompasses the conversion of case to lowercase, the removal of punctuation marks, and so on.
* **Tokenization** is breaking up a character sequence into specified units called tokens.
* **Padding** is a way to make input sentences of different sizes the same by padding them. Padding the sequences means ensuring that the sequences have a uniform length.
* **Stemming** is truncating words down to their stem. For example, the words "rainy" and "raining" both have the stem "rain".

## Dataset Cleaning

Here, you create the clean\_text function, which returns a list containing words once it has been cleaned. You will save all text as lowercase with lower() and encode it with utf8 for character standardization:

*def clean\_text(txt):*

*txt = "".join(v for v in txt if v not in string.punctuation)\*

*.lower()*

*txt = txt.encode("utf8").decode("ascii",'ignore')*

*return txt*

*corpus = [clean\_text(x) for x in all\_headlines]*

### **Generating a Sequence and Tokenization**

TensorFlow provides a dedicated class for generating a sequence of N-gram tokens – Tokenizer from keras.preprocessing.text:

*from keras.preprocessing.text import Tokenizer*

*tokenizer = Tokenizer()*

Once you have instantiated a Tokenizer(), you can use the fit\_on\_texts() method to extract tokens from a corpus. This step will attribute an integer index to each unique word from the corpus:

*tokenizer.fit\_on\_texts(corpus)*

After the tokenizer has been trained on a corpus, you can access the indexes allocated to each word from your corpus with the word\_index attribute:

*tokenizer.word\_index*

You can convert a sentence into a tokenized version using the texts\_to\_sequences() method:

*tokenizer.texts\_to\_sequences([sentence])*

You can create a function that will generate an N-gram sequence of tokenized sentences from an input corpus with the following snippet:

*def get\_seq\_of\_tokens(corpus):*

*tokenizer.fit\_on\_texts(corpus)*

*all\_words = len(tokenizer.word\_index) + 1*

*input\_sequences = []*

*for line in corpus:*

*token\_list = tokenizer.texts\_to\_sequences([line])[0]*

*for i in range(1, len(token\_list)):*

*n\_gram\_sequence = token\_list[:i+1]*

*input\_sequences.append(n\_gram\_sequence)*

*return input\_sequences, all\_words*

*inp\_sequences, all\_words = get\_seq\_of\_tokens(corpus)*

*inp\_sequences[:10]*

The get\_seq\_of\_tokens() function trains a Tokenizer() on the given corpus. Then you need to iterate through each line of the corpus and convert them into their tokenized equivalents. Finally, for each tokenized sentence, you create the different sequences of N-gram from it.

Next, you will see how you can deal with variable sentence length with padding.

#### **Padding Sequences**

As discussed previously, deep learning models expect fixed-length input. But with text, the length of a sentence can vary. One way to overcome this is to transform all sentences to have the same length. You will need to set the maximum length of sentences. Then, for sentences that are shorter than this threshold, you can add padding, which will add a specific token value to fill the gap. On the other hand, longer sentences will be truncated to fit this constraint. You can use pad\_sequences() to achieve this:

*from keras.preprocessing.sequence import pad\_sequences*

You can create the generate\_padded\_sequences function, which will take input\_sequences and generate the padded version of it:

*def generate\_padded\_sequences(input\_sequences):*

*max\_sequence\_len = max([len(x) for x in input\_sequences])*

*input\_sequences = np.array(pad\_sequences\*

*(input\_sequences, \*

*maxlen=max\_sequence\_len, \*

*padding='pre'))*

*predictors, label = input\_sequences[:,:-1], \*

*input\_sequences[:,-1]*

*label = ku.to\_categorical(label, num\_classes=all\_words)*

*return predictors, label, max\_sequence\_len*

*predictors, label, max\_sequence\_len = generate\_padded\_sequences\*

*(inp\_sequences)*

Now that you know how to process raw text, have a look at the modeling step in the next section.

## Back Propagation Through Time (BPTT)

There are many types of sequential models. You've already used simple RNNs, deep RNNs, and LSTMs. Let's take a look at a couple of additional models used for NLP.

Remember that you trained feed-forward models by first making a forward pass through the network that goes from input to output. This is the standard feed-forward model where the layers are densely connected. To train this kind of model, you can backpropagate the gradients through the network, taking the derivative of the loss of each weight parameter in the network. Then, you can adjust the parameters to minimize the loss.

But in RNNs, as discussed earlier, your forward pass through the network also consists of going forward in time, updating the cell state based on the input and the previous state, and generating an output, Y. At that time step, computing a loss and then finally summing these losses from the individual time steps gets your total loss.

This means that instead of backpropagating errors through a single feed-forward network at a single time step, errors are backpropagated at each individual time step, and then, finally, across all time steps—all the way from where you are currently, to the beginning of the sequence.

This is why it's called backpropagation through time. As you can see, all errors are flowing back in time to the beginning of your data sequence.

A great example of machine translation and one of the most powerful and widely used applications of RNNs in industry is Google Translate. In machine translation, you input a sequence in one language and the task is to train the RNN to output that sequence in a new language. This is done by employing a dual structure with an encoder that encodes the sentence in its original language into a state vector and a decoder. This then takes that encoded representation as input and decodes it into a new language.

There's a key problem though in this approach: all content that is fed into the encoder structure must be encoded into a single vector. This can become a huge information bottleneck in practice because you may have a large body of text that you want to translate. To get around this problem the researchers at Google developed an extension of RNN called **attention**.

Now, instead of the decoder only having access to the final encoded state, it can access the states of all the time steps in the original sentence. The weights of these vectors that connect the encoder states to the decoder are learned by the network during training. This is called attention because when the network learns, it places its attention on different parts of the input sentence.

In this way, it effectively captures a sort of memory access to the important information in that original sentence. So, with building blocks such as attention and gated cells, like LSTMs, RNNs have really taken off in recent years and are being used in the real world quite successfully.

You should have by now gotten a sense of how RNNs work and why they are so powerful for processing sequential data. You've seen why and how you can use RNNs to perform sequence modeling tasks by defining this recurrence relation. You also learned how you can train RNNs and looked at how gated cells such as LSTMs can help us model long-term dependencies.

In the following exercise, you will see how to use an LSTM model for predicting the next word of a text.

## Exercise 9.03: Building an RNN with an LSTM Layer for Natural Language Processing

In this exercise, you will use an RNN with an LSTM layer to predict the next 2 words of a given words.

The tmdb\_5000\_movies.csv dataset contains raw text that consist of movie titles. You will be training an LSTM model that will predict the next word of given sentence.

Note

You can find the dataset [here](https://raw.githubusercontent.com/fenago/tf/main/Chapter9-Recurrent%20Neural%20Networks/datasets/tmdb_5000_movies.csv)

Perform the following steps to complete this exercise:

* 1. Import the libraries needed:

*import pandas as pd*

*import numpy as np*

*import matplotlib.pyplot as plt*

*import seaborn as sns*

*import nltk*

*nltk.download('punkt')*

*from nltk.tokenize import word\_tokenize*

*from keras.preprocessing.text import Tokenizer*

*from keras.preprocessing.sequence import pad\_sequences*

*from keras.models import Sequential*

*from keras.layers import Embedding, LSTM, Dense, Dropout*

*from sklearn.preprocessing import LabelEncoder*

*import warnings*

*warnings.filterwarnings('ignore')*

*sns.set()*

* 1. Load the dataset and view the top 5 observations.

*path = "https://raw.githubusercontent.com/fenago/tf/main/Chapter9-Recurrent%20Neural%20Networks/datasets/tmdb\_5000\_movies.csv"*

*df = pd.read\_csv('datasets/tmdb\_5000\_movies.csv')*

*df.head(3)*

You should get the following output:

Table

Description automatically generated

Figure 9.3: Expected output

* 1. Subset only the ‘original\_title’ column from the data frame and assign to new data frame called df.

*df = df['original\_title']*

* 1. Convert the above data frame to the list and view top 5 movie titles.

*movie\_name = df.to\_list()*

*movie\_name[:5]*

You should get the following output:

Graphical user interface, text

Description automatically generated

* 1. Use tokenizer.fit\_on\_text to extract tokens from the corpus. Each integer output corresponds with a specific word. With texts\_to\_sequences, convert each sentence into its tokenized equivalent. Check the top 10 sequences.

*tokenizer = tf.keras.preprocessing.text.Tokenizer()*

*tokenizer.fit\_on\_texts(movie\_name)*

*seq = tokenizer.texts\_to\_sequences(movie\_name)*

*seq[:10]*

You should get the following output:

Text, letter

Description automatically generated

* 1. Drop the single word and create a feature and target list X and y. check the first 10 X and y lists.

*X = []*

*y = []*

*total\_words\_dropped = 0*

*for i in seq:*

*if len(i) > 1:*

*for index in range(1, len(i)):*

*X.append(i[:index])*

*y.append(i[index])*

*else:*

*total\_words\_dropped += 1*

*print("Total Single Words Dropped are:", total\_words\_dropped)*

*X[:10]*

*y[:10]*

You should get the following output:



Text

Description automatically generated

A picture containing text

Description automatically generated

* 1. Pad the sequence. Use pad\_sequence to pad the sequences and make their lengths equal and check the results.

*X = tf.keras.preprocessing.sequence.pad\_sequences(X)*

*X*

You should get the following output:

Table, calendar

Description automatically generated

* 1. Using tf.keras.utils.to\_categorical(), convert the y into one-hot encoded.

*y = tf.keras.utils.to\_categorical(y)*

*y*

You should get the following output:

A picture containing scatter chart

Description automatically generated

* 1. Prepare your model for training. Add an input embedding layer with model.add(Embedding). Add a hidden LSTM layer with 100 units and finally add dense layer with size of vocab\_size

*vocab\_size = len(tokenizer.word\_index) + 1*

*model = tf.keras.Sequential([*

*tf.keras.layers.Embedding(vocab\_size, 14),*

*tf.keras.layers.LSTM(100, return\_sequences=True),*

*tf.keras.layers.LSTM(100),*

*tf.keras.layers.Dense(100, activation='relu'),*

*tf.keras.layers.Dense(vocab\_size, activation='softmax'),*

*])*

* 1. Check the model summary

*model.summary()*

You should get the following output:

Table

Description automatically generated

* 1. Set model loss function, metric and optimizer using compile() method.

*model.compile(*

*optimizer=tf.keras.optimizers.Adam(learning\_rate=0.004),*

*loss='categorical\_crossentropy',*

*metrics=['accuracy'])*

* 1. Fit your model with model.fit and set it to run on 150 epochs.

*model.fit(X, y, epochs=150)*

You should get the following output:

Table

Description automatically generated

* 1. Create a vocabulary array using tokenizer.word\_index.keys() method

*vocab\_array = np.array(list(tokenizer.word\_index.keys()))*

* 1. Define a function to make prediction, which takes a word and ask for number of next words you want to predict.

*def make\_prediction(text, n\_words):*

*for i in range(n\_words):*

*text\_tokenize = tokenizer.texts\_to\_sequences([text])*

*text\_padded = tf.keras.preprocessing.sequence.pad\_sequences(text\_tokenize, maxlen=14)*

*prediction = np.squeeze(np.argmax(model.predict(text\_padded), axis=-1))*

*prediction = str(vocab\_array[prediction - 1])*

*text += " " + prediction*

*return text*

* 1. Make the predictions using the make\_prediction function.

*make\_prediction("pirates", 3)*

You should get the following output:



In this exercise, you have successfully predicted some movie reviews.

Now, that you have all the essential knowledge about RNNs, try to test yourself by performing the next activity.

## Activity 9.02: Building an RNN LSTM for Predicting Movie Sentiment

The IMDB.csv dataset contains a list of review related to movies. Each of the review is classified as having positive or negative.

You have been tasked to analyze a sample of reviews for the movies. Your goal is build an RNN model that will be able to predict the sentiment of each review: either positive or negative.

Note

You can find the IMDB.csv [here](https://drive.google.com/file/d/12fjVcxssDM1ReQUUeVAcByr62So6ff8O/view?usp=sharing)

1. Import the necessary packages.
2. Split the train and test sets
3. Prepare the text data (tokenize words and add padding).
4. Define and train an RNN model composed of two different layers of LSTM with, respectively, 50 and 100 units followed by 20% dropout and ReLU as the activation function and sigmoid in final layer.
5. Make predictions on the testing set with the trained model.

You should get the following output

Diagram

Description automatically generated with low confidence

Note

The solution to this activity can be found via [this link](https://github.com/fenago/tf/blob/main/Chapter9-Recurrent%20Neural%20Networks/Exercise%209.03-%20Building%20an%20RNN%20with%20an%20LSTM%20Layer%20for%20Natural%20Language%20Processing.ipynb).

## Summary

In this chapter, you explored different recurrent models for sequential data. You learned that each sequential data point is dependent on the prior sequence of data points, such as natural language text. You also learned why you must use models that allow for the sequence of data to be used by the model, and sequentially generate the next output.

This chapter introduced RNN models that can make predictions for sequential data. You observed the way RNNs can loop back on themselves, which allows the output of the model to feed back into the input. You reviewed the types of challenges that you face with these models, such as vanishing and exploding gradients, and how to address them.

In the next chapter, you will learn how to utilize custom TensorFlow components to use within your models, including loss functions and layers.