Table of Contents

Getting started with RNNs

As the field of artificial intelligence grows, a number of new applications are built for natural language processing and sequence modeling. Recurrent neural networks, or RNNs in short, form the foundation for such applications that deal with sequences of data. This course helps students to get started with the concepts and applications of RNNs. My name is Kumaran Ponnambalam, in this course I will start off by introducing the architecture of RNNs. Then, I will discuss the training process of RNNs through forward propagation and gradient descent. I will also explore advanced RNN architectures, like GRUs and LSTMs. Finally, I will look at word embeddings and how it helps in building text based models. Let's now start learning the concepts around RNNs.

A review of deep learning

Let's begin this course with a review of deep learning, a rapidly-growing field in the world of artificial intelligence. Deep learning is a subset of machine learning, used to build models for a variety of complex use cases. Neural network architectures that have three or more layers usually qualify as deep learning networks. They imitate how humans process incoming data, create knowledge, and use that knowledge to make decisions. This is a field that has seen exponential growth in the last few years. While the concepts and experiments of deep learning have existed for a long time, recent advances in large-scale data processing and high-performance CPUs has spurred its wide adoption. Deep learning has evolved with different types of

architectures, based on the use cases. Convolutional neural networks, or CNNs, are popular in image recognition. Recurrent neural networks, or RNNs, are used for creating models for sequential use cases. It has multiple variants, like gated recurrent units and long short-term memory. We will study RNNs and its variants in this course. Other than these two popular ones, there are also self-organizing maps and auto encoders. What are the applications of deep learning? The types of applications have grown multifold in the last few years. Deep learning is used for natural language processing, to extract information out of text data. It is used for understanding human speech, as well as synthesizing speech from text. Image recognition applications are growing for security and logistics use cases. Self-driving cars is a new emerging technology that extensively uses deep learning for various driving tasks. In addition, there are a number of applications being built in the customer experience, healthcare, and robotics domains. Now let's start exploring more about RNNs and sequence models.

Why sequence models?

RNNs are also called sequence models, as they are primarily used to model patterns in sequences of data. Why do we need a special type of neural network for sequences? Deep learning models are used to take a set of inputs, run through the model, and predict outputs. They do not consider time as a factor. The inputs happen at a point in time and the outputs are generated based on these inputs at that time. These models are not dependent upon what happened in the past, or what will happen in the future. In short, they do not model relationships across time. Let's take a simple example. We want to predict the word between won and match. There is no single answer to this problem. Why? If the word before won is he, then the right prediction would be the word his. This is because we need to consider the gender, which is indicated by the word he, that occurred two words before in the sequence. Similarly, if the word before won is she, the right prediction would be her. If the word before is they, it indicates a plural, and hence

the right prediction would be their. As seen here, the prediction would depend upon what happened prior in this specific sequence. Sequence models predict the future values in a given sequence based on the past patterns in the sequence. Sequence models have the ability to capture information about the past and store it in memory. It will then use this memory to predict future occurrences. Sequence models can predict multiple future values, if needed. There are also bidirectional models that can predict prior values in the sequence based on the values that happened after. We will review the architecture of recurrent neural networks in the next video.

A recurrent neural network

What does the architecture of an RNN look like? Let's take a look at a simple recurrent neural network now. We have a time sequence with four time steps, namely T one, T two, T three, and T four. These time steps need not be equally spaced out. Rather they could be logically separated, like words in a sentence. At each time step, a corresponding input is available. Let's say they are X one, X two, X three, and X four. This input is then send through an RNN. The RNN is the same network that is used in each of these time steps, but the input and output would be different based on the time step. At each time step, the RNN may produce an output. In this case, they are Y one, Y two, Y three, and Y four. This function is similar to a basic neural network. However, in addition to the output, the RNN will also produce a hidden state that is then passed on as input to the next step. This hidden state is called memory that passes on information about the previous time steps to the next time steps. The RNN itself is a multi-layered network with its own nodes, weights, biases, inputs, and outputs. To summarize, the same neural network is used in each step. This network has layers, nodes, weights, biases, and activation functions. The network can be as simple as a one-layer network or a deep network with tens of layers. It has two inputs, namely the input coming from the current time step and the hidden state that is got from the previous time step. It again has two outputs, namely the

output for the current time step and the hidden state to the next time step. Let's take an example of a translation use case where the input is in English and the output is in Spanish. If we pass an input string, this is an example. Then each word becomes a time step. On passing through the RNN, it produces the output in Spanish. This use case is doing a one-to-one translation, but real-life use cases may produce zero to multiple words for each input. There are different types of RNNs based on the input and output mappings. We will discuss them in the next video.

Types of RNNs

There are multiple types of RNN architectures that support various use cases. Let's review these types and their use cases. Let's start with a many-to-many RNN where each time step will have inputs and outputs. A good example for this is speech recognition where as each word is spoken, the output would be the text representation of the word. Then we have many-to-one RNNs. Here, there is an input at each time step, but there are no corresponding outputs. An output is only produced after all the inputs are provided in the final time step. An example is predicting the price of stock for the next day given its historical prices. Also classification problems, like sentiment analysis, will take a sequence of text as input and produce a single output at the end. Then comes one-to-many RNNs. These types of architectures are rare. An example would be music synthesis where a single input is provided to the model and the model generates a sequence of musical notes. We then have encoder-decoder RNNs which are popular in transformers. In this case, a set of inputs are provided first going through the encoding phase, then a set of outputs are generated out of the decoder. The number of outputs may not match the number of inputs. One example is machine translation where an input sequence in a specific language is translated to another language. During translation, there is no word-for-word translation, rather the semantics is passed on to the new language. Similarly, text summarization is another example where an input is summarized to a

smaller word count. Finally, what about one-to-one RNNs? With no hidden state available, this is just a classical neural network. Hidden states are what makes RNNs special.

Applications of RNNs

What are some of the popular real world applications for RNNs or sequence models? Time series forecasting is used to predict future values in a time series based on the past. Use cases like stock price predictions and volume forecasting falls into this category. Text classification is used to classify text input and derive semantics like intent and sentiment. Topic modeling is used to cluster text data and generate common topics across them. Sentiment analysis is a popular classification use case for RNNs. Named entity recognition is used to extract contact-specific data, like names, phone numbers and currencies from a body of text. Speech-to-text is used to convert human speech into equivalent text. Text-to-speech is used to take a body of text and generate corresponding human speech. Machine translation is used to translate text from one language to another. Question answering use case is used to answer specific questions from a query by extracting the answers from a body of text. Text summarization can summarize a large body of text into one of smaller word counts. As we can see, RNN applications are popular around text use cases, but they're equally good for sequential numeric data.

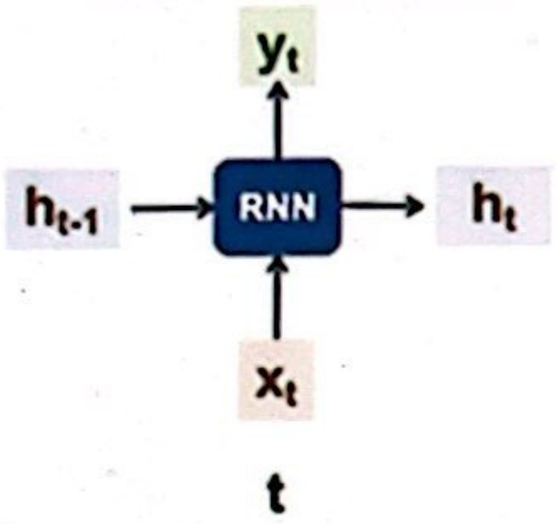
Training RNN models

How is the training process of RNNs different from training regular neural networks? Training RNNs have a number of similarities with the base training process. Training data will undergo preparation and pre-processing before they are used for model training. Weights and biases will be initialized. There is a similar forward propagation step in RNNs, too. After forward propagation is completed we will compute prediction loss and cost. We will then use this value to perform back-propagation and update the weights and biases. Gradient descent happens until the observed accuracy improves to expected levels. What

is unique about RNN training? We train by time steps and each time step may produce an output depending upon the type of architecture. We will, in addition, compute the hidden state, and propagate this to future time steps. This also plays a role in loss computation and back-propagation. Finally, when there are multiple outputs, we compute cumulative error and perform back-propagation.

Forward propagation with RNN

Let's dive deeper into the forward propagation process for RNNs. Similar to a regular RNN, we will use multiple samples to train an RNN model. Each sample is a sequence that has multiple time steps. Even though there are multiple time steps, it is essentially the same model that is used for each time step. The model contains layers, nodes, weights, biases and activation functions, similar to regular neural networks. In addition, a hidden state is also computed and used for model training. The hidden state from the previous step is used to compute the output of the current state as well as the next hidden state. Let's review the formula used for these computations. At each time step, T, we have an input, XT, that goes through a single RNN network. In addition, there is a hidden state, HT minus one, that comes from the previous time step. We produce HT, the next hidden state, as well as YT, the current output. For the first time step, the previous hidden state would be initialized to zeroes.



$$h_t = f(w_h h_{t-1} + w_x x_t + b_h)$$

$$y_t = f(w_y h_t + b_y)$$

y	Dependent Variable	b	Bias
x	Independent Variable	h	Hidden State
w	Weight	f	Activation Function

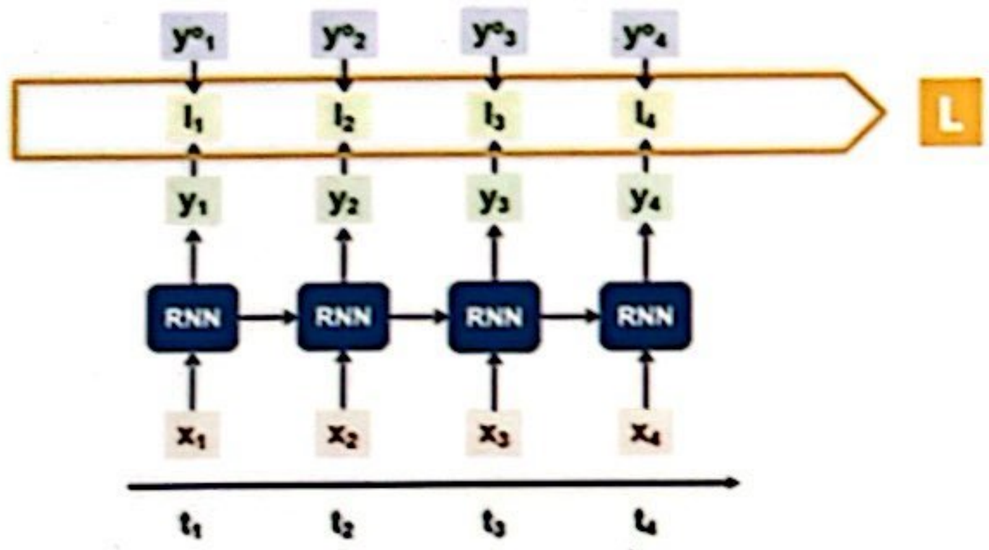
To compute HT, we will use two sets of weights, WH and WX. We use one bias, BH. We first use the formula HT equals to an activation function of WH, multiplied by HT minus one plus WH, multiplied by XT plus BH. For YT, we will use another, WY, and bias, BY. The hidden state, HT, is then used to compute YT. HT already uses both the input, XT, and the previous state, HT minus one. So both inputs influence the computation of YT. Unlike regular neural networks, which uses one sets of weights and biasses, we have three sets of weights and two sets of biasses. Their values are initialized randomly and adjusted during the gradient descent process.

Computing RNN loss

Once we complete forward propagation with an RNN, we need to compute the loss value for the prediction. For computing loss, we use similar loss functions like regular neural networks. We will compute the loss for each time step and then aggregate the loss across time steps to compute overall loss for the given samples. We then compute the cost across a batch for multiple samples and use it for back propagation. As an example, we have an RNN with four time steps with four inputs and four outputs. For each of the outputs, we also have a corresponding actual target value that is available during training. We then compute the difference between the predicted and actual values

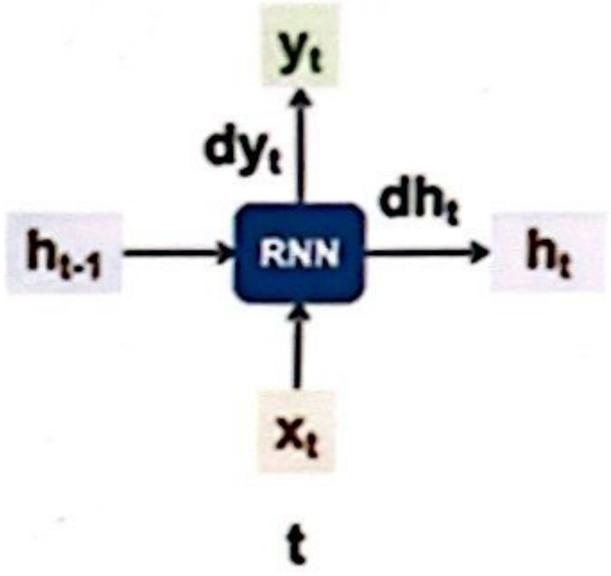
of each time step to find individual loss. We then aggregate the loss values across time steps to get overall loss for the samples. Loss values across samples in a batch are then used to compute the cost.

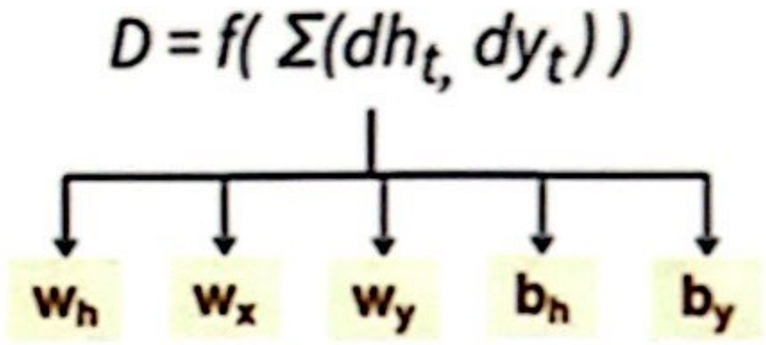
RNN Loss



Backward propagation with RNN

How does back propagation work in the case of RNNs? In the previous video we discussed how to compute the cost function for a batch. We will then use this cost function to compute derivatives or delta for both the hidden state and the output for each step. The derivative represents the adjustments that needs to be made to various weights and biases. We then compute cumulative delta for all the time steps. We then apply this cumulative delta to all the weights in biases for the RNN model. We will continue the gradient descent process until the cost value reaches acceptable thresholds. Let's now walk through this process.





We take the network at time step T. We first compute the derivative or delta DHT for the output HT based on the cost computed. We will similarly compute the delta for the output DYT. For sake of simplicity we are not discussing the actual formula in this video. Then we will compute the cumulative delta as a function of all the deltas of individual time steps. We then apply this derivative to adjust the weights and biases for the RNN model. Once the weights and biases are adjusted, we will continue with gradient descent for model training.

Predictions with RNN

Predictions with RNN are pretty straightforward and similar to regular neural networks. The inputs to the RNN during prediction will vary depending upon the type of the model. It may be the list of previous time steps, in the case of time series predictions, or a sequence of text, like in the case of natural language processing. The output may just be the next time step or another sequence of text. Whatever be the case, the input data will be prepared using the same steps used to prepare the training data. Then the input data is fed into the RNN model and predictions are obtained.

A simple RNN example: Predicting stock prices

Having learned about RNN basics, let's build a simple RNN model in this chapter to predict stock prices. We will use Keras, which will do

most of the heavy lifting needed for building the RNN. The code for this chapter is available in the notebook, Code-03_XX Predicting Stock Prices. We first open up this notebook. There are multiple dependent packages that are required for the exercises in this course. This is listed in Section 3.01. Please make sure to run this code and ensure that all dependencies exist before executing any exercises in this course. In this exercise, we will predict the future stock price for a company based on the historical stock prices. In this data file, FB-stock-prices.csv, we have the stock prices for Facebook for a period of one year. It has two columns, namely date and price. We will use this data to predict the stock price for the next day, using a simple RNN.

Data preprocessing for RNN

Let's now start loading up the stock data from the CSV file. The code for this is available in section 3.2 of the notebook.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

np.random.seed(1)

#Load the Meta stock price data from CSV
prices = pd.read_csv('FB-stock-prices.csv')

#Review loaded data
print(prices.dtypes)
prices.head()
```

First, we load up the stock prices into the prices, data frame. We then print the data types and the top files to check if the data has been loaded properly. Then, we will visualize the stock data. We plot the date points on the X axis and prices on the Y axis. This gives us a visualization of the price trends for the Facebook stock. Next, we will scale this data using the standard scaler. Do note that given that there is only one feature variable here, scaling is optional. Then, we split the data into training and test data sets. We will keep the last 50 days of data in the test data set and the remaining in the training data set.

For sequence model, we should not be splitting data randomly but we should split it sequentially.

Preparing time series data with lookback

For preparing time series data for RNN, some special steps need to be followed. When it comes to time series, we have stock prices available as a single sequence for a given stock symbol. We will usually get only one such sequence as training data, and we need to build a model to predict for the next day based on the previous days. The features would be a list of stock values for the immediate previous time steps. How can we create multiple such sequences from a single input sequence? It can be done in the following way. We first establish a lookback value which is the number of previous time steps that will be used as training data. Then we can create multiple samples from a single sequence as follows. Let's say the lookback is five. We then create the first sample by taking the sixth value as a target value shown here as T, and the first five as feature values shown here as F. Then the second sample will be created by sliding one step ahead. Now the seventh value becomes the target, and the values from the second to the sixth time step became the feature variables. By following this process, we can create multiple samples. So if there are a total of X values in the input sequence, and if Y is the lookback, we should be able to create X minus Y minus one samples from it. The logic we just discussed is implemented in section 03.03 of the notebook.

```
#Prepare RNN Dataset.
#Each data point (X) is linked to the previous data points of size=lookback
#The predicted value (Y) is the next point

def create_rnn_dataset(data, lookback=1):

    data_x, data_y = [], []
    for i in range(len(data)- lookback -1):
            #All points from this point, looking backwards upto lookback
            a = data[i:(i+ lookback), 0]
            data_x.append(a)
            #The next point
            data_y.append(data[i + lookback, 0])
    return np.array(data_x), np.array(data_y)

#lookback for 25 previous days
lookback=25
```

```
#Create X and Y for training
train_reg_x, train_reg_y = create_rnn_dataset(training_prices,lookback)

#Reshape for use with RNN
train_reg_x = np.reshape(train_reg_x,
                         (train_reg_x.shape[0],1, train_reg_x.shape[1]))

print("Shapes of X, Y: ",train_reg_x.shape, train_reg_y.shape)
```

We create a function called create_rnn_dataset that would take as input, the input sequence and the lookback value. It will then create multiple samples with a target and multiple features following the same logic we discussed. Here we set the lookback to 25 and create the data set. We will have to reshape the output to make the input requirements for RNN. There were originally 201 stock values in the training input that has now given 175 samples with a lookback of 25.

Creating an RNN model

- Having prepared the training data in the previous video, let's proceed to create an RNN model and train it. The code for this is available in section 3.4.

```
from keras.models import Sequential
from keras.layers import SimpleRNN,Dense
import tensorflow as tf

tf.random.set_seed(3)

#Create a Keras Model
price_model=Sequential()
#Add Simple RNN layer with 32 nodes
price_model.add(SimpleRNN(32, input_shape=(1,lookback)))
#Add a Dense layer at the end for output
price_model.add(Dense(1))

#Compile with Adam Optimizer. Optimize for minimum mean square error
price_model.compile(loss="mean_squared_error",
                    optimizer="adam",
                    metrics=["mse"])

#Print model summary
price_model.summary()

#Train the model
price_model.fit(train_reg_x, train_reg_y,
```

epochs=5, batch_size=1, verbose=1)

We first initialize tensor flow to a fixed seed. So, we get repeatable results with this notebook. Then, we create a sequential model in Keras. We add a simple RNN layer to this model. The layer is available out of the box in Keras. We pass the total number of notes in the layer as 32 and provide the input dimensions. We finally add a dense layer that would output a predicted stock price. We then compile this model using "mean_squared_error" as the loss function and "adam", as the optimizer. We print the model summary. Then, we train the model using the fit method. We've passed the input feature samples and corresponding target variables to this fit method. We also set epochs to five and batch size to one. On executing the model, the results are printed. We get a loss of 0.04 at the end. Do note that except for the way we create the training data set and adding the simple RNN layer, the remaining steps are similar to how we build a regular neural network model. Keras masks the complexities around hidden state processing, and makes it easy for us to build RNN models.

Testing and predictions with RNN

Having built the stock price model with simple RNN, let's proceed to test the model.

```
#Preprocess the test dataset, the same way training set is processed
test_req_x, test_req_y = create_rnn_dataset(test_prices,lookback)

print(test_req_x.shape, test_req_y.shape)

test_req_x = np.reshape(test_req_x,
                        (test_req_x.shape[0],1, test_req_x.shape[1]))

#Evaluate the model
price_model.evaluate(test_req_x, test_req_y, verbose=1)

#Predict on the test dataset
predict_on_test = price_model.predict(test_req_x)

#Inverse the scaling to view results
predict_on_test = scaler.inverse_transform(predict_on_test)
```

```
#Extract original test values that map to the same dates on the predicted
test values
orig_test_values = prices["Price"].loc[total_size-len(predict_on_test):]
orig_test_values.reset_index(drop=True, inplace=True)

plt.figure(figsize=(20,10)).suptitle("Plot Predictions for Original Test
Data", fontsize=20)
plt.plot(orig_test_values, label="Original Price")
plt.plot(predict_on_test, label="Predicted Price")
plt.legend()
plt.show()
```

For predictions, the feature variables need to go through the same set of pre-processing and preparation as was done for training. For the test data, we have already scaled it. Now we will create the same RNN data set as we did for the training data. Then we reshape it. We now end up with 24 samples in the test data. We will then use the model to predict for each of them. We will then use the scalar function to do an inverse transform of the return values and get them back to the original scale. After this, we have 24 predicted values for the test data. We will now plot it against the actual stock prices for the same days and compare the results. Comparing the results, we see that the predictions follow similar trends as the original data. Accuracy can be improved by using more advanced techniques like LSTM. Now let's move on to predictions. To predict the stock price for a given day, we need the stock prices for the 25 previous days as look back. We have this input available in an array. We will follow the same processing that was done for training data. We will scale and reshape. Then we run it through the model and get the predictions. We will finally scale back to the original scale. The predicted stock price is shown here. This completes our exercise with a simple RNN.

```
#Previous prices
previous_prices = np.array([325,335,340,341,342,
                            325,329,316,312,317,
                            324,330,328,332,336,
                            341,339,335,328,328,
                            341,348,343,341,339])

#Scale
scaled_prices = scaler.transform(previous_prices.reshape(-1, 1))
#Reshape
model_input = np.reshape(scaled_prices,
```

```
                                (scaled_prices.shape[1],1, scaled_prices.shape[0]))

#predictions
prediction = price_model.predict(model_input)

#scale back
norm_prediction = scaler.inverse_transform(prediction)

print("The predicted Stock price is :", norm_prediction[0][0])
```

Predictions with RNN

Predictions with RNN are pretty straightforward and similar to regular neural networks. The inputs to the RNN during prediction will vary depending upon the type of the model. It may be the list of previous time steps, in the case of time series predictions, or a sequence of text, like in the case of natural language processing. The output may just be the next time step or another sequence of text. Whatever be the case, the input data will be prepared using the same steps used to prepare the training data. Then the input data is fed into the RNN model and predictions are obtained. Having reviewed the training and prediction processes for RNNs, we will implement an example in Keras in the next chapter.

RNN Architecture

The vanishing gradient problem

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] In this chapter, we will explode the problem of vanishing gradients and then discuss a few architectures in RNN that helps alleviate this problem. Let's take an example of an RNN that takes as input the last four words in a sequence and predicts the next word. So in this case, the inputs are X1 to X4, and there is a single output, Y, at the end. For example, if we provide the input as they wanted more for, we want the model to predict the word, themselves, but there is a

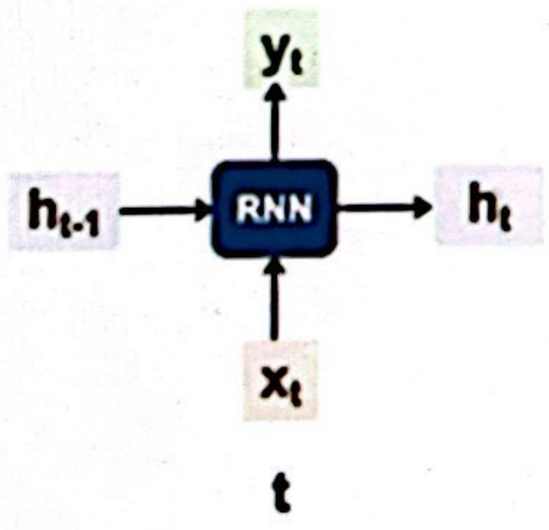
problem here. Consider a similar input sequence, he wanted more for. In this case, the prediction should be himself. Between these two sequences, the second, third and fourth words are the same, and the output is dependent on the first word. In other words, the output Y at T4 is dependent on the input at T1. So there is a long running dependency that needs to be properly captured and propagated through these hidden states. In a regular RNN, the output at any given time step is influenced heavily by the input at that time step. As the hidden state propagates through the network, the weights of the inputs decay over the time step. For example, at T2, the hidden state generated would be based on X1 and X2, with X1 having a higher weight. When we get to T3, X3 will have a higher weight than X2, and X2 will have a higher weight than X1. The weight of X1 decays and looses significance. This is okay in most sequence models, where the next value is heavily influenced by the nearest previous values but in some cases, we need to persist long running dependencies. We will explore two alternate architectures that help solve the problem in the next few videos.

The gated recurrent unit

Gated recurrent units or GRUs is an alternate sequence model architecture that helps prolong the memory of certain time steps and take care of long running dependencies. In a regular RNN, the hidden state from the previous layer is aggregated with inputs in the next layer to compute the new hidden state. As we progress through the time steps, the weight of a given input in the hidden state will decay. A GRU computes an additional update gate at each time step. The update gate produces a result of zero or one. If the value is zero, then the current input is ignored and the previous hidden state becomes the current hidden state. If the value is one, then the previous hidden state is fully ignored and a new hidden state is computed entirely based on the input at that time step. There are some implementations of GRU that will use an additional reset gate also. The gates ensure propagation of the selected states across the network without decay

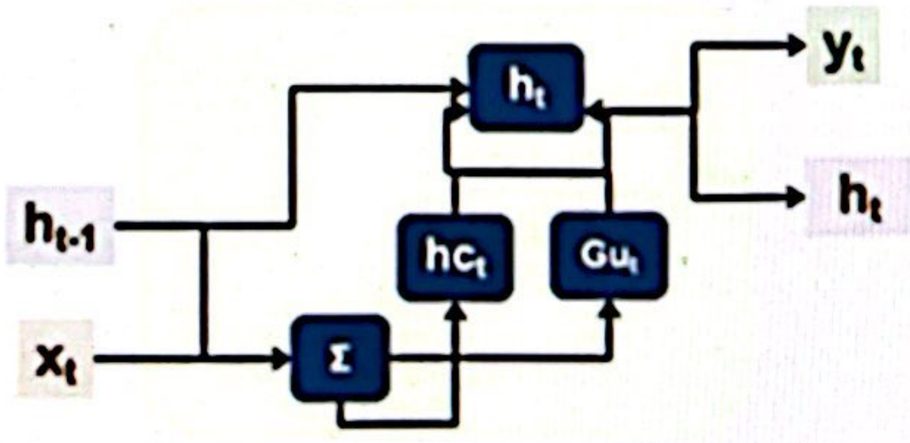
and help preserve certain features needed for predictions down the line. Let's take a look at the computation now. We have the original RNN formula for hidden state and output shown here on the left. In the case of a GRU, we still have the same two inputs, Xt and Ht-1. We have the same two outputs, Ht and Yt. We first compute a candidate hidden state for this time step using the same formula for hidden state as the regular RNN. We then compute an update gate based on the previous hidden state and the current input. We use a separate set of weights and biases for computing the update gate. We finally use an activation function that will return a one or zero. Now we compute the hidden state for this time step using the formula shown here. In this formula, if the update gate GUt is one, then it returns only the candidate hidden state and completely ignores the previous hidden state. If the value is zero, then it ignores the candidate hidden state and returns the same value as the previous hidden state. So the previous hidden state is entirely consumed or ignored. Now we can proceed to compute the output using the same formula as the RNN. Using the same example as the previous video, the model using GRU would probably do the following. It'll set the update gate to one at T1. Then it'll set the update gate for T2, T3, and T4 to zero. Thus, the hidden state created at T1 will propagate without decay to T4 and fully influence the prediction Y.

Computing Hidden State with GRU



$$h_t = f(w_h h_{t-1} + w_x x_t + b_h)$$

$$y_t = f(w_y h_t + b_y)$$



$$hc_t = f(w_h h_{t-1} + w_x x_t + b_h)$$

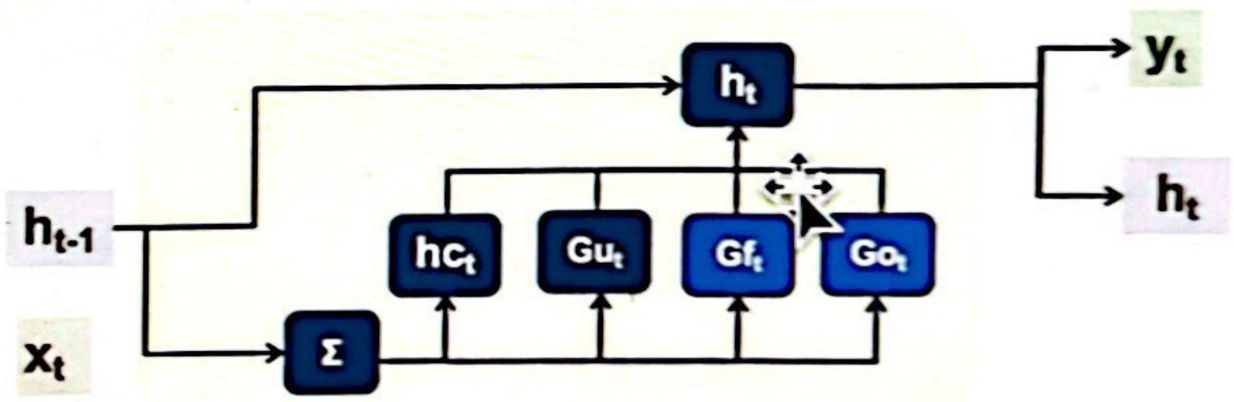
$$Gu_t = f(w_U [h_{t-1}, x_t] + b_U)$$

$$h_t = Gu_t\, hc_t + (1 - Gu_t)\, h_{t-1}$$

Another popular architecture in RNNs that addresses the vanishing gradient problem is long short-term memory or LSTM. As discussed in the previous videos, the weight of an (indistinct) within the hidden state as we move across time steps. LSTM uses multiple gates to decide if the previous hidden state needs to be passed on or ignored. Unlike GRU, it's possible to use a previous hidden state as well as the current import to compute the next hidden state. It ensures long term memory of certain features that are from the previous time steps. When do we choose GRU and when do we choose LSTM? LSTM is known to be better at predicting longer sequences but will have computational overhead. GRU is more efficient than LSTM but can handle reasonably sized sequences well.

Let's look at the various gates computed for LSTM.

Computing Hidden State with LSTM



$$hc_t = f(w_h h_{t-1} + w_x x_t + b_h)$$

$$Gu_t = f(w_u [h_{t-1}, x_t] + b_u)$$

$$Gf_t = f(w_f [h_{t-1}, x_t] + b_f)$$

$$Go_t = f(w_o [h_{t-1}, x_t] + b_o)$$

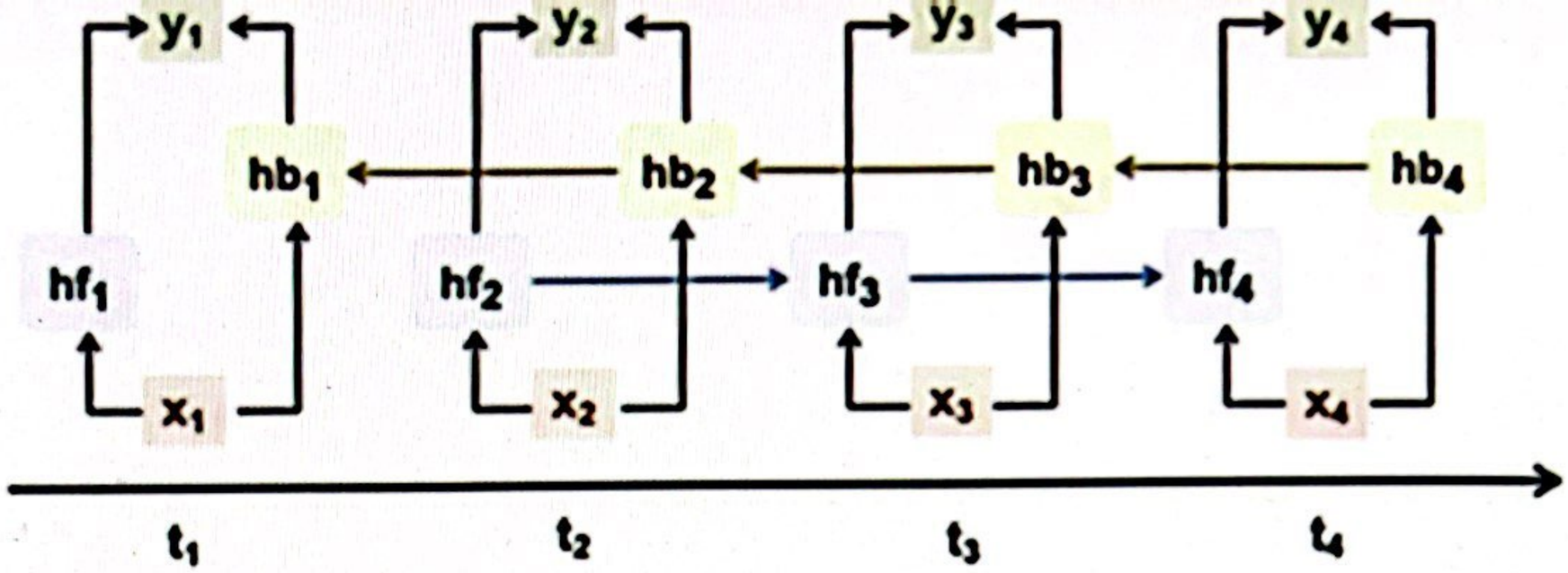
$$h_t = Go_t (Gu_t hc_t + Gf_t h_{t-1})$$

We start off with the constructs and formulate as GRU. We will compute the candidate hidden state and an update gate is LSTM also with the same formula. The activation functions may be different. In

addition, we will compute a (indistinct) gate. Its formula is similar to the update gate except that it has its own weight and biases. We also compute another gate called the output gate, again with its own weights and biases. Finally, we compute the hidden state throughout each time step using all these gates. The update gate is used to (indistinct) the candidate hidden state. The (indistinct) gate is used to filter the previous hidden state. The output gate is then used to (indistinct) a certain output to produce the final hidden state.

Bidirectional RNNs

Consider the problem of predicting the missing word in a sentence. In this case, we have sequences, he prepared and application, and we need to find the word in between which is an. Let's look at another example where the previous sequence is he prepared and the next word is meal. And we need to predict the missing word which is a. As we can see, the previous sequence is the same and the predicted word is dependent upon the future words or sequences. Sometimes, it's dependent on both the previous and the next sequences. The architectures we have seen so far only predict based on the past sequences. So as we see, a time step in a sequence is dependent upon both the previous time steps and the future time steps. - We need to be able to model both of them to build an RNN. This is solved by bidirectional RNNs which pass hidden states in both directions. A popular use case for bidirectional RNN is named entity recognition where we try to identify an entity based on the past and future sequences. Similarly, in speech recognition, we will use past and future utterances to predict the current utterance. Grammar checkers also use a similar model to predict the right word to use in a given context. Here is how a BRNN works.



We again have multiple time steps, T1 to T4. We have inputs X1 to X4. Then there are outputs Y1 to Y4 similar to regular RNNs. We first compute a forward hidden state HF1. The forward hidden state is the same as what we have seen so far in other unidirectional models. Now, we will also compute a backward hidden state in addition to the forward hidden state. This is again computed at each time step. The forward hidden state is then passed forward in the time steps. The hidden state HF1 is used to compute the next hidden state HF2 and so on. The backward hidden state is passed backwards. So we start from HB4 and then pass it on to the previous time step to compute HB3 and so on. Then we compute Y at each time step using both the forward and backward hidden states. This ensures that the features that are before and after the current time step are taken into account for predicting the output at each time step. In the next chapter, we will implement an example for LSTM.

Forecasting service loads with LSTM

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] In this chapter we will use LSTM to predict future service loads for a week. The exercise file for this chapter is code_05_XX

Forecasting Service Loads. The data file for this exercise is request_every_hour.csv. Consider that we have a web service that is running in a network and serving requests from its clients. The number of requests that this service handles will vary based on the time of the day and day of the week. This file contains the total service request that this service receives every hour. It has such values for a period of five weeks with each week beginning on a Monday. There are 840 values in this file one each for every hour during these five weeks. We need to use this data and predict the service load for every hour in the sixth week. In other words, we need to predict an entire week by hour which is 168 predictions. Let's first start by loading up this data file in a data frame. The code for this is available in section 5.1 of the notebook. We load the file and then display some data to check if the loading is proper. Let's then discuss time series patterns in the next video.

Sample code

```
!pip install keras
!pip install tensorflow
!pip install pandas
!pip install matplotlib
!pip install sklearn
```

Time series patterns

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] When we have a time series, there are three types of patterns that can occur in this data. We may have cyclic patterns that repeat at periodic intervals. There are seasonal patterns that happen during special occasions. Then there are growth patterns where there is a permanent change in the trends. Let's explore these with the data we have. In Section 05.02, we will do three different plots.

```
#daily

plt.figure(figsize=(20,5)).suptitle("Daily", fontsize=20)
plt.plot(requests.head(24))
plt.show()

#weekly
plt.figure(figsize=(20,5)).suptitle("Weekly", fontsize=20)
plt.plot(requests.head(168))
plt.show()

#all
plt.figure(figsize=(20,10)).suptitle("Overall", fontsize=20)
plt.plot(requests)
plt.show()
```

We will first do a daily plot with just 24 records. We will then do a weekly plot for all days in a single week with 168 records. Then, we will do an overall plot for all the five weeks. When we look at the daily plot, we see that it looks like a bell curve. The volumes are very less during the morning but steadily increases during the day and hits a peak during noon. Then the volume starts decreasing in the afternoon and reaches a low point around midnight. Let's look at the weekly pattern for seven days from Monday to Sunday. We see that the bell curve repeats every day with similar highs and lows. This is called a cyclic pattern. We also see that for the last day, the overall volumes are less than the other days since it is a Sunday. This can be considered as a seasonal pattern. When we look at the overall pattern, the cyclic pattern continues. But we also see that from the third week, we see an increase in overall volumes. This is called a growth patterns. A growth pattern can be negative also. When we try to build a sequence model from a time series, it should be able to capture and predict all the three patterns namely cyclic, seasonal and growth. We will discuss how to achieve this in the next videos of the chapter.

Preparing time series data for LSTM

Selecting transcript lines in this section will navigate to timestamp in the video

building models. This is in Section 5.03 of the notebook.

```
from sklearn.preprocessing import StandardScaler

#Scale the data
print("Request Range before scaling : " ,
          min(requests.Requests),
          max(requests.Requests))

scaler = StandardScaler()
scaled_requests=scaler.fit_transform(requests)
print("Request Range after scaling : " ,
          min(scaled_requests),
          max(scaled_requests))

#Training data has to be sequencial - first 4 weeks
train_size = 24 * 7 * 4

#Number of samples to lookback for each sample
lookback=24 * 7

#Separate training and test data
train_requests = scaled_requests[0:train_size,:]

#Add an additional week for lookback.
test_requests = scaled_requests[train_size-lookback:,:]

print("\n Shaped of Train, Test : ",
      train_requests.shape, test_requests.shape)
```

```
Request Range before scaling :  1 488
Request Range after scaling :  [-2.28221282] [2.23748868]

 Shaped of Train, Test :  (672, 1) (336, 1)
```

```
#Prepare RNN Dataset.
#Each data point (X) is linked to the previous data points of size=lookback
#The predicted value (Y) is the next point

def create_rnn_dataset(data, lookback=1):

    data_x, data_y = [], []
    for i in range(len(data)- lookback -1):
            #All points from this point, looking backwards upto lookback
            a = data[i:(i+ lookback), 0]
            data_x.append(a)
            #The next point
            data_y.append(data[i + lookback, 0])
    return np.array(data_x), np.array(data_y)
```

```
#Create X and Y for training
train_reg_x, train_reg_y = create_rnn_dataset(train_requests,lookback)

#Reshape for use with LSTM
train_reg_x = np.reshape(train_reg_x,
                         (train_reg_x.shape[0],1, train_reg_x.shape[1]))

print("Shapes of X, Y: ",train_reg_x.shape, train_reg_y.shape)
```

```
Shapes of X, Y:  (503, 1, 168) (503,)
```

The steps followed here are similar to the steps we followed earlier for predicting stock prices. We first start with scaling the data using a standard scaler. We will use the first four weeks of data as training data, and the last week as test data. Note that for sequential data, we cannot use random splitting like we do for regular machine learning. Next, we will set the look-back for an entire week. We need to do this so that it can capture various cyclic and seasonal trends in data. We then proceed to separate the training and test data. As the look-back interval and the test data set size is the same, we will only get one test sample. So to create test data for a week, we will also add additional data from the fourth week for look-back purposes. Now we proceed to create an RNN data set with look-backs, similar to how we did in the stock prediction example. We will finally reshape the output to match the requirements for Keras.

Creating an LSTM model

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Having prepared the time series data for service loads, we will now proceed to build the model in section 5.4.

```
from keras.models import Sequential
from keras.layers import LSTM,Dense
import tensorflow as tf
```

```
tf.random.set_seed(3)

#Create a Keras Model
ts_model=Sequential()
#Add LSTM
ts_model.add(LSTM(256, input_shape=(1,lookback)))
ts_model.add(Dense(1))

#Compile with Adam Optimizer. Optimize for minimum mean square error
ts_model.compile(loss="mean_squared_error",
                 optimizer="adam",
                 metrics=["mse"])

#Print model summary
ts_model.summary()

#Train the model
ts_model.fit(train_reg_x, train_reg_y,
             epochs=5, batch_size=1, verbose=1)
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 lstm_1 (LSTM)                (None, 256)               435200

 dense_1 (Dense)              (None, 1)                 257

=================================================================
Total params: 435,457
Trainable params: 435,457
Non-trainable params: 0
_________________________________________________________________
Epoch 1/5
503/503 [==============================] - 2s 2ms/step - loss: 0.1530 - mse:
0.1530
Epoch 2/5
503/503 [==============================] - 1s 2ms/step - loss: 0.0762 - mse:
0.0762
Epoch 3/5
503/503 [==============================] - 1s 2ms/step - loss: 0.0666 - mse:
0.0666
Epoch 4/5
503/503 [==============================] - 1s 3ms/step - loss: 0.0493 - mse:
0.0493
Epoch 5/5
503/503 [==============================] - 1s 2ms/step - loss: 0.0527 - mse:
0.0527
<keras.callbacks.History at 0x7f77d9737f10>
```

This is similar to the steps we did in stock price prediction. The only difference is that we will use an LSTM layer instead of a simple RNN layer. We start here by building a sequential model. We add an LSTM layer to this model. Finally, we add a dense layer for output. We then compile the model with a loss function, optimizer and metrics. We then print the model summary and train the model. We end up with the loss of around 0.05.

Testing the LSTM model

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Now we can proceed to test the LSTM model for predicting service loads. For this, the test data will need to go through the same processing as the training data. We will first evaluate the model against the test data and check for the loss. Then we proceed to use this model and predict for both the training dataset and the test dataset. In other words, we will use the training features and predict for each hour in the five weeks where we know the actual value. We first predict on the training data and then we predict on the test dataset. Now we will proceed to plot these in a graph and check how well the predicted values agree with the actual values. All three plots should be of the same size. So we will pad the training and test data with empty values to match the size for five weeks. We first plot the original data. Then the predictions on training data. And finally, the predictions on the test data. Let's examine the output now. The original values are shown in blue. The predictions on training data are shown in orange. The predictions on test data are shown in green. From the plots, we see that these predictions line up well with the actuals. More importantly, it captures the cyclic and seasonal trends that we saw in the original data. Now we will proceed to predict for an entire new week in the next video.

```
test_req_X, test_req_Y = create_rnn_dataset(test_requests,lookback)
test_req_X = np.reshape(test_req_X,
                        (test_req_X.shape[0],1, test_req_X.shape[1]))

#Evaluate the model
ts_model.evaluate(test_req_X, test_req_Y, verbose=1)

#Predict for the training dataset
predict_on_train= ts_model.predict(train_req_X)
#Predict on the test dataset
predict_on_test = ts_model.predict(test_req_X)

#Inverse the scaling to view results
predict_on_train = scaler.inverse_transform(predict_on_train)
predict_on_test = scaler.inverse_transform(predict_on_test)
```

```
6/6 [==============================] - 0s 2ms/step - loss: 0.1552 - mse: 0.15
52
```

```
#Plot the predictions

#Total size of plot
total_size = len(predict_on_train) + len(predict_on_test)

#Plot original data
orig_data=requests.Requests.to_numpy()
orig_data=orig_data.reshape(len(orig_data),1)

#Create a plot for original data
orig_plot = np.empty((total_size,1))
orig_plot[:, :] = np.nan
orig_plot[0:total_size, :] = orig_data[lookback:-2,]

#Create a plot for predictions on training
predict_train_plot = np.empty((total_size,1))
predict_train_plot[:, :] = np.nan
predict_train_plot[0:len(predict_on_train), :] = predict_on_train

#Create a plot for predictions on testing
predict_test_plot = np.empty((total_size,1))
predict_test_plot[:, :] = np.nan
predict_test_plot[len(predict_on_train):total_size, :] = predict_on_test

#Plot the graphs
plt.figure(figsize=(20,10)).suptitle("Plot Predictions for Original, Training
& Test Data", fontsize=20)
plt.plot(orig_plot)
plt.plot(predict_train_plot)
plt.plot(predict_test_plot)
plt.show()
```

Forecasting service loads: Predictions

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] The model that we have built can predict the next value in the time series once we provide the previous 168 actual values. This means it can only predict for the first hour of the next week. For predicting the second hour, we will be missing the actual value for the first hour. How do we then predict for an entire week? The quote for this prediction is in section 5.6.

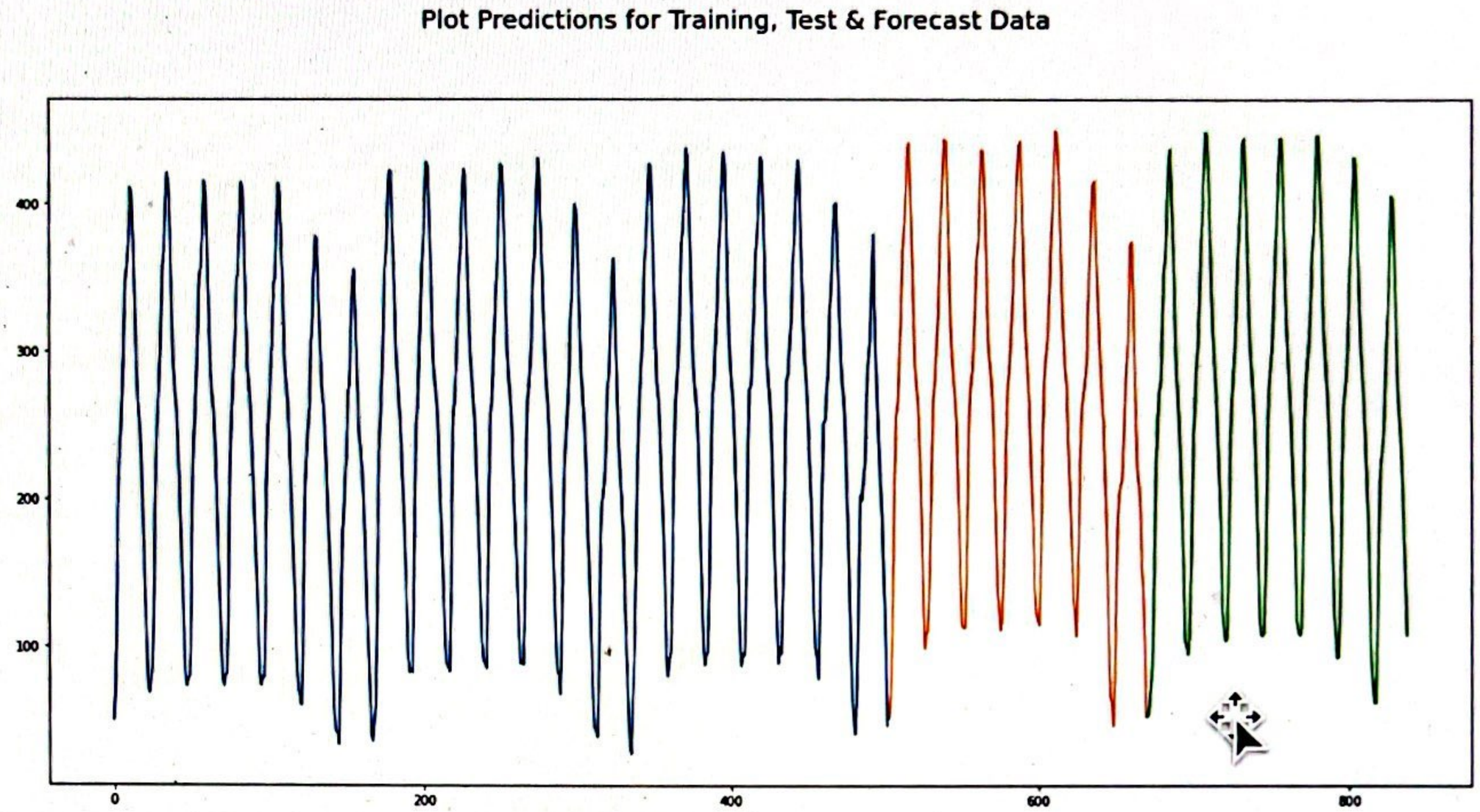
```
#Use last part of the training data as the initial lookback
curr_input= test_req_x[-1,:].flatten()

#Predict for the next week
predict_for = 24 * 7

for j in range(predict_for):

    #Take the last lookback no. of samples as X
    this_input=curr_input[-lookback:]
    #Create the input
    this_input=this_input.reshape((1,1,lookback))
    #Predict for the next point
    this_prediction=ts_model.predict(this_input)

    #Add the current prediction to the input
    curr_input = np.append(curr_input,this_prediction.flatten())

#Extract the last predict_for part of curr_input, which contains all the new
predictions
predict_on_future=np.reshape(np.array(curr_input[-
predict_for:]),(predict_for,1))

# #Inverse to view results
predict_on_future=scaler.inverse_transform(predict_on_future)

print(predict_on_future[:5])
```

```
[[ 52.72603437]
 [ 54.45360355]
 [ 70.10673284]
 [ 88.4503461 ]
 [150.5231352 ]]
```

```
#Plot the training data with the forecast data
total_size = len(predict_on_train) + len(predict_on_test) +
len(predict_on_future)

#Setup training chart
predict_train_plot = np.empty((total_size,1))
predict_train_plot[:, :] = np.nan
predict_train_plot[0:len(predict_on_train), :] = predict_on_train

#Setup test chart
predict_test_plot = np.empty((total_size,1))
predict_test_plot[:, :] = np.nan
predict_test_plot[len(predict_on_train):len(predict_on_train)+len(predict_on_
test), :] = predict_on_test

#Setup future forecast chart
predict_future_plot = np.empty((total_size,1))
predict_future_plot[:, :] = np.nan
predict_future_plot[len(predict_on_train)+len(predict_on_test):total_size, :]
= predict_on_future

plt.figure(figsize=(20,10)).suptitle("Plot Predictions for Training, Test &
Forecast Data", fontsize=20)
plt.plot(predict_train_plot)
plt.plot(predict_test_plot)
plt.plot(predict_future_plot)
plt.show()
```

In order to take care of the missing value in the look back sequence, we can use the predicted value in the look back. So for the first hour prediction, P1, we will use the last 168 values of the actual data. Then for the next hour, P2, we will use our first prediction P1 along with the last 167 values from the actual data. This provides contiguous values for the previous sequence even though one of them is predicted. We can continue in the same pattern and keep adding the current prediction to the look back for the next prediction. Do note that as we keep using predictions instead of actuals in the input, the output will tend to become less and less accurate and cannot keep up the patterns after a period of time. So it's recommended to not predict for a future sequence whose size is greater than the look back size. In this case, both are one week. This code block implements the logic to create look backs with predictions. Then we do inverse transform to scale back

the values. Finally, we will plot the training, prediction on test, and prediction for the future in a single plot.



The blue line is for predictions for four weeks of training data. The orange line is for predictions for one week of test data. The green line is for one week of future using predicted values and look backs. As we can see, the cyclic and seasonal patterns are maintained in the future predictions also. This completes our exercise on LSTM.