Time Series Forecasting with Neural Nets

Rick Alayza, Trincy Thomas Kozhikkadan, Krystofer Newman

***Abstract*—This project uses a Long Short-Term Memory (LSTM) recurrent neural network (RNN) for time series prediction based on a time series data set that has 275 data points (y(n), n=1, 2, …, 275).**

***Keywords*—Recurrent neural networks, Long Short-Term Memory Networks, Sigmoid**

I. Introduction

Recurrent Neural Network computations allow for information to persist during computation through connected nodes forming a directed graph. Unlike traditional neural networks, recurrent neural networks or RNNs use their internal states to process information such as time series data. These models allow for computations like the given data set of 275 points and also allow for predictions using y(n+T) where T=30. This report will have an overview of recurrent neural networks, long short-term memory networks, computation and its results.

II. RNNs & LSTM Networks

Long Short-Term Memory networks (LSTMs) are a variant of Recurrent Neural Networks that allow for learning of long-term dependencies. These networks are designed for remembering information for long periods of time due to the implemented chain of neural nets. Figure one shows an example of an LSTM and RNN architecture [1].

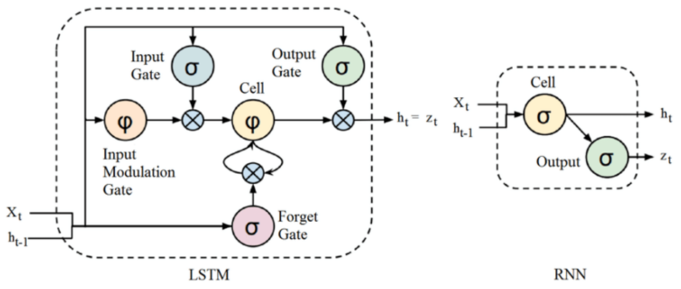

Figure 1. Examples of LSTM and RNN architectures

In a typical LSTM model, a chain of modules as pictured to the left in Figure 1, will be linked together to pass a vector from one node to the next during computation [2]. These links allow predictions to be trained on the memory of the past inputs of the vectors. These LSTM chains of neural nets allow for solving problems of all types including speech and handwriting recognition.

III. Python LSTM Neural Network

To setup the neural network correctly, the data was vectorized. This vectorization is a powerful method to package the information in a way that the RNN can us it. What makes this problem unique is the time series aspect. That signifies that the data set does not have multiple features. This neural network does not apply to other situations were multiple features are part of the dataset. I the process of creating the neural network, it is important to understand the packages used in the code as well as the actual coding language. The neural network was implemented and tested with the help of website focused on the topic of machine learning [3].

A. Python Packages

Implementing Python was the solution to this problem, along with other installations. The first and most important Python installation was Keras. Keras is an API for high level neural networks [4]. Within Keras, the Sequential function was called in the code listed in the Appendix. The Sequential function allows for linear layers to be stacked. These layers are associated with the layers in a neural network. With a simple adjustment, the number of neural network layers can be increased or decreased. For this solution, two layers were added. The Dense layer was used for the operation implementation. This full connected neural network has every input node and output node connected. The LSTM layer has the recurrent neural network (RNN). This layer will model the behavior of the training data in a time sequence. This layer is what separates this RNN from other designs (Figure 1).
The next important Python installation was NumPy. NumPy is a scientific computing package [5]. This package allows for creation and manipulation of arrays objects. This array object can be in the form of a vector, vertical, array, horizontal, and matrix, multi-dimensional. Mainly in this solution, vectors and arrays were implemented. For example, the code would not work if the dataset in the Excel file was only ported into a variable. The values had to be conditioned and formed before executing the RNN construction and training. In this solution, the dataset is imported from the provided spreadsheet, separated into respective training and testing sets, then packaged as a dataset (Line 12) using NumPy. Even after packaging in the create_dataset, NumPy was used again to reshape each sub-dataset into usable dimensions for the LSTM with datapoint dimensions of 3x1x1. That was the dimensional criteria for LSTM. Understanding the capability of this

package and how it played a key role in the representation of information was vital aspect of this solution.

The final important package installed in Python was TensorFlow. TensorFlow is a machine learning framework[6]. TensorFlow works in conjunction with Keras. This program is mainly tasked with dataflow structures. For example, Keras will construct the deep learning model while using TensorFlow functions and symbolic library. Essentially, Keras is a program wrapper for TensorFlow.

The remaining packages are important in execution but used in limited amounts in the code compared to the former packages. Pandas is a data structure tool [7]. Often used in data analysis, pandas was only used to interface with the spreadsheet to extract the data. Sklearn is a Python library for machine learning. This package is built on top of NumPy[8]. This package was mainly used to apply a scaler fit on the whole dataset. The scaler fit functions to bring the values in the dataset to a range between -1 and 1. This normalization functions to remove extremes in the dataset that might influence the machine learning. The final package required to execute the code was matplotlib. Matplotlib is library that allows for programs like Python to plot data. This package was only used at the end of the code to generated a visual representation of the dataset, training predictions, and test predictions.

B. Code

Along with understanding important packages used in a particular, it is vital to understand the individual executions or functions called. This section will highlight unique sections of code and offer an explanation relative to the overall objective or task. The Sequential command in Line 45 starts the process to construct the neural network. It defines the type of neural network to be used. As mentioned earlier, this command called a neural network model specific to time series data. Till this point in the code, the definition of the RNN has not been made. The previous lines only import, condition, and shape the given data to be used train the neural network. Line 46 defines the number of hidden neurons in the single hidden layer and the input shape. This code execution had 40 hidden neurons. The default used in the tutorial was set to a value of 4. In experimenting with the neural network on this problem, the number of hidden layers did not have a significant impact on the output and training accuracy. It is important to note that the LSTM neurons use the default sigmoid activation formula (Equation 1). The input shape of 1 has the dimension of 3x1x1. This value was not modified because this value would prevent the code from executing if the incorrect value is set. The next line used the Dense function. For this classical neural network design had a singular output. Increasing this value increases the number of output nodes. For a time-series neural network, a singular output was sufficient. The compiler function (Line 48) configures the model for training [4]. Within the compile function, the loss function was defined. For this instance, the loss function used when compiling was the root mean square error. This applied the root mean square error calculation between the predicted value output of the neural network and the actual value in the dataset to calculate the loss as the neural network trains. The final line for this section initializes the training of the model to the number of epochs defined. This function takes the input values, specifically NumPy values. Epochs is the number of iterations over the whole dataset. For example, 2 epochs will train through the entire dataset twice regardless of the number of iterations performed on each datapoint.

$$\frac{1}{1+e^{x}} \qquad 1$$

IV. RESULTS

The dataset was divided into two sub-datasets. One was used to train the RNN and the other was used to test the neural network predictions against actual values. The required prediction of 30 values beyond the original dataset is included in the test sub-dataset. Figure 2 is the output of the code with a time-series along the x-axis, or index, and the dataset value for the y-axis. The blue line represented the dataset. The orange line represented the values used in the training. The green line represented the test sub-dataset. The test values overlapping the blue indicate the values that were used to compare predictions at the end of the dataset. The remaining green line were the actual predicted values generated by the neural network. Figure 3 is a zoomed in image of a portion of the test values and the predicted values. According to the code output, the root mean square error for the training was 19.86. By visual inspection, the neural network did a good job of predicting the values during training. More so, the overlap in the test group also has a good level of accurate prediction. Table 1 has the values for the 30 predicted points.

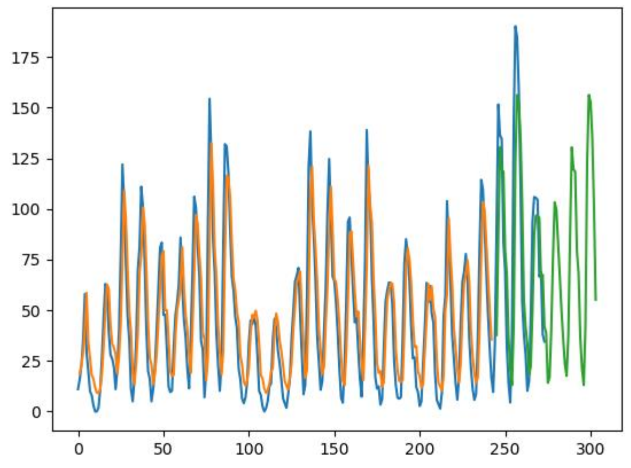

Figure 2 Code Output

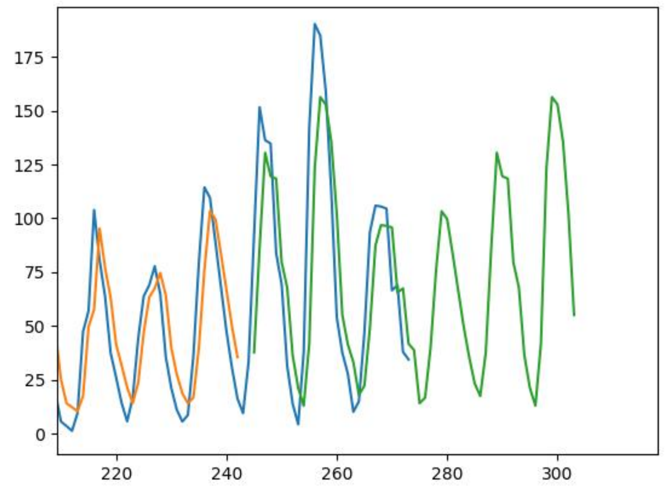

Figure 3 Code Output (Zoomed View)

Table 1: Predicted Values			
1	38.85733	16	86.48565
2	38.85708	17	130.45955
3	14.172772	18	119.55465
4	16.766094	19	118.3932
5	40.212147	20	79.59135
6	76.22963	21	67.89791
7	103.32304	22	36.311058
8	99.67182	23	21.250668
9	83.486046	24	13.047722
10	66.592926	25	41.818687
11	49.8065	26	123.44533
12	35.54595	27	156.28847
13	23.315609	28	152.82361
14	17.543253	29	135.6004
15	37.75463	30	101.72965

A. *Interesting Observation*

Figure 4 was included because it was an interesting observation. Figure 4 was a result of removing the scaler fit transformation (Line 28). Without the fit transformation, the neural network was not able to accurately predict the peaks of the dataset.

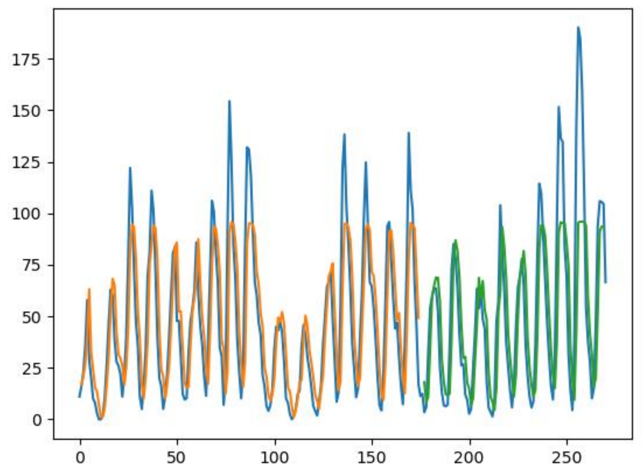

Figure 4: Code Output without Regularization

V. Andrew Ng – Connections

The connections between this project and educational video done by Prof. Andrew Ng are few. Andrew Ng does discuss the theory of gradient descent and cost functions. The theory of the cost function does hold in this instance. The cost function in this application was seen through the loss value. The loss value for each epoch indicates the minimization of the cost function. On the other hand, the gradient descent does not have significant influence. Prof. Andrew Ng stressed the importance of the gradient descent but time-series functions are unique in this situation. The LSTM neural network is tuned using backpropagation and overcomes the vanishing gradient problem [3]. There is a good overlap over the sidmoid function (Equation 1). This value is important in application but is only stressed in the Prof. Andrew Ng's online course on Coursera. The Coursera online class had programming labs that allowed the student to implement the sigmoid activation function relative to the desired neural network application. It is important to note that the online Coursera course does not cover time-series neural networks. The online class, as well as this class, focuses on a more classical application of neural networks where datasets have one or more features, general more than one.

VI. References

[1] https://www.researchgate.net/figure/An-example-of-a-basic-LSTM-cell-left-and-a-basic-RNN-cell-right-Figure-follows-a_fig2_306377072

[2] http://colah.github.io/posts/2015-08-Understanding-LSTMs

[3] https://machinelearningmastery.com/time-series-prediction-lstm-recurrent-neural-networks-python-keras/

[4] https://keras.io/

[5] http://www.numpy.org/

[6] https://www.tensorflow.org/api_docs/python/

[7] https://pandas.pydata.org/

[8] http://scikit-learn.org/stable/

VII. APPENDIX

A. Code

```
#Project 3 Team 5 : Krys Newman, Trincy Kozhikkadan, Rick
Alayza
import numpy
import matplotlib.pyplot as plt
from pandas import read_csv
import math
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
# convert an array of values into a dataset matrix
def create_dataset(dataset, look_back=1):
        dataX, dataY = [], []
        for i in range(len(dataset)-look_back-1):
                a = dataset[i:(i+look_back), 0]
                dataX.append(a)
                dataY.append(dataset[i + look_back, 0])
        return numpy.array(dataX), numpy.array(dataY)
# fix random seed for reproducibility
numpy.random.seed(7)
# load the dataset
dataframe = read_csv('C:\Python35\Scripts\project2_time
series data_students.csv', usecols=[1], engine='python',
skipfooter=0,header=0)
dataset = dataframe.values
dataset = dataset.astype('float32')

# normalize the dataset
scaler = MinMaxScaler(feature_range=(0, 1))
dataset = scaler.fit_transform(dataset)

# split into train and test sets
train_size = int(len(dataset) * 0.80)
test_size = len(dataset) - train_size
# Predicted values are lumped in with test
train, test = dataset[0:train_size,:],
dataset[train_size:len(dataset),:]
# reshape into X=t and Y=t+1
look_back = 1
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back)
# reshape input to be [samples, time steps, features]
trainX = numpy.reshape(trainX, (trainX.shape[0], 1,
trainX.shape[1]))
testX = numpy.reshape(testX, (testX.shape[0], 1,
testX.shape[1]))

# create and fit the LSTM network
model = Sequential()
model.add(LSTM(40, input_shape=(1, look_back)))

model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(trainX, trainY, epochs=100, batch_size=1,
verbose=2)
# make predictions
trainPredict = model.predict(trainX)
testPredict = model.predict(testX)
# invert predictions
trainPredict = scaler.inverse_transform(trainPredict)
trainY = scaler.inverse_transform([trainY])
testPredict = scaler.inverse_transform(testPredict)
testY = scaler.inverse_transform([testY])

# calculate root mean squared error
trainScore = math.sqrt(mean_squared_error(trainY[0],
trainPredict[:,0]))
print('Train Score: %.2f RMSE' % (trainScore))
# shift train predictions for plotting
trainPredictPlot = numpy.empty_like(dataset)
trainPredictPlot[:, :] = numpy.nan
trainPredictPlot[look_back:len(trainPredict)+look_back, :] =
trainPredict
# shift test predictions for plotting
testPredictPlot = numpy.empty_like(dataset)
testPredictPlot[:, :] = numpy.nan
testPredictPlot[len(trainPredict)+(look_back*2)+1:len(dataset)
-1, :] = testPredict
# 30 point Prediction values y(n+30)
submit = testPredict[-30:]
print(submit, submit.shape)
# plot baseline and predictions
plt.plot(scaler.inverse_transform(dataset)[0:274,:])
plt.plot(trainPredictPlot)
plt.plot(testPredictPlot)
plt.show()
```