Bishop's University

CS 596 – Special Topics on Deep Learning

Final project: Exploring Recurrent Neural Networks (RNNs) with PyTorch

Full names	Students id	Contributions
Yuming Su	002344631	report and coding
Dan Luo	002352806	test and coding

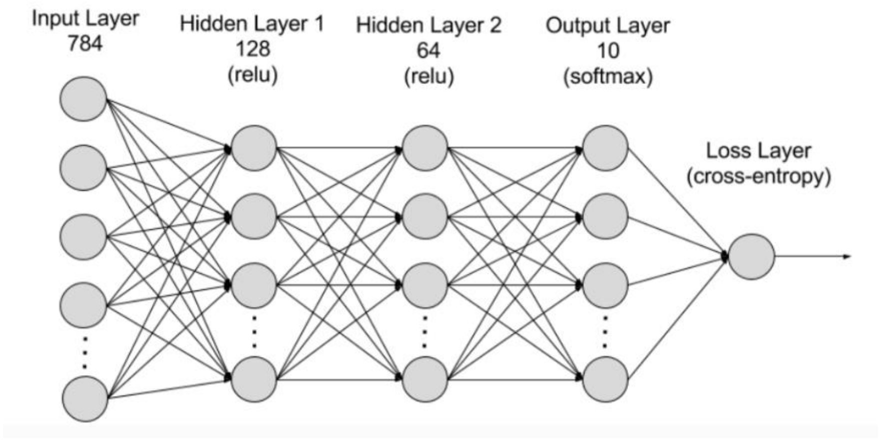
Notes:
Detal inforamtion please check the .ipynb file.

Background Research:

Recurrent neural networks (RNNs) are a type of deep learning model. Usually a RNN is trained to process and convert the sequential data input into a specific sequential data output.

We learned CNN architecture in this course and it is very effective for tasks involving spatial relationships within the data such as images. Not like CNN, RNN is better used in analyzing temporal and sequential data, such as text, or statstic results. It is popular in time series analysis, natural language processing (NLP), speech recognition, and handwriting recognition, etc.

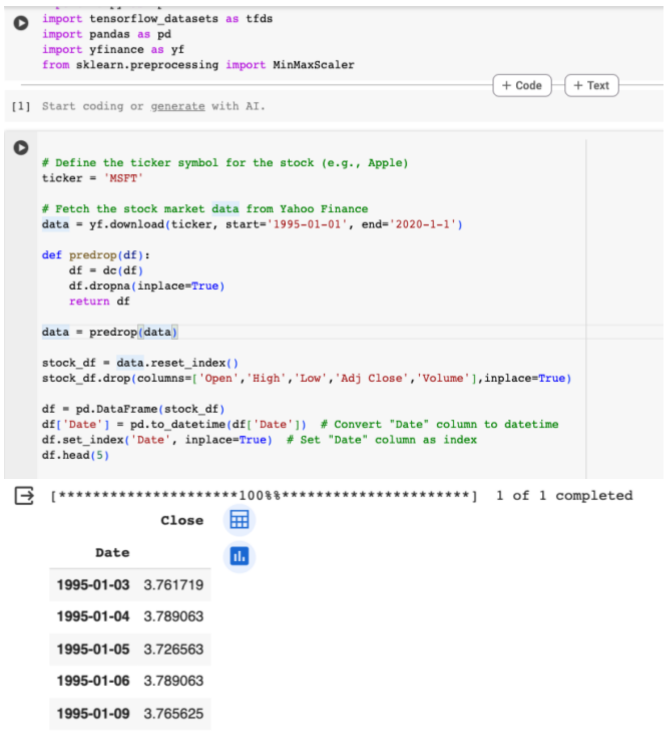
The basic structure of RNN neurons are organized as input, output, and hidden layers. The input layer receives the information to process, and the output layer provides the result. Data processing, analysis, and prediction are made in the hidden layer.
Here is a example:



Processing Steps:

We will work on Jupyter Notebook of Google Colab. First of all, we need to decide our research approach and data selection. We will pick up the data, a stock price of time series, because this kind of analyzing is popular in RNNs and the data is easy to access and stable.

For our required data, we could utilize importing yfinance library into Python code directly. yfinance is a popular open source library developed by Yahoo.



```
import tensorflow_datasets as tfds
import pandas as pd
import yfinance as yf
from sklearn.preprocessing import MinMaxScaler
```

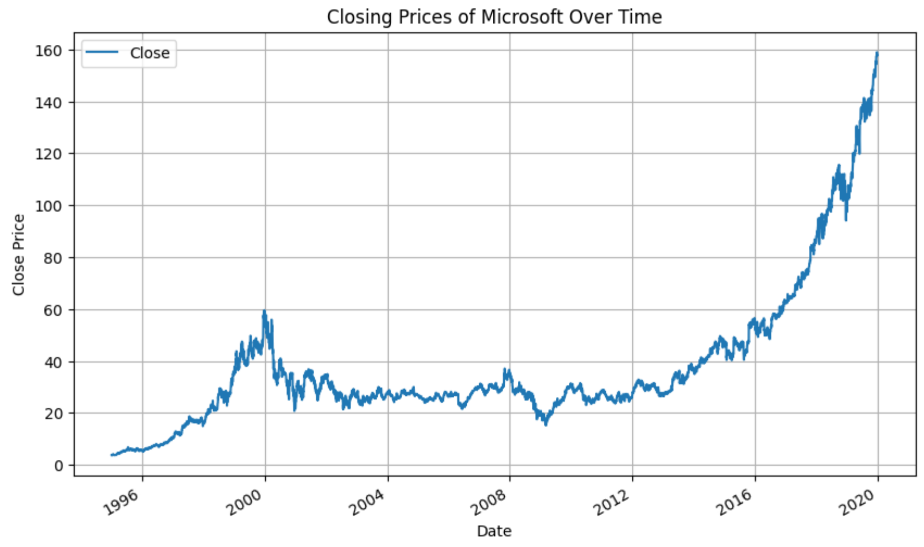
```
[1] Start coding or generate with AI.
```

```
# Define the ticker symbol for the stock (e.g., Apple)
ticker = 'MSFT'

# Fetch the stock market data from Yahoo Finance
data = yf.download(ticker, start='1995-01-01', end='2020-1-1')

def predrop(df):
    df = dc(df)
    df.dropna(inplace=True)
    return df

data = predrop(data)

stock_df = data.reset_index()
stock_df.drop(columns=['Open','High','Low','Adj Close','Volume'],inplace=True)

df = pd.DataFrame(stock_df)
df['Date'] = pd.to_datetime(df['Date'])  # Convert "Date" column to datetime
df.set_index('Date', inplace=True)  # Set "Date" column as index
df.head(5)
```

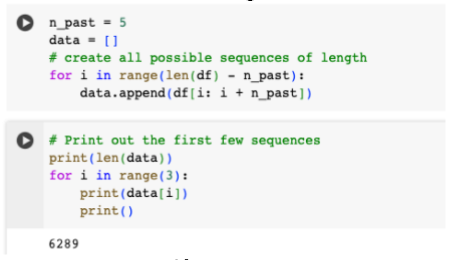
```
[*********************100%%**********************]  1 of 1 completed
```

	Close
Date	
1995-01-03	3.761719
1995-01-04	3.789063
1995-01-05	3.726563
1995-01-06	3.789063
1995-01-09	3.765625

Through this method, we can fetch the standard stock data with a start time and end time. We pick up the stock price of microsoft from 1995-1-1 to 2020-1-1 as our data, and drop the useless columns, because usually the stock type of data is consisted with 6 parts which are: Open, High, Low, Adj Close, Volume, Close. We only keep close price.
Here is a clear preview chart of our data:



Now, we start to make our time sequences data. It is an important part in our RNN architecture. Like we mentioned before, RNN models are specifically designed to handle sequential data. Here we will have the time series.
We created sequences of historical data by the number of days passed on the DataFrame.
This basically means that we converted the original dataset to time series dataset. Inside each one of our time series dataset, it includes all the data of certain time period.
The data choose is :current index + timeline -1.
Here is an example, the timeline is 5 and current index is 1. We will have the first 5 close price as our first value. We print out the first 3 lines to show:

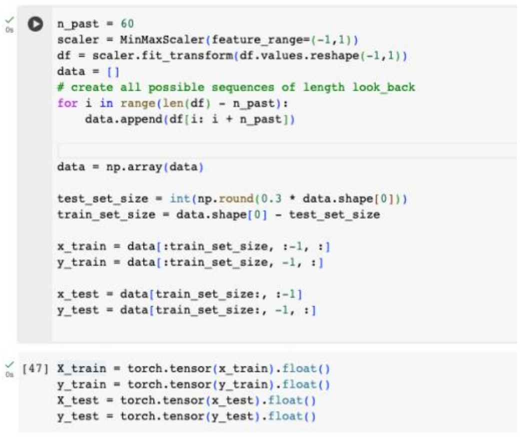


```
n_past = 5
data = []
# create all possible sequences of length
for i in range(len(df) - n_past):
    data.append(df[i: i + n_past])
```

```
# Print out the first few sequences
print(len(data))
for i in range(3):
    print(data[i])
    print()
```

```
6289
                Close
Date
1995-01-03   3.761719
1995-01-04   3.789063
1995-01-05   3.726563
1995-01-06   3.789063
1995-01-09   3.765625

                Close
Date
1995-01-04   3.789063
1995-01-05   3.726563
1995-01-06   3.789063
1995-01-09   3.765625
1995-01-10   3.812500

                Close
Date
1995-01-05   3.726563
1995-01-06   3.789063
1995-01-09   3.765625
1995-01-10   3.812500
1995-01-11   3.835938
```

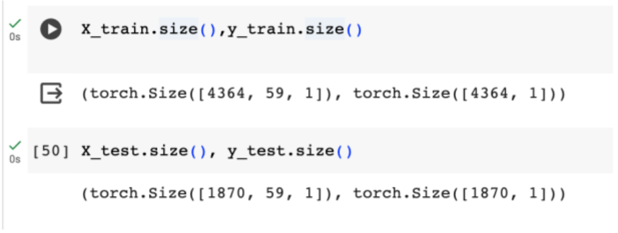
In actual performance, the 5 is too little to our data. We tried 30 and 60. 60 may give a better result as the time period we use.



```
n_past = 60
scaler = MinMaxScaler(feature_range=(-1,1))
df = scaler.fit_transform(df.values.reshape(-1,1))
data = []
# create all possible sequences of length look_back
for i in range(len(df) - n_past):
    data.append(df[i: i + n_past])


data = np.array(data)

test_set_size = int(np.round(0.3 * data.shape[0]))
train_set_size = data.shape[0] - test_set_size

x_train = data[:train_set_size, :-1, :]
y_train = data[:train_set_size, -1, :]

x_test = data[train_set_size:, :-1]
y_test = data[train_set_size:, -1, :]
```

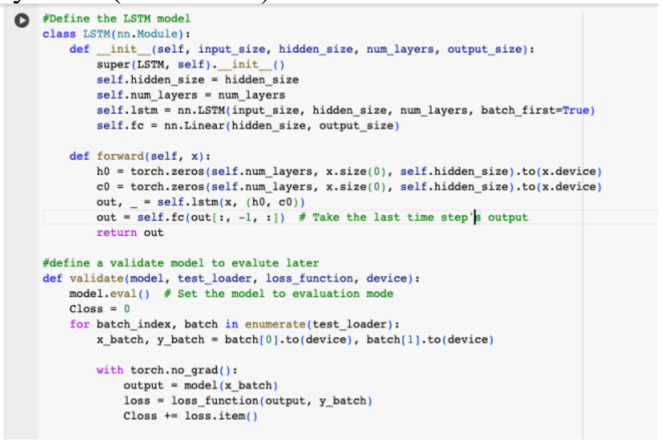
```
[47] X_train = torch.tensor(x_train).float()
     y_train = torch.tensor(y_train).float()
     X_test = torch.tensor(x_test).float()
     y_test = torch.tensor(y_test).float()
```

Now start to prepare the process our data. We scaled the dataframe using MinMaxScaler to normalize the data to a specified range (-1 to 1). Then we convert the data into numpy arrays and split it into training and test sets. The split size is set to 30% which means 30 percent of data for trainning and 70 percent of data for testing. X are used for input features and y used as output features.



```
X_train.size(),y_train.size()

(torch.Size([4364, 59, 1]), torch.Size([4364, 1]))

[50] X_test.size(), y_test.size()

(torch.Size([1870, 59, 1]), torch.Size([1870, 1]))
```

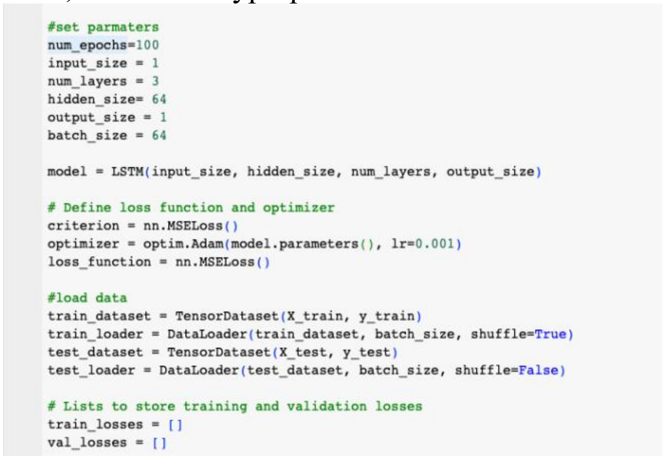
Then we start to define model. This is a basic LSTM (Long Short-Term Memory) model using PyTorch (nn.Module).



```
#Define the LSTM model
class LSTM(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, output_size):
        super(LSTM, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)
        c0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)
        out, _ = self.lstm(x, (h0, c0))
        out = self.fc(out[:, -1, :])  # Take the last time step's output
        return out

#define a validate model to evalute later
def validate(model, test_loader, loss_function, device):
    model.eval()  # Set the model to evaluation mode
    Closs = 0
    for batch_index, batch in enumerate(test_loader):
        x_batch, y_batch = batch[0].to(device), batch[1].to(device)

        with torch.no_grad():
            output = model(x_batch)
            loss = loss_function(output, y_batch)
            Closs += loss.item()
```

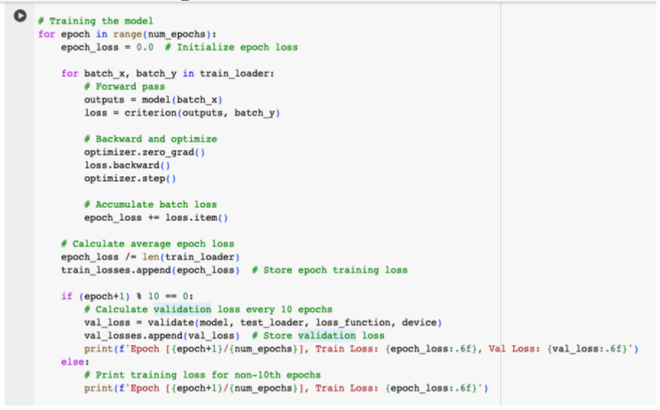
LSTM is a type of recurrent neural network (RNN) designed to overcome the vanishing gradient problem and capture long-term dependencies in sequential data. It consists of memory cells and gates that regulate the flow of information, allowing the model to retain important information over long sequences.
In the __init__ method of LSTM model, we will have some important parameters to test in later experiment. The "forward" part of code defines how the input data flows through the network and

produces output. It’s important to this model.
Next, we set the hyperparameters and load data:



```
#set parmaters
num_epochs=100
input_size = 1
num_layers = 3
hidden_size= 64
output_size = 1
batch_size = 64

model = LSTM(input_size, hidden_size, num_layers, output_size)

# Define loss function and optimizer
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
loss_function = nn.MSELoss()

#load data
train_dataset = TensorDataset(X_train, y_train)
train_loader = DataLoader(train_dataset, batch_size, shuffle=True)
test_dataset = TensorDataset(X_test, y_test)
test_loader = DataLoader(test_dataset, batch_size, shuffle=False)

# Lists to store training and validation losses
train_losses = []
val_losses = []
```

After these steps, we could start to train and evaluate the result:



```
# Training the model
for epoch in range(num_epochs):
    epoch_loss = 0.0  # Initialize epoch loss

    for batch_x, batch_y in train_loader:
        # Forward pass
        outputs = model(batch_x)
        loss = criterion(outputs, batch_y)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # Accumulate batch loss
        epoch_loss += loss.item()

    # Calculate average epoch loss
    epoch_loss /= len(train_loader)
    train_losses.append(epoch_loss)  # Store epoch training loss

    if (epoch+1) % 10 == 0:
        # Calculate validation loss every 10 epochs
        val_loss = validate(model, test_loader, loss_function, device)
        val_losses.append(val_loss)  # Store validation loss
        print(f'Epoch [{epoch+1}/{num_epochs}], Train Loss: {epoch_loss:.6f}, Val Loss: {val_loss:.6f}')
    else:
        # Print training loss for non-10th epochs
        print(f'Epoch [{epoch+1}/{num_epochs}], Train Loss: {epoch_loss:.6f}')
```

We will experiment with different hyperparameters (learning rate, batch size, epochs, time sequences, number of hidden layers) to optimize the performance of your model.

Analysis and Interpretation:

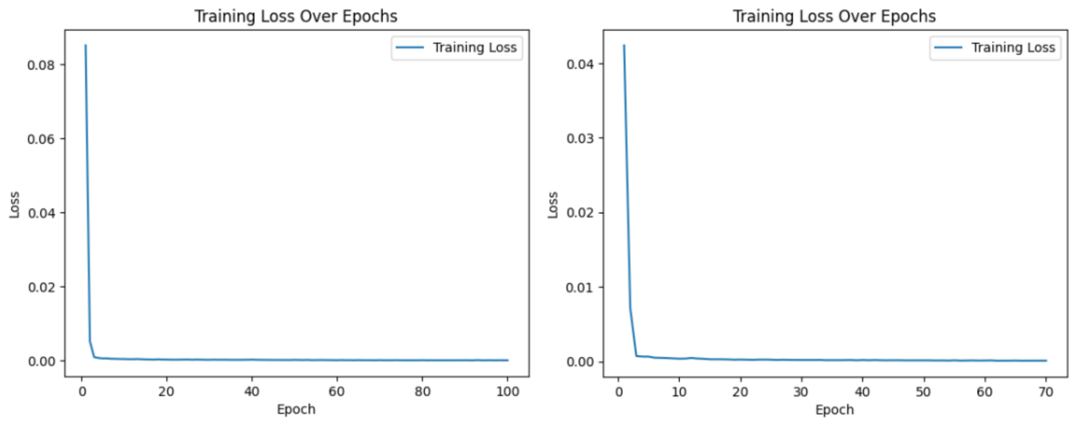
We will compare the average test Loss. After many times running, We have this table:

Learning rate	Batch size	Epochs	Time of Sequence	Hidden Layer Size	Result: Test Loss R-squre
0.01	52	50	30	38	0.0919 0.6217
0.01	64	50	30	64	0.0303 0.8770
0.001	64	100	60	64	0.0312 0.9130
0.01	64	100	5	64	0.0403 0.8211
0.01	64	125	30	72	0.0760

					0.6709
0.001	**64**	**70**	**30**	**72**	**0.0165 0.9326**
0.001	64	60	60	72	0.0234 0.9156

The best result we got is from the 6th in which test learning rate=0.001, Batch size=64, Epochs=70, Time of sequence=30, Hidden layers=72. It has average test Loss= 0.0165 and R-squre=0.9326.

In our test, we focus on learning rate, epochs, time of sequence and hidden layers.
Obvious learning rate 0.001 is better than 0.01 in our model.
We didn't do a lot of tests on batch size.
Eopch is not that important in our model. We only need to make sure the loss of train because stable in epochs. Here is a train loss plot from fifth test:



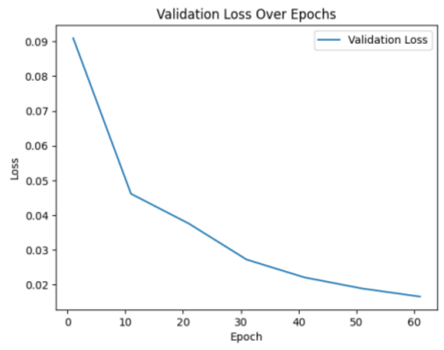
If we compare with the best result chart, we could find that, the train loss happens not that often expect the first 5 to 10 epochs.

For time of Sequence, we find that the difference between choose 30 days and 60 days is not very obvious. Sometimes we pick up 30 days, the test result even better than 60 days. However, if you pick up too low time sequences like we picked 5 on our test, the result will be affected. We need to make sure there are enough time sequence period to use in trainning process, because the time sequence length determines the amount of historical data which the model can utilize to make predictions. In the same way, we found that longer time sequence can lead more time spending.

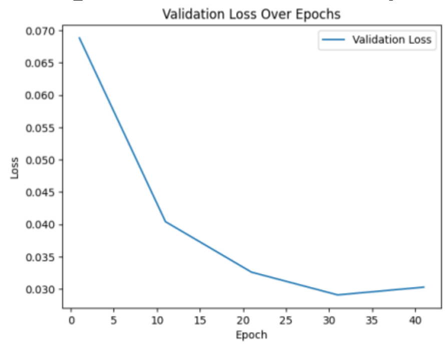
The Hidden layer in the LSTM model processes input sequences over time, updating hidden states at each time step and capturing temporal dependencies in the data. It plays a crucial role in learning representations of the input data and making predictions based on sequential information. Here we found that larger hidden layer size can help to improve the prediction result, but the speed of running may get slower too.

0.001	**64**	**70**	**30**	**72**	**0.0165 0.9326**

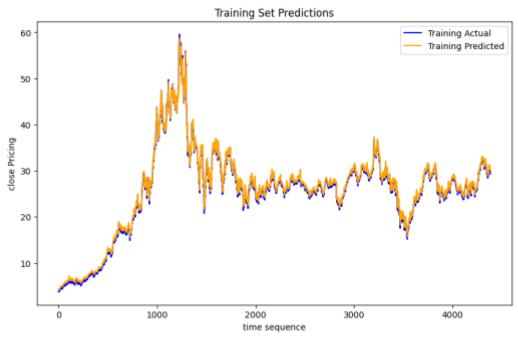
Here are some plots of the best result in which test learning rate=0.001, Batch size=64, Epochs=70, Time of sequence=30, Hidden layers=72, average test loss=0.0165, R-squre=0.9326 to demostrate the result:

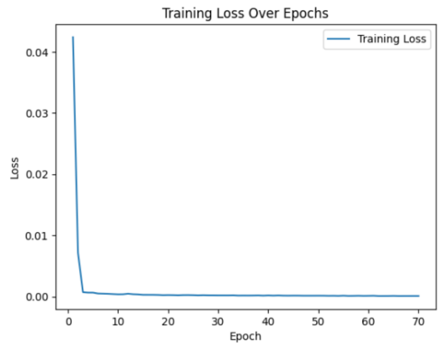


For the validation of Loss, we can compare with the second test which has 0.01, 64, 50, 30, 64. average test loss=0.0303. R-squre=0.8770 :

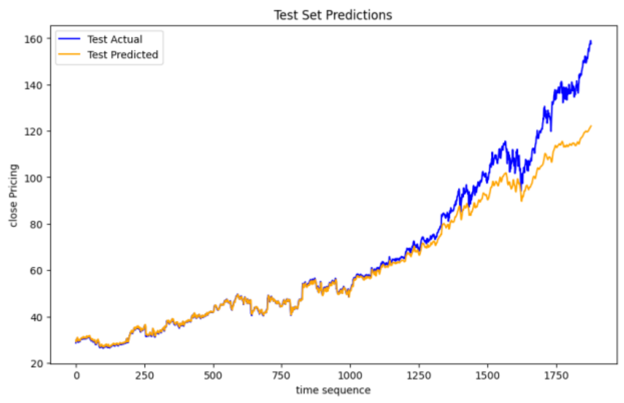


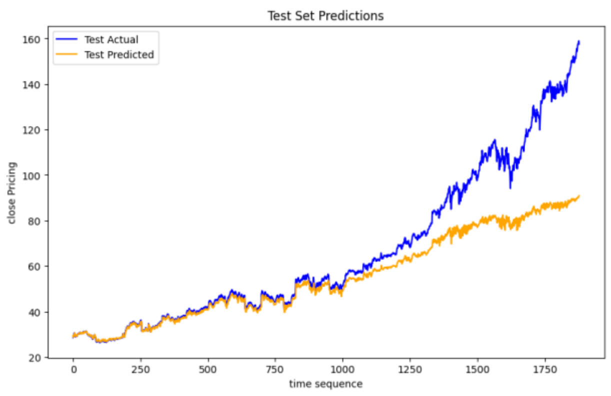
More charts from the best test:





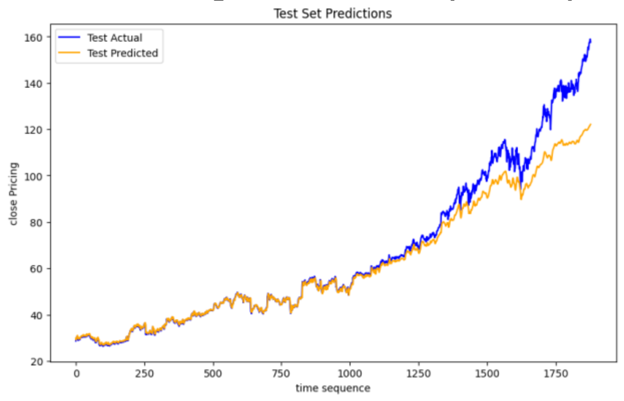
The best demostrate result plot is the test set prediction. We could clearly see our predicted result and actual result on test set. Here is a compare with the fifth test:

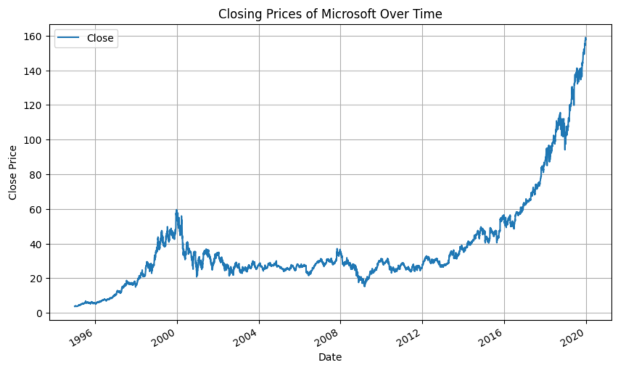




we could find the difference on the 2 charts. The second one is from the fifth test which test loss is about 0.07 and R-squre is about 0.67.

The interest thing is that if we compare this plot with the original data stock price plot:





We could see that, the prediction is mainly get worse from 2016 to 2020. That is very difficult to predict, because it's more like the uncontrollable outside economic influence.

In conclusion, we may say that using matchine learning to predict stock prices may not be always a good idea, because sometimes the stock market and complex economic environment can not always be learned from history trainning process. However, using matchine learning in predict stock price is still a very valuable tool to analysis and reference. While RNNs, particularly LSTM networks, are well-suited for modeling sequential data and capturing temporal dependencies based on their particular characteristics.