Lesson 27 DeepLearning Application Last Update: June3, 2021

Stock Market Application

We want to be able to predict the stock market especially the SP500. The SP500 contains 500 Stocks like Apple, Netflix, Google, FaceBook etc. The SP500ondex is calculated from the sum on all theses stocks per minute.

We have a stock market file **'data_stocks.csv'** that contains one minute periods of all stocked values for 4 months.

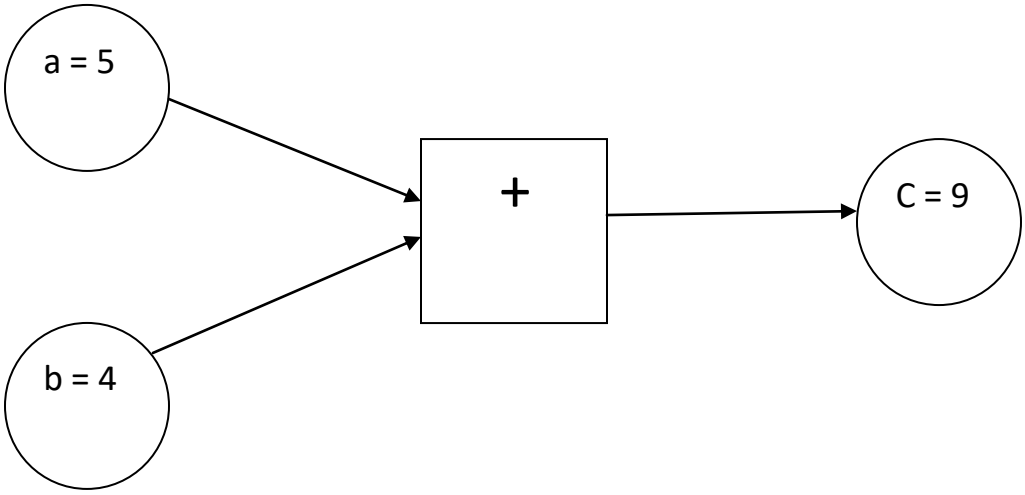
We will use the stock price values for the X data and the total sum the SP500 index for the Y data. Our goal is the fit the X data to predict the Y data of the SP500 index.

Which means if we know the values of all the stock for a row now we can predict if the SP500 will go up and down. We can then make a '**buy**' trade on the SP500 and then as soon as the SP500 start to go down we can cash in.

This program just works on the data file but does not connect directly to the stock market value feeds.

We are using tensor flow for out neural networks.

TensorFlow operates on a graph representation of the underlying computational task. This approach allows the user to specify mathematical operations as elements in a graph of data, variables and operators. Since neural networks are actually graphs of data and mathematical operations, TensorFlow is well suited for neural networks and deep learning.



In the figure above, two numbers are supposed to be added. Those numbers are stored in two variables, a and b. The two values are flowing through the graph and arrive at the square node, where they are being added. The result of the addition is stored into another variable, c. Actually, a, b and c can be considered as placeholders. Any numbers that are fed into a and b get added and are stored into c. This is exactly how TensorFlow works. The user defines an abstract representation of the model (neural network) through placeholders and variables. Afterwards, the placeholders get "filled" with real data and the actual computations take place. The following code implements the example from above in TensorFlow:

```
# Import TensorFlow
#import tensorflow as tf
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()

# Define a and b as placeholders
a = tf.placeholder(dtype=tf.int8)
b = tf.placeholder(dtype=tf.int8)

# Define the addition
c = tf.add(a, b)

# Initialize the graph
graph = tf.Session()
```

```
# Run the graph
result = graph.run(c, feed_dict={a: 5, b: 4})

# print result
print("result:",result)  #9

#close graph
graph.close()
```

After having imported the TensorFlow library, two placeholders are defined using tf.placeholder(). They correspond to the two circles on the left of the image above. Afterwards, the mathematical addition is defined via tf.add(). The result of the computation is c = 9. With placeholders set up, the graph can be executed with any integer value for a and b using **feed_dict={a: 5, b: 4}).** A tensor is returned with the value 9.

Todo:

Type in or copy/paste in the above code and run it. Try oyje values and mathematical operations like subtract, multiply and divide.

Predict SP500

We now apply tensor flow to predict SP500.

Here are the program steps:

Step 1 import tensor flow

```
# Import TensorFlow
#import tensorflow as tf
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
```

We have to add the additional import statement to prevent warnings and errors:

```
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
```

step 2 import rest of imports

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
```

step 3 load in data file

```
# Import data
df = pd.read_csv('data_stocks.csv')
print(df.head())
```

```
            DATE      SP500  NASDAQ.AAL  ...  NYSE.YUM  NYSE.ZBH  NYSE.ZTS
0     1491226200  2363.6101     42.3300  ...     63.86   122.000   53.3500
1     1491226260  2364.1001     42.3600  ...     63.74   121.770   53.3500
2     1491226320  2362.6799     42.3100  ...     63.75   121.700   53.3650
3     1491226380  2364.3101     42.3700  ...     63.88   121.700   53.3800
4     1491226440  2364.8501     42.5378  ...     63.91   121.695   53.2400
...          ...        ...         ...  ...       ...       ...       ...
41261 1504209360  2472.2200     44.7200  ...     76.88   114.310   62.7250
41262 1504209420  2471.7700     44.7300  ...     76.90   114.330   62.7100
41263 1504209480  2470.0300     44.7400  ...     76.88   114.310   62.6850
41264 1504209540  2471.4900     44.7100  ...     76.83   114.230   62.6301
41265 1504209600  2471.4900     44.7400  ...     76.81   114.280   62.6800
```

Step 4: drop the date column since it is not needed

```
# Drop date variable
df = df.drop(['DATE'], axis = 'columns')
```

step 5: get data from data frame

```
# get data
data = df.values
print("data")
print(data)
```

```
data
[[2363.6101  42.33   143.68  ...  63.86   122.     53.35  ]
 [2364.1001  42.36   143.7   ...  63.74   121.77   53.35  ]
 [2362.6799  42.31   143.6901 ...  63.75   121.7    53.365 ]
```

```
...
[2470.03   44.74   164.01  ...  76.88   114.31   62.685 ]
[2471.49   44.71   163.88  ...  76.83   114.23   62.6301]
[2471.49   44.74   163.98  ...  76.81   114.28   62.68  ]]
```

step 6: scale the complete data set

```
# scale data
scaler = MinMaxScaler()
scaler.fit(data)
data_scaled = scaler.transform(data)
```

step 7: separate data into target and features

the target is the sP500 index and the features are the individual stock prices

```
# set target and features
X = data_scaled[:,1:]
y = data_scaled[:,0]

print("scaled X")
print(X)
```

```
scaled X
[[0.10993038 0.14455852 0.05131045 ... 0.04878049 0.5092156  0.09098787]
 [0.11212898 0.14537988 0.0767811  ... 0.04017217 0.49935705 0.09098787]
 [0.10846464 0.14497331 0.07327427 ... 0.04088953 0.49635662 0.09228769]
 ...
 [0.28655185 0.97946612 0.99021779 ... 0.98278336 0.17959709 0.89991334]
 [0.28435324 0.97412731 0.98634182 ... 0.97919656 0.17616802 0.89515598]
 [0.28655185 0.97823409 0.99372462 ... 0.97776184 0.17831119 0.89948007]]
```

```
print("scaled Y")
print(y)
```

```
scaled Y
[0.21342456 0.21645842 0.20766516 ... 0.87233051 0.88137019 0.88137019]
0.005039798
```

step 8: split data into %80 train and 20% test set

 # split data set %80 train %20% test do not shuffle
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, shuffle=False)

step 9 set model parameters

The model consists of four hidden layers. The first layer contains 1024 neurons, slightly more than double the size of the inputs. Subsequent hidden layers are always half the size of the previous layer, which means 512, 256 and finally 128 neurons. A reduction of the number of neurons for each subsequent layer compresses the information the network identifies in the previous layers

 # Model architecture parameters

n_stocks = 500
n_neurons_1 = 1024
n_neurons_2 = 512
n_neurons_3 = 256
n_neurons_4 = 128
n_target = 1

Step 9: make tensor flow placeholders

We need two placeholders in order to fit our model: `X` contains the network's inputs (the stock prices of all S&P 500 constituents at time `T = t`) and Y the network's outputs (the index value of the S&P 500 at time `T = t + 1`).

The shape of the placeholders correspond to `[None, n_stocks]` with `[None]` meaning that the inputs are a 2-dimensional matrix and the outputs are a 1-dimensional vector. It is crucial to understand which input and output dimensions the neural net needs in order to design it properly.

 # Placeholder
X = tf.placeholder(dtype=tf.float32, shape=[None, n_stocks])
Y = tf.placeholder(dtype=tf.float32, shape=[None])

The `None` argument indicates that at this point we do not yet know the number of observations that flow through the neural net graph in each batch, so we keep if flexible.

Step 11 setup initializers

Initializers are used to initialize the network's variables before training. Since neural networks are trained using numerical optimization techniques, the starting point of the optimization problem is one the key factors to find good solutions to the underlying problem. There are different initializers available in TensorFlow, each with different initialization approaches. Here, is the **tf.variance_scaling_initializer(),** which is one of the default initialization strategies.

```
# Initializers
sigma = 1
weight_initializer = tf.variance_scaling_initializer(mode="fan_avg", distribution="uniform", scale=sigma)
bias_initializer = tf.zeros_initializer()
```

step 12 set up layers

While placeholders are used to store input and target data in the graph, variables are used as flexible containers within the graph that are allowed to change during graph execution. Weights and biases are represented as variables in order to adapt during training. Variables need to be initialized, prior to model training.

```
# Layer 1: Variables for hidden weights and biases
W_hidden_1 = tf.Variable(weight_initializer([n_stocks, n_neurons_1]))
bias_hidden_1 = tf.Variable(bias_initializer([n_neurons_1]))

# Layer 2: Variables for hidden weights and biases
W_hidden_2 = tf.Variable(weight_initializer([n_neurons_1, n_neurons_2]))
bias_hidden_2 = tf.Variable(bias_initializer([n_neurons_2]))

# Layer 3: Variables for hidden weights and biases
W_hidden_3 = tf.Variable(weight_initializer([n_neurons_2, n_neurons_3]))
bias_hidden_3 = tf.Variable(bias_initializer([n_neurons_3]))

# Layer 4: Variables for hidden weights and biases
W_hidden_4 = tf.Variable(weight_initializer([n_neurons_3, n_neurons_4]))
bias_hidden_4 = tf.Variable(bias_initializer([n_neurons_4]))

# Output layer: Variables for output weights and biases
W_out = tf.Variable(weight_initializer([n_neurons_4, n_target]))
bias_out = tf.Variable(bias_initializer([n_target]))
```

It is important to understand the required variable dimensions between input, hidden and output layers. As a rule of thumb in multilayer perceptrons (MLPs, the type of networks used here), the second dimension of the previous layer is the first dimension in the current layer for weight matrices. This might sound complicated but is essentially just each layer passing its output as input to the next layer. The biases dimension equals the second dimension of the current layer's weight matrix, which corresponds the number of neurons in this layer.

Step 13 Hidden layer

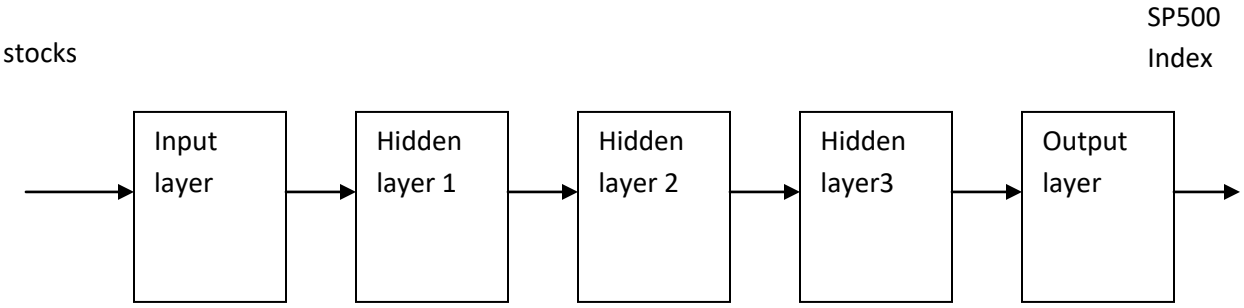
After definition of the required weight and bias variables, the network topology, the architecture of the network, needs to be specified. Hereby, placeholders (data) and variables (weighs and biases) need to be combined into a system of sequential matrix multiplications.

Furthermore, the hidden layers of the network are transformed by activation functions. Activation functions are important elements of the network architecture since they introduce non-linearity to the system. There are dozens of possible activation functions out there, one of the most common is the rectified linear unit (ReLU) which will also be used in this model. ReLU is an activation function where as the output is always positive

```
# Hidden layer
hidden_1 = tf.nn.relu(tf.add(tf.matmul(X, W_hidden_1), bias_hidden_1))
hidden_2 = tf.nn.relu(tf.add(tf.matmul(hidden_1, W_hidden_2), bias_hidden_2))
hidden_3 = tf.nn.relu(tf.add(tf.matmul(hidden_2, W_hidden_3), bias_hidden_3))
hidden_4 = tf.nn.relu(tf.add(tf.matmul(hidden_3, W_hidden_4), bias_hidden_4))

# Output layer (must be transposed)
out = tf.transpose(tf.add(tf.matmul(hidden_4, W_out), bias_out))
```

Step 14 define network architecture

The image below illustrates the network architecture. The model consists of three major building blocks. The input layer, the hidden layers and the output layer. This architecture is called a feed forward network. Feed forward indicates that the batch of data solely flows from left to right. Other network architectures, such as recurrent neural networks, also allow data flowing "backwards" in the network.



Step 15 cost function

The cost function of the network is used to generate a measure of deviation between the network's predictions and the actual observed training targets. For regression problems, the mean squared error (MSE) function is commonly used. MSE computes the average squared deviation between predictions and targets. Basically, any differentiable function can be implemented in order to compute a deviation measure between predictions and targets.

```
# Cost function
mse = tf.reduce_mean(tf.squared_difference(out, Y))
```

Step 16 optimier

The optimizer takes care of the necessary computations that are used to adapt the network's weight and bias variables during training. Those computations invoke the calculation of so called gradients, that indicate the direction in which the weights and biases have to be changed during training in order to minimize the network's cost function.

```
# Optimizer
opt = tf.train.AdamOptimizer().minimize(mse)
```

Here the Adam Optimizer is used, which is one of the current default optimizers in deep learning development. Adam stands for "**Ada**ptive **M**oment Estimation" and can be considered as a combination between two other popular optimizers AdaGrad and RMSProp.

Step 15 fit the model

Fitting the neural network

After having defined the placeholders, variables, initializers, cost functions and optimizers of the network, the model needs to be trained. Usually, this is done by minibatch training. During minibatch training random data samples of `n = batch_size` are drawn from the training data and fed into the network. The training dataset gets divided into `n / batch_size` batches that are sequentially fed into the network. At this point the placeholders `X` and `Y` come into play. They store the input and target data and present them to the network as inputs and targets.

A sampled data batch of `X` flows through the network until it reaches the output layer. There, TensorFlow compares the models predictions against the actual observed targets `Y` in the current batch. Afterwards, TensorFlow conducts an optimization step and updates the networks parameters, corresponding to the selected learning scheme. After having updated the weights and biases, the next batch is sampled and the process repeats itself. The procedure continues until all batches have been presented to the network. One full sweep over all batches is called an epoch.

The training of the network stops once the maximum number of epochs is reached or another stopping criterion defined by the user applies.

```
# Make Session
net = tf.Session()

# Run initializer
net.run(tf.global_variables_initializer())

# Setup interactive plot
plt.ion()
fig = plt.figure()
ax1 = fig.add_subplot(111)
line1, = ax1.plot(y_test)
line2, = ax1.plot(y_test*0.5)
plt.show()

# Number of epochs and batch size
epochs = 10
batch_size = 256
```

```
for e in range(epochs):

    # Shuffle training data
    shuffle_indices = np.random.permutation(np.arange(len(y_train)))
    X_train = X_train[shuffle_indices]
    y_train = y_train[shuffle_indices]

    # Minibatch training
    for i in range(0, len(y_train) // batch_size):
        start = i * batch_size
        batch_x = X_train[start:start + batch_size]
        batch_y = y_train[start:start + batch_size]
        # Run optimizer with batch
        net.run(opt, feed_dict={X: batch_x, Y: batch_y})

        # Show progress
        if np.mod(i, 5) == 0:
            # Prediction
            pred = net.run(out, feed_dict={X: X_test})
            line2.set_ydata(pred)
            plt.title('Epoch ' + str(e) + ', Batch ' + str(i))
            #file_name = 'img/epoch_' + str(e) + '_batch_' + str(i) + '.jpg'
            file_name = 'epoch_' + str(e) + '_batch_' + str(i) + '.jpg'

            #plt.savefig(file_name)
            plt.pause(0.01)

# Print final MSE after Training
mse_final = net.run(mse, feed_dict={X: X_test, Y: y_test})
print(mse_final)
```

During the training, we evaluate the networks predictions on the test set .
Every 5th batch we can visualize the output predictions and optionally save the images a to a disk.

The model quickly learns the shape and location of the time series in the test data and is able to produce an accurate prediction after some epochs.

One can see that the networks rapidly adapts to the basic shape of the time series and continues to learn finer patterns of the data. This also corresponds to the Adam learning scheme that lowers the learning rate during model training in order not to overshoot the optimization minimum. After 10 epochs, we have a pretty close fit to the test data.

The final test MSE equals 0.00078 (it is very low, because the target is scaled). The mean absolute percentage error of the forecast on the test set is equal to 5.31% which is pretty good.

Here is the complete program:

```
# stock_market.py
# SP500 prediction using tensor flow

#Import TensorFlow
#import tensorflow as tf
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler

# import data
df = pd.read_csv('data_stocks.csv')
print (df.head)

# Drop date variable
df = df.drop(['DATE'], axis = 'columns')

# get data
data = df.values
print("data")
print(data)

# scale data
scaler = MinMaxScaler()
scaler.fit(data[:,0:])
data_scaled = scaler.transform(data)

# set target and features
X_data = data_scaled[:,1:]
y_data = data_scaled[:,0]
```

```
print("scaled X")
print(X_data)

print("scaled Y")
print(y_data)

# split data set %80 train %20% test do not shuffle
X_train, X_test, y_train, y_test = train_test_split(X_data, y_data, test_size=0.20, shuffle=False)

# Model architecture parameters
n_stocks = 500
n_neurons_1 = 1024
n_neurons_2 = 512
n_neurons_3 = 256
n_neurons_4 = 128
n_target = 1

# Placeholder
X = tf.placeholder(dtype=tf.float32, shape=[None, n_stocks])
Y = tf.placeholder(dtype=tf.float32, shape=[None])

# Initializers
sigma = 1
weight_initializer = tf.variance_scaling_initializer(mode="fan_avg", distribution="uniform",
scale=sigma)
bias_initializer = tf.zeros_initializer()

# Layer 1: Variables for hidden weights and biases
W_hidden_1 = tf.Variable(weight_initializer([n_stocks, n_neurons_1]))
bias_hidden_1 = tf.Variable(bias_initializer([n_neurons_1]))

# Layer 2: Variables for hidden weights and biases
W_hidden_2 = tf.Variable(weight_initializer([n_neurons_1, n_neurons_2]))
bias_hidden_2 = tf.Variable(bias_initializer([n_neurons_2]))

# Layer 3: Variables for hidden weights and biases
W_hidden_3 = tf.Variable(weight_initializer([n_neurons_2, n_neurons_3]))
bias_hidden_3 = tf.Variable(bias_initializer([n_neurons_3]))

# Layer 4: Variables for hidden weights and biases
W_hidden_4 = tf.Variable(weight_initializer([n_neurons_3, n_neurons_4]))
bias_hidden_4 = tf.Variable(bias_initializer([n_neurons_4]))
```

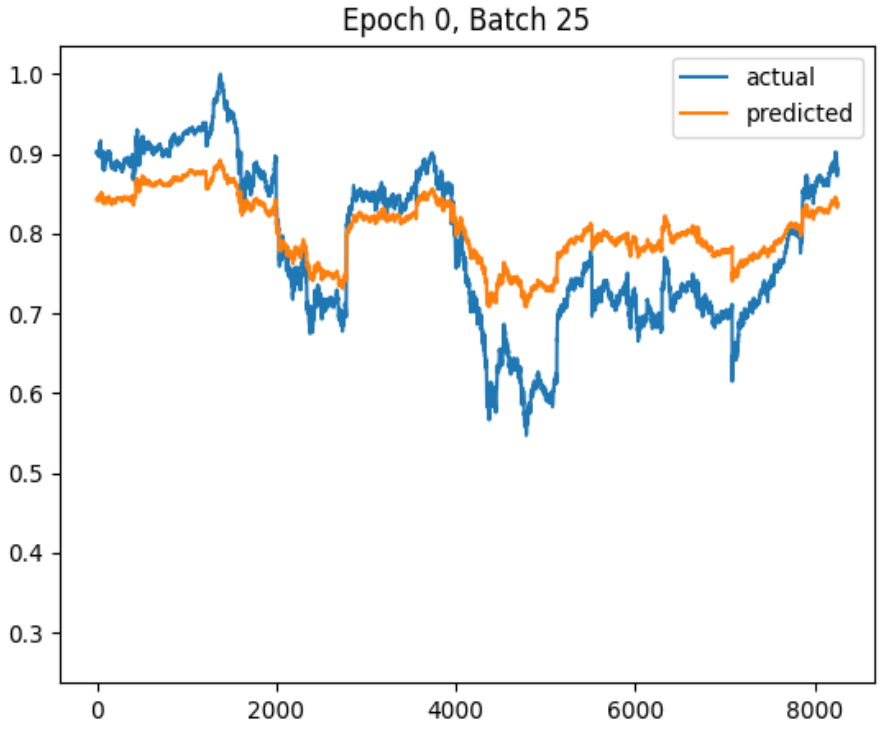
```
# Output layer: Variables for output weights and biases
W_out = tf.Variable(weight_initializer([n_neurons_4, n_target]))
bias_out = tf.Variable(bias_initializer([n_target]))

# Hidden layer
hidden_1 = tf.nn.relu(tf.add(tf.matmul(X, W_hidden_1), bias_hidden_1))
hidden_2 = tf.nn.relu(tf.add(tf.matmul(hidden_1, W_hidden_2), bias_hidden_2))
hidden_3 = tf.nn.relu(tf.add(tf.matmul(hidden_2, W_hidden_3), bias_hidden_3))
hidden_4 = tf.nn.relu(tf.add(tf.matmul(hidden_3, W_hidden_4), bias_hidden_4))

# Output layer (must be transposed)
out = tf.transpose(tf.add(tf.matmul(hidden_4, W_out), bias_out))

# Cost function
mse = tf.reduce_mean(tf.squared_difference(out, Y))

# Optimizer
opt = tf.train.AdamOptimizer().minimize(mse)

# Make Session
net = tf.Session()

# Run initializer
net.run(tf.global_variables_initializer())

# Setup interactive plot
plt.ion()
fig = plt.figure()
ax1 = fig.add_subplot(111)
line1, = ax1.plot(y_test,label="actual")
line2, = ax1.plot(y_test*0.5,label="predicted")
plt.legend()
plt.show()

# Number of epochs and batch size
epochs = 10
batch_size = 256

# for each epochs
for e in range(epochs):

    # Shuffle training data
    shuffle_indices = np.random.permutation(np.arange(len(y_train)))
```

```
    X_train = X_train[shuffle_indices]
    y_train = y_train[shuffle_indices]

    # Minibatch training
    for i in range(0,len(y_train) // batch_size):
        start = i * batch_size
        batch_x = X_train[start:start + batch_size]
        batch_y = y_train[start:start + batch_size]

        # Run optimizer with batch
        net.run(opt, feed_dict={X: batch_x, Y: batch_y})

        # Show progress
        if np.mod(i, 5) == 0:
            # Prediction
            pred = net.run(out, feed_dict={X: X_test})
            line2.set_ydata(pred)
            plt.title('Epoch ' + str(e) + ', Batch ' + str(i))
            #file_name = 'img/epoch_' + str(e) + '_batch_' + str(i) + '.jpg'
            #file_name = 'epoch_' + str(e) + '_batch_' + str(i) + '.jpg'

            #plt.savefig(file_name)
            plt.pause(0.01)

# Print final MSE after Training
mse_final = net.run(mse, feed_dict={X: X_test, Y: y_test})
print(mse_final)
```

Todo:

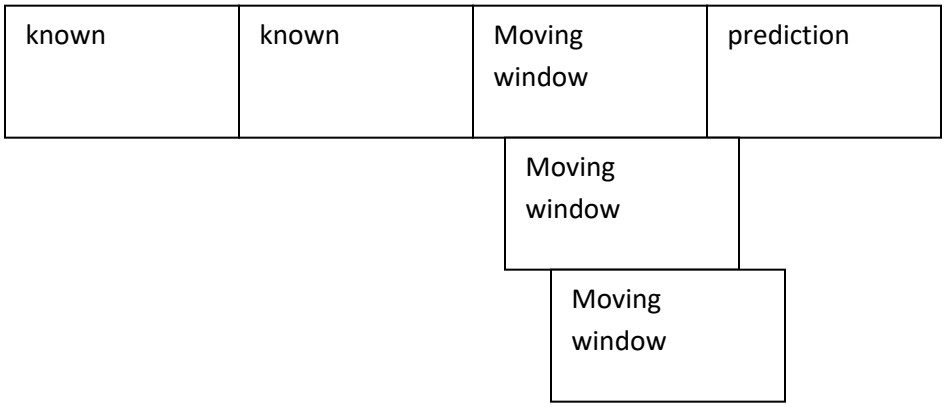
Type in or copy and paste and run the program. Try different split ratios, Apply the scalar separately on the train data's and test data.

You should get something like this:

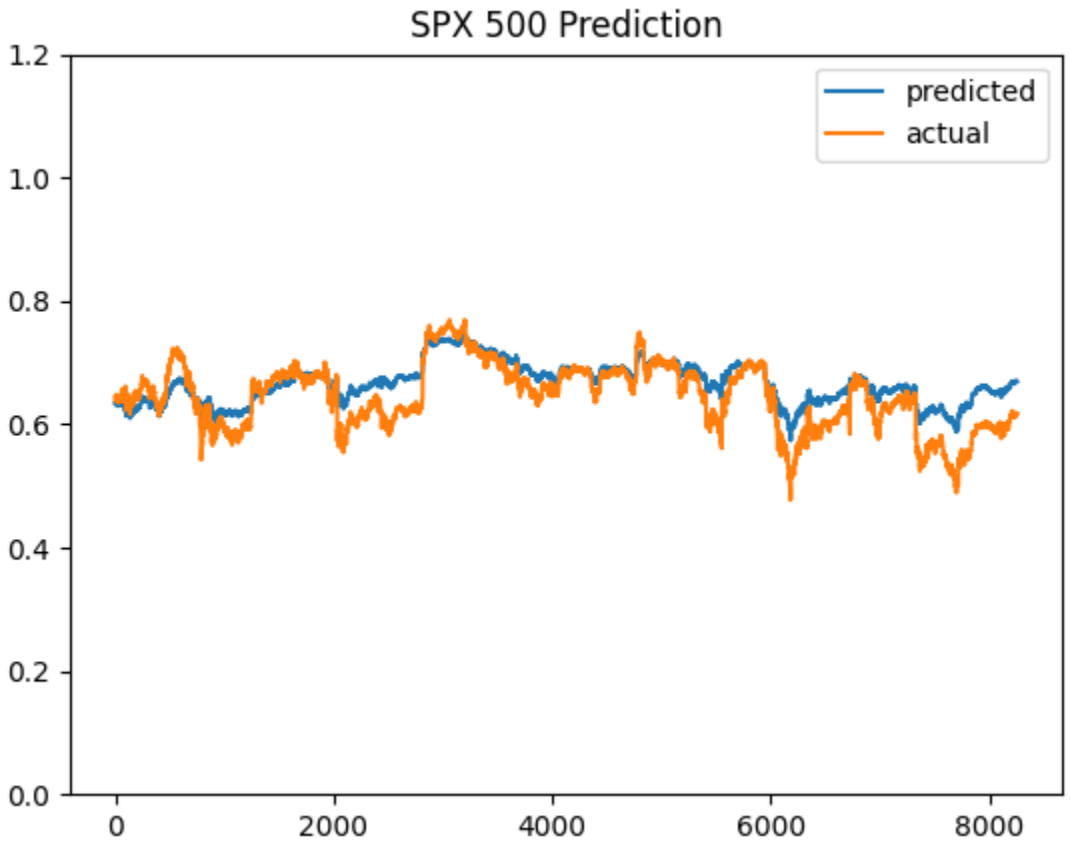


DeepLearning Homework 1

Use the above program to buy and sell on the SPX 500 index. You are not buying and selling individual stocks just the SPX500 index . You have to use a moving window to predict the next SPX 500 index, since the tensor-flow program predicts from the training size, not individual rows. Spit the SPX500 data into 2 sections, known and prediction. In this situation the predicted SPX 500 index is already known. we can use this known part to compare predicted data to actual data.



You can experiment with the width. Try to use the width that would give you the most profit. You can make profit by buying and selling. This is known as long and short trades. For a long trade you start from a low index level and when it reached a higher level you cash out. For short trades start at a high level and cash out at a low level. You still make money on the index absoluter difference. When your prediction says the stocks will go up then buy. If your prediction says you stocks will go down then cash out. You van also sell when the stock goes down, in this situation when the index starts to go cash out. Keep track how much money you make and loose. You can make markers for buying and selling. In another array and plot has a scatter plot.



END