STOCK PRICE PREDICTION DOCUMENTATION

Table of Contents:

1. Introduction

2. Data Collection and Preprocessing

3. Recurrent Neural Networks

3.1 Basic RNNs

3.1.1 Advantages and Disadvantages

3.2 Long Short-Term Memory Networks(LSTMs)

3.2.1 Advantages and Disadvantages

3.3 Gated Recurrent Unit Networks (GRUs)

3.3.1 Advantages and Disadvantages

4. Model Training

5. Model Prediction

6. Conclusion

7. References

Introduction:

Stock price prediction is a critical task in financial markets. It involves forecasting the future price movements of stocks based on historical data and other related information. This documentation explores using Recurrent Neural Networks (RNNs) and their variants—LSTM and GRU—for stock price prediction.

Data Collection and Preprocessing:

Importing Libraries:

```
import numpy as np

import pandas as pd

import math

import sklearn

import sklearn.preprocessing

import datetime

import os

import matplotlib.pyplot as plt

import tensorflow as tf


# split data in 80%/10%/10% train/validation/test sets

valid_set_size_percentage = 10

test_set_size_percentage = 10


#display parent directory and working directory
```

```
print(os.path.dirname(os.getcwd())+':', os.listdir(os.path.dirname(os.getcwd())));
print(os.getcwd()+':', os.listdir(os.getcwd()));
/kaggle: ['src', 'lib', 'working', 'input']
/kaggle/working: ['script.ipynb', '__output__.json']
```

Analyze data:

load stock prices from prices-split-adjusted.csv

analyze data

```
# import all stock prices
df = pd.read_csv("../input/prices-split-adjusted.csv", index_col = 0)
df.info()
df.head()

# number of different stocks
print('\nnumber of different stocks: ', len(list(set(df.symbol))))
print(list(set(df.symbol))[:10])
df.tail()
df.info()
plt.figure(figsize=(15, 5));
plt.subplot(1,2,1);
plt.plot(df[df.symbol == 'EQIX'].open.values, color='red', label='open')
plt.plot(df[df.symbol == 'EQIX'].close.values, color='green', label='close')
plt.plot(df[df.symbol == 'EQIX'].low.values, color='blue', label='low')
plt.plot(df[df.symbol == 'EQIX'].high.values, color='black', label='high')
plt.title('stock price')
```

```
plt.xlabel('time [days]')
plt.ylabel('price')
plt.legend(loc='best')
#plt.show()

plt.subplot(1,2,2);
plt.plot(df[df.symbol == 'EQIX'].volume.values, color='black', label='volume')
plt.title('stock volume')
plt.xlabel('time [days]')
plt.ylabel('volume')
plt.legend(loc='best');
```

<u>Manipulate data:</u>

choose a specific stock

drop feature: volume

normalize stock data

create train, validation and test data sets

```
# function for min-max normalization of stock
def normalize_data(df):
    min_max_scaler = sklearn.preprocessing.MinMaxScaler()
    df['open'] = min_max_scaler.fit_transform(df.open.values.reshape(-1,1))
    df['high'] = min_max_scaler.fit_transform(df.high.values.reshape(-1,1))
    df['low'] = min_max_scaler.fit_transform(df.low.values.reshape(-1,1))
    df['close'] = min_max_scaler.fit_transform(df['close'].values.reshape(-1,1))
    return df
```

```
# function to create train, validation, test data given stock data and sequence length
def load_data(stock, seq_len):
    data_raw = stock.as_matrix() # convert to numpy array
    data = []

    # create all possible sequences of length seq_len
    for index in range(len(data_raw) - seq_len):
        data.append(data_raw[index: index + seq_len])

    data = np.array(data);
    valid_set_size = int(np.round(valid_set_size_percentage/100*data.shape[0]));
    test_set_size = int(np.round(test_set_size_percentage/100*data.shape[0]));
    train_set_size = data.shape[0] - (valid_set_size + test_set_size);

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

    x_valid = data[train_set_size:train_set_size+valid_set_size,:-1,:]
    y_valid = data[train_set_size:train_set_size+valid_set_size,-1,:]

    x_test = data[train_set_size+valid_set_size:,:-1,:]
    y_test = data[train_set_size+valid_set_size:,-1,:]

    return [x_train, y_train, x_valid, y_valid, x_test, y_test]
```

```
# choose one stock
df_stock = df[df.symbol == 'EQIX'].copy()
df_stock.drop(['symbol'],1,inplace=True)
df_stock.drop(['volume'],1,inplace=True)

cols = list(df_stock.columns.values)
print('df_stock.columns.values = ', cols)

# normalize stock
df_stock_norm = df_stock.copy()
df_stock_norm = normalize_data(df_stock_norm)

# create train, test data
seq_len = 20 # choose sequence length
x_train, y_train, x_valid, y_valid, x_test, y_test = load_data(df_stock_norm,
seq_len)
print('x_train.shape = ',x_train.shape)
print('y_train.shape = ', y_train.shape)
print('x_valid.shape = ',x_valid.shape)
print('y_valid.shape = ', y_valid.shape)
print('x_test.shape = ', x_test.shape)
print('y_test.shape = ',y_test.shape)
df_stock.columns.values =  ['open', 'close', 'low', 'high']
x_train.shape =  (1394, 19, 4)
y_train.shape =  (1394, 4)
```

x_valid.shape = (174, 19, 4)

y_valid.shape = (174, 4)

x_test.shape = (174, 19, 4)

y_test.shape = (174, 4)

linkcode

```
plt.figure(figsize=(15, 5));
plt.plot(df_stock_norm.open.values, color='red', label='open')
plt.plot(df_stock_norm.close.values, color='green', label='low')
plt.plot(df_stock_norm.low.values, color='blue', label='low')
plt.plot(df_stock_norm.high.values, color='black', label='high')
#plt.plot(df_stock_norm.volume.values, color='gray', label='volume')
plt.title('stock')
plt.xlabel('time [days]')
plt.ylabel('normalized price/volume')
plt.legend(loc='best')
plt.show()
```

Recurrent Neural Networks (RNNs):

3.1 Basic RNNs:

3.1.1 Advantages and Disadvantages

Advantages: Captures temporal dependencies, simple architecture

Disadvantages: Vanishing gradient problem, short-term memory

3.2 Long Short-Term Memory Networks (LSTMs):

3.2.1 Advantages and Disadvantages

Advantages: Solves vanishing gradient problem, good for long-term dependencies

Disadvantages: Computationally expensive, complex architecture

3.3 Gated Recurrent Unit Networks (GRUs):

3.3.1 Advantages and Disadvantages

Advantages: Fewer parameters than LSTM, efficient for some tasks

Disadvantages: Less interpretability than LSTM, may not capture as long dependencies as LSTM

Model Training:

Model training for stock price prediction using RNNs involves several critical steps to ensure effective learning and prediction capabilities. Initially, the model's weights are set, typically using methods like random initialization or more advanced techniques such as Xavier initialization to balance input and output variances. Training data is fed sequentially into the network, and predictions are generated through the forward pass, capturing temporal patterns and dependencies. The backward pass, employing backpropagation through time (BPTT), computes gradients of the loss function—commonly Mean Squared Error (MSE) or Mean Absolute Error (MAE)—with respect to the model's weights.

These gradients guide the optimization algorithm, such as Adam or RMSprop, to update the weights iteratively, minimizing the loss. Hyperparameters, including learning rate, batch size, number of layers, and hidden units, are finely tuned through experimentation and validation to enhance model performance and generalization. The training process continues until convergence, often monitored through a combination of validation metrics and early stopping criteria to prevent overfitting.

RNNs with basic, LSTM, GRU cells

Code:

```
## Basic Cell RNN in tensorflow


index_in_epoch = 0;

perm_array  = np.arange(x_train.shape[0])

np.random.shuffle(perm_array)


# function to get the next batch
```

```
def get_next_batch(batch_size):
    global index_in_epoch, x_train, perm_array
    start = index_in_epoch
    index_in_epoch += batch_size

    if index_in_epoch > x_train.shape[0]:
        np.random.shuffle(perm_array) # shuffle permutation array
        start = 0 # start next epoch
        index_in_epoch = batch_size

    end = index_in_epoch
    return x_train[perm_array[start:end]], y_train[perm_array[start:end]]

# parameters
n_steps = seq_len-1
n_inputs = 4
n_neurons = 200
n_outputs = 4
n_layers = 2
learning_rate = 0.001
batch_size = 50
n_epochs = 100
train_set_size = x_train.shape[0]
test_set_size = x_test.shape[0]

tf.reset_default_graph()
```

```
X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
y = tf.placeholder(tf.float32, [None, n_outputs])

# use Basic RNN Cell
layers = [tf.contrib.rnn.BasicRNNCell(num_units=n_neurons,
activation=tf.nn.elu)
          for layer in range(n_layers)]

# use Basic LSTM Cell
#layers = [tf.contrib.rnn.BasicLSTMCell(num_units=n_neurons,
activation=tf.nn.elu)
#         for layer in range(n_layers)]

# use LSTM Cell with peephole connections
#layers = [tf.contrib.rnn.LSTMCell(num_units=n_neurons,
#                              activation=tf.nn.leaky_relu, use_peepholes = True)
#         for layer in range(n_layers)]

# use GRU cell
#layers = [tf.contrib.rnn.GRUCell(num_units=n_neurons,
activation=tf.nn.leaky_relu)
#         for layer in range(n_layers)]

multi_layer_cell = tf.contrib.rnn.MultiRNNCell(layers)
rnn_outputs, states = tf.nn.dynamic_rnn(multi_layer_cell, X, dtype=tf.float32)
```

```
stacked_rnn_outputs = tf.reshape(rnn_outputs, [-1, n_neurons])
stacked_outputs = tf.layers.dense(stacked_rnn_outputs, n_outputs)
outputs = tf.reshape(stacked_outputs, [-1, n_steps, n_outputs])
outputs = outputs[:,n_steps-1,:] # keep only last output of sequence

loss = tf.reduce_mean(tf.square(outputs - y)) # loss function = mean squared
error
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(loss)

# run graph
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for iteration in range(int(n_epochs*train_set_size/batch_size)):
        x_batch, y_batch = get_next_batch(batch_size) # fetch the next training
batch
        sess.run(training_op, feed_dict={X: x_batch, y: y_batch})
        if iteration % int(5*train_set_size/batch_size) == 0:
            mse_train = loss.eval(feed_dict={X: x_train, y: y_train})
            mse_valid = loss.eval(feed_dict={X: x_valid, y: y_valid})
            print('%.2f epochs: MSE train/valid = %.6f/%.6f'%(
                iteration*batch_size/train_set_size, mse_train, mse_valid))

    y_train_pred = sess.run(outputs, feed_dict={X: x_train})
    y_valid_pred = sess.run(outputs, feed_dict={X: x_valid})
    y_test_pred = sess.run(outputs, feed_dict={X: x_test})
```

Model Prediction:

The prediction phase in stock price forecasting with Recurrent Neural Networks (RNNs) involves utilizing the trained model to generate future stock price estimates based on historical input sequences. RNNs, particularly LSTMs and GRUs, are well-suited for this task as they can learn temporal dependencies and patterns in sequential data, capturing long-term trends and short-term fluctuations. During prediction, the model processes input features through its recurrent structure, using its internal state to incorporate information from previous time steps and generate a forecast for the next time step. This iterative process allows the model to predict stock prices over multiple future intervals, providing a sequential forecast that can be used for decision-making in trading or investment strategies. Accurate model predictions depend on well-prepared data, robust training, and proper handling of temporal dynamics inherent in financial time series.

Code:

```
y_train.shape


ft = 0 # 0 = open, 1 = close, 2 = highest, 3 = lowest


## show predictions
plt.figure(figsize=(15, 5));
plt.subplot(1,2,1);


plt.plot(np.arange(y_train.shape[0]), y_train[:,ft], color='blue', label='train
target')


plt.plot(np.arange(y_train.shape[0], y_train.shape[0]+y_valid.shape[0]),
y_valid[:,ft],
```

```
        color='gray', label='valid target')

plt.plot(np.arange(y_train.shape[0]+y_valid.shape[0],
                   y_train.shape[0]+y_test.shape[0]+y_test.shape[0]),
        y_test[:,ft], color='black', label='test target')

plt.plot(np.arange(y_train_pred.shape[0]),y_train_pred[:,ft], color='red',
        label='train prediction')

plt.plot(np.arange(y_train_pred.shape[0],
y_train_pred.shape[0]+y_valid_pred.shape[0]),
        y_valid_pred[:,ft], color='orange', label='valid prediction')

plt.plot(np.arange(y_train_pred.shape[0]+y_valid_pred.shape[0],

y_train_pred.shape[0]+y_valid_pred.shape[0]+y_test_pred.shape[0]),
        y_test_pred[:,ft], color='green', label='test prediction')
plt.title('past and future stock prices')
plt.xlabel('time [days]')
plt.ylabel('normalized price')
plt.legend(loc='best');
plt.subplot(1,2,2);
plt.plot(np.arange(y_train.shape[0], y_train.shape[0]+y_test.shape[0]),
        y_test[:,ft], color='black', label='test target')
plt.plot(np.arange(y_train_pred.shape[0],
y_train_pred.shape[0]+y_test_pred.shape[0]),
        y_test_pred[:,ft], color='green', label='test prediction')
```

```
plt.title('future stock prices')

plt.xlabel('time [days]')

plt.ylabel('normalized price')

plt.legend(loc='best');


corr_price_development_train = np.sum(np.equal(np.sign(y_train[:,1]-
y_train[:,0]),

            np.sign(y_train_pred[:,1]-y_train_pred[:,0])).astype(int)) /
y_train.shape[0]

corr_price_development_valid = np.sum(np.equal(np.sign(y_valid[:,1]-
y_valid[:,0]),

            np.sign(y_valid_pred[:,1]-y_valid_pred[:,0])).astype(int)) /
y_valid.shape[0]

corr_price_development_test = np.sum(np.equal(np.sign(y_test[:,1]-y_test[:,0]),

            np.sign(y_test_pred[:,1]-y_test_pred[:,0])).astype(int)) / y_test.shape[0]


print('correct sign prediction for close - open price for train/valid/test:
%.2f/%.2f/%.2f'%(

    corr_price_development_train, corr_price_development_valid,
corr_price_development_test))
```

Conclusion:

Recurrent Neural Networks (RNNs), including LSTM and GRU, offer powerful tools for stock price prediction by capturing temporal dependencies in financial data. Proper preprocessing, model selection, and tuning are crucial for effective predictions. While RNNs have advantages, their complexity and resource requirements must be managed.

References:

 Graves, A., Mohamed, A.-R., & Hinton, G. (2013). Speech recognition with deep recurrent neural networks. In 2013 IEEE International Conference on Acoustics, Speech and Signal Processing.

 Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. Neural Computation, 9(8), 1735-1780.