Group member contributions:

	Task 1		Task 2		Task 3	
	Code	Report	Code	Report	Code	Report
He Yunfan	80%	20%	33%	40%	33%	50%
Liu Ziwei	/	80%	33%	30%	33%	50%
Chen Jisen	20%	/	33%	30%	33%	/

Project Report_Task 1

1. Abstract

In Task 1, we tried two methods to predict the value of label, which is the difference between future n-th midPrice and current midPrice. This value is also equal to (the future n-th tick's AskPrice1 +the future n-th tick's BidPrice1- the current tick's AskPrice1 -the current tick's BidPrice1)/2 .

For method 1, we first split and disorganize the order of the data sets and then use regression method to predict the label. However, the loss the quite large and the model we trained is not that ideal.

For method 2, we first use normalization and then split and shuffle the data set before using LTSM to predict the future result with sequence. It turns out that the loss we gain from our LSTM method is quite small and the model we trained is beyond our satisfactory.

Therefore, we will choose Method2---LSTM as our final method to construct our model to predict the difference of midPrice.

2. Method 1_Regression Model

2.1 Environment Configuration

We use colab as our running platform. It is easy to use and it has good computation ability.

2.2 Procedure

We use keras as the whole foundation and construct our training and testing model based on keras. In Method_1, We divide all our procedure into 4 parts:

1) Load the dataset
2) Split test and train set
3) Model and training
4) Question we faced

In the following parts, we will detailedly express what we have done in these four parts and the obstacles we faced.

2.2.1 load the dataset

We import the dataset provided by teacher assistant into our construction so that we can use this dataset to train and test our network. The code we use to load dataset is posted below.

Task 1: Regression based stock price prediction

1. Load the dataset

```
[ ]  # Install the PyDrive wrapper & import libraries.
     # This only needs to be done once per notebook.
     !pip install -U -q PyDrive
     from pydrive.auth import GoogleAuth
     from pydrive.drive import GoogleDrive
     from google.colab import auth
     from oauth2client.client import GoogleCredentials
     import os
     import zipfile
     import pandas as pd

     # Authenticate and create the PyDrive client.
     # This only needs to be done once per notebook.
     auth.authenticate_user()
     gauth = GoogleAuth()
     gauth.credentials = GoogleCredentials.get_application_default()
     drive = GoogleDrive(gauth)

     # Download a file based on its file ID.
     #
     # A file ID looks like: laggVyWshwcyP6kEI-y_W3P8D26sz
     file_id = '1RayMjAmnOPhXMOI-bypjFBwU2RtgDwSw'
     downloaded = drive.CreateFile({'id': file_id})

     downloaded.GetContentFile('data.zip')


     path = os.listdir('.')
     print (path)


     with zipfile.ZipFile("data.zip","r") as datazip:
       datazip.extractall("")


     data = pd.read_csv("data.csv")
```

```
['.config', 'data.zip', 'data.csv', 'adc.json', 'sample_data']
```

2.2.2 split test and train set

First, we use train_test_split function which is keras-provided function and import it into our model to split and disorganize our training dataset to train our model in a more efficient way.

```
from sklearn.model_selection import train_test_split
```

Then, for the setting of split, we set the test size to be 0.9, which means we set the ratio of train data and test data to be 1:9 for a quicker training and set the future_n to be 10 for convenience. In the following adjusting, we can easily change the value of test_size and future_n to change the train-test ratio and the time interval between the existing dataset and our predicted midPrice.

```
test_size=0.9
future_n=10;
```

Besides, we use midPrice_array to represent the dataset only containing the information of midPrice of different moment to calculate the label of the existing training dataset. In the formula of calculating label, we use “midPrice_array[future_n:]” and “midPrice_array[:-future_n]” to separately represent the array without the first n-order data in midPrice_array and the array without the last n-order data in midPrice_array. Therefore, their difference is the label---the known midPrice after n ticks in our training dataset.

```
midPrice_array=np.array(data['midPrice'])
# midPrice_n_tick=midPrice_array[future_n:]
# midPrice_z_tick=midPrice_array[:-future_n]
label=midPrice_array[future_n:]-midPrice_array[:-future_n] #may need resize
```

After that, before using 108 indicator as our feature of regression model, we need to drop some features: ‘AskPrice1’, ‘BidPrice1’, ‘midPrice’, ‘AskVolume1’, ’BidVolume1’, ‘UpdataTime’ and ‘UpdateMillisec’. Since ‘AskPrice1’ and ‘BidPrice1’ are used to calculate ‘midPrice’ and ‘midPrice’ is our label, we need to delete them from feature. Besides, since in the dataset, we use order to rank the dataset, ignoring the concept about time, so we also omit the feature of ‘UpdateTime’ and ‘UpdataMillisec’ which are concerned with time.

```
input_raw=data.drop(['midPrice','UpdateTime', 'UpdateMillisec', 'AskPrice1','BidPrice1','AskVolume1','BidVolume1'],axis=1)
```

Finally, use input_unsplit as input training dataset and use ‘train_test_split’ function to seperately calculate the train and test valuse of input X and output Y.

```
input_unsplit=np.array(input_raw)[:-future_n]

X_train,X_test, y_train, y_test =train_test_split(input_unsplit,label,test_size=test_size, random_state=0)
```

The full code of split test and train set is attached as below:

▼ **2. Split test and train set**

We first want to use train_test_split to split the dataset.
Test method: standard variance, and plot the graphs of expected one and real one.

Feature used: indicator 1-108, ...
Feature not used: 'AskPrice1', 'BidPrice1', 'midPrice'
Label: (the future n-th tick's AskPrice1 + the future n-th tick's BidPrice1 - the current tick's AskPrice1 - the current tick's BidPrice1) / 2, the mid price

```
[45]  from sklearn.model_selection import train_test_split
      import numpy as np
      n_samples=len(data)
      test_size=0.9
      future_n=10;
      midPrice_array=np.array(data['midPrice'])
      # midPrice_n_tick=midPrice_array[future_n:]
      # midPrice_z_tick=midPrice_array[:-future_n]
      label=midPrice_array[future_n:]-midPrice_array[:-future_n] #may need resize
      input_raw=data.drop(['midPrice','UpdateTime', 'UpdateMillisec', 'AskPrice1','BidPrice1','AskVolume1','BidVolume1'],axis=
      input_unsplit=np.array(input_raw)[:-future_n]

      X_train,X_test, y_train, y_test =train_test_split(input_unsplit,label,test_size=test_size, random_state=0)
```

2.2.3 Model and training

As foundation, we use keras to construct the whole regression model and assemble the construction of neural network. Therefore, we input the following module to effectively construct the neutral network. Since we use keras as basis, we need only construct each layer's model and keras will automatically compose them.

```
from keras import models
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import Activation
import numpy as np
import matplotlib.pyplot as plt
```

The summary table of our neutral network is created by using model.summary() function, and our model is more clearly shown in this table with the use of several dense layers, dropout layers and activation functions.

Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 64)	8512
dropout_1 (Dropout)	(None, 64)	0
activation_1 (Activation)	(None, 64)	0
dense_2 (Dense)	(None, 32)	2080
activation_2 (Activation)	(None, 32)	0
dense_3 (Dense)	(None, 32)	1056
dense_4 (Dense)	(None, 1)	33

Total params: 11,681
Trainable params: 11,681
Non-trainable params: 0

1) . Dense layer is called densely connected neural network layer, which is used to import a nonlinear change to the input features and output the relation between these features. Here since we want only output one predicted value, we use several dense layers to decrease the size of x to size 1. For these dense layer, we all use 'ReLU' function as our non-linear activation function in dense layer.

```
model.add(Dense(64, input_dim=input_dim, activation='relu'))
```

```
model.add(Dense(32, activation='relu'))
```

```
model.add(Dense(1))
```

2) . Dropout layer is used to prevent over-fitting. With the use of dropout layer we can drop some output of last layer to get a set of more accurate inputs. Here we use the ratio to be 0.2, which means we only keep 20% data of last layer's output.

```
model.add(Dropout(0.2))
```

3) . For seperate activation layer here, we use linear activation.

```
model.add(Activation("linear"))
```

The detailed code of constructing our model is attached as below:

▼ **3. training**

We use the simplest regression model of keras.
The model summary is below.

```
[ ]  from keras import models
     from keras.layers import Dense
     from keras.layers import Dropout
     from keras.layers import Activation
     import numpy as np
     import matplotlib.pyplot as plt

     input_dim=X_train[0].size

     model = models.Sequential()

     model.add(Dense(64, input_dim=input_dim, activation='relu'))

     model.add(Dropout(0.2))

     model.add(Activation("linear"))

     model.add(Dense(32, activation='relu'))

     model.add(Activation("linear"))

     model.add(Dense(32, activation='relu'))

     model.add(Dense(1))

     model.compile(loss='mean_squared_error', optimizer="adam", metrics=["accuracy"])

     model.summary()
```

2.2.4. Question in Solution

Since the loss is a bit larger than we expect, we may need to complete our model to decrease its training time and loss. But before revising this model, we use LSTM model to see the effects of both two models, and choose the better one to revise. It turns out that using LSTM is a better way.

2.3 Test Results and Evaluation

2.3.1. Test Result

In construction part, we use model.compile(loss='mae', optimizer='adam', metrics=[“accuracy”]) to output the loss and training time. Unfortunately, The loss is quit large. The test result is not that ideal and it seems that there is something not efficient in our model.

2.3.2. Evaluation

```
model.compile(loss='mean_squared_error', optimizer="adam", metrics=["accuracy"])
```

Since our perdition problem is not a classification problem, so we are unable to use tanh method as activation to judge whether our predicted value tends to correct(1) or wrong(-1).

However, in evaluation part, we do come up with several new ideas to better evaluate our result and can apply these method into following project.

First is plotting the figure our predicted difference of MidPrice and compare the ground truth figure and the test figure.

Second is setting the upper and lower bound for each to-be-predicted value, maybe using standard deviation or other measurements to set a range for each value. If our predicted value is in this range, then we can regard this test result to be accurate.

```
# predict 1000 value for testing
predict_number=1000
Y_esitmated=np.array([])
for i in range(predict_number):
  Y_esitmated=np.append(Y_esitmated,model.predict(X_test[i]))

fig=plt.figure()
plt.plot(range(predict_number),Y_esitmated)
plt.plot(range(predict_number),y_test[0:1000])
plt.show()
```

However, since the model trained by this method is quite far away from our expectation, we decide to use LSTM to find whether there exists a better model, and it does work, which we mentioned below.

3. Method 2_LSTM

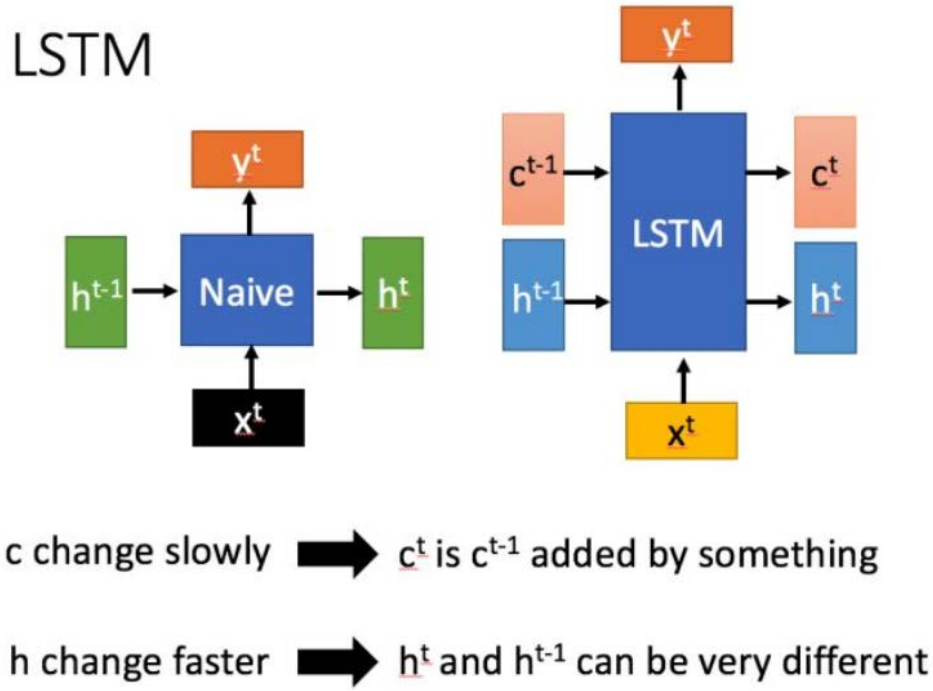
3.1 Environment Configuration

xxxxxxx

3.2 Foundation of LSTM structure

LSTM, which is also called Long short-term memory, is an artificial recurrent neural network (RNN) used to solve the problem of gradient vanish and gradient explosion. LSTM network is really well-suited to “classifying, processing and making prediction based on time series data” and that’s why we tried to use LSTM to solve our prediction which is also based on time sequence (wiki LSTM).

The following figure greatly explains the difference between ordinary RNN network and LSTM network, where RNN has only one transmission state h^t (hidden state) while LSTM has two transmission states: c^t (cell state) and h^t (hidden state).



For inner structure of LSTM, It can be divided into four states and relative three steps in the whole model.

3.2.1. Four states

For the four states, they can be represented as **z, z^i, z^f ,z^o**. These four values are created by first combining the vector of x^t and h^(t-1), then multiplying it with weight and finally applying activation function to translate the result into a value between 0 and 1 (for z^i, z^f and z^0) or a value between -1 and 1 (for z), which can be more clearly mentioned below.

$$z = tanh\left(W \begin{bmatrix} x^t \\ h^{t-1} \end{bmatrix}\right)$$

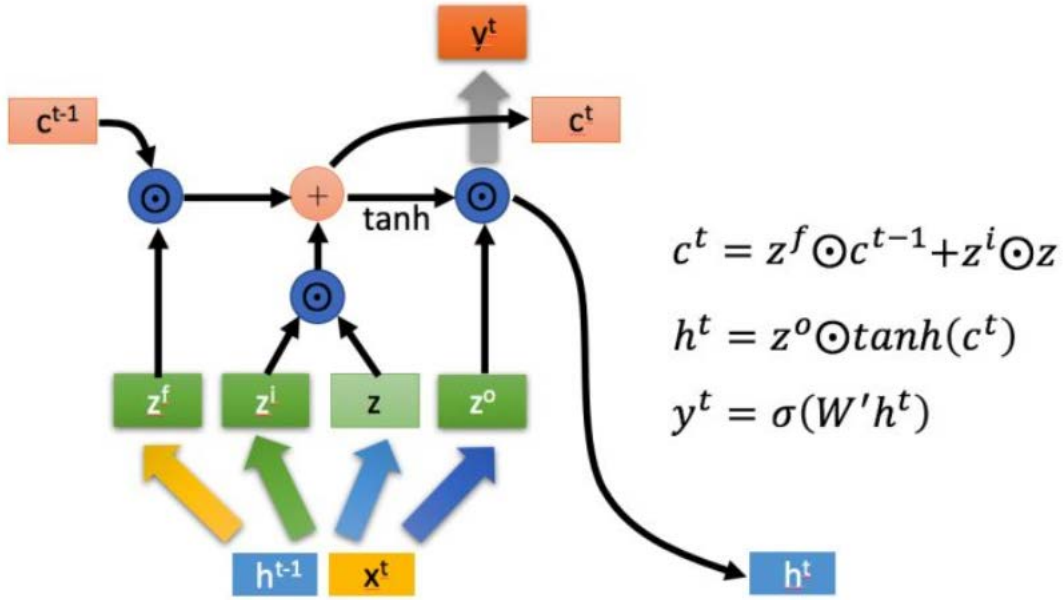
$$z^i = \sigma\left(W^i \begin{bmatrix} x^t \\ h^{t-1} \end{bmatrix}\right)$$

$$z^f = \sigma\left(W^f \begin{bmatrix} x^t \\ h^{t-1} \end{bmatrix}\right)$$

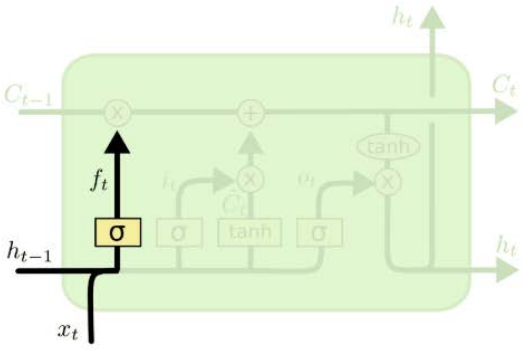
$$z^o = \sigma\left(W^o \begin{bmatrix} x^t \\ h^{t-1} \end{bmatrix}\right)$$

3.2.2 Three states

For the three steps, they can be classified as **forget stage, selective memory stage and output stage**.



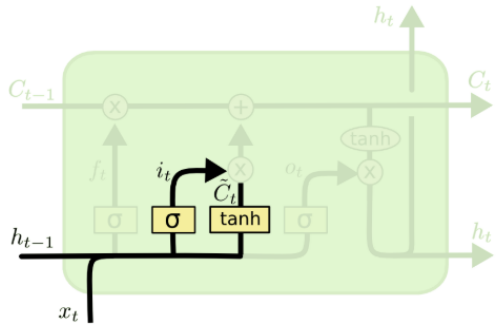
For the **first step called "forget gate layer"** , it is used to decide what information we are going to throw away from the cell state. The value of z^f is gained in the step under control of h^{t-1} and x^t and is used as an output of a number between 0 and 1 to control the cell state C^{t-1}**.** A 1 represents "completely keep this" while a 0 represents "completely get rid of this"(zhihu). The procedure of this step is mentioned as below, where ft represent actually z^f .



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

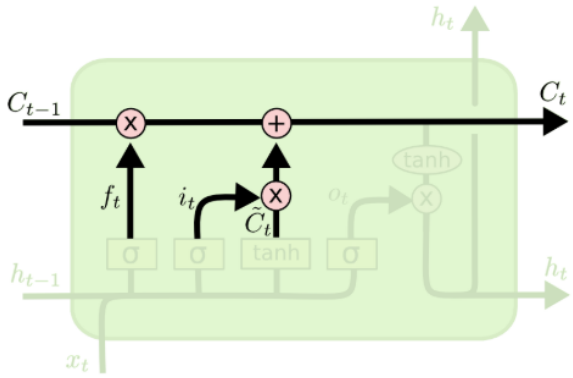
For **second step called "input gate layer"**, it is used to decide what new information we're going to store in the cell state. This step can be divided into three parts, where in the first part, we use sigmoid function as activation function to get an output z^i between 0 and 1 to control the cell

state C^t , in the second part, we update the old cell state C^{t-1} into a new cell state represented by **z** with activation function of tanh, and in the final part, we combine these above two parts together to add new data information which we want to update in addition to forget step. The procedure of this step is mentioned as below, where i_t represents actually z^i and $\tilde{C}_t$ represents actually z.



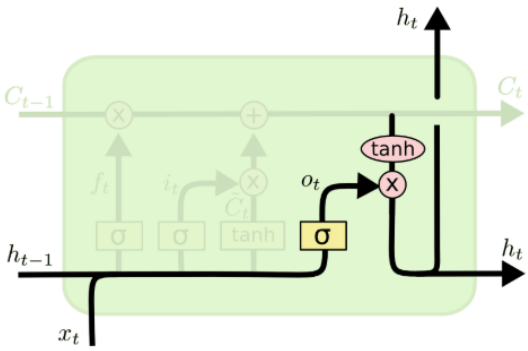
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Combining step1 and step2 we can drop the information about the old subject's gender and add the new information to get new cell state value C^t**.**



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

For **final step called "output gate layer",** it is used to decide what information we're going to use as output. z^o is also used as an output of a number between 0 and 1 to control the output hidden state**.** Besides, as in step2, we can use tanh as activation function to get the value of next hidden state h^t as output**.** The procedure of this step is mentioned as below, where o_t represents actually z^o .



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

3.3 Procedure

We use keras as the whole foundation and construct our training and testing model based on keras. In Method_2, We also divide our procedure into 4 parts:

1) Load the dataset
2) Normalization
3) Split and build training data
4) Model and training
5) Question we faced

In the following parts, we will detailedly express what we have done in these four parts and the obstacles we faced.

3.3.1 Load the dataset

In order to use the data sets from Google Cloud, we use pydrive model, os model, zipfile model to input the data sets. Besides, in order to import the data sets as data frame pattern, we also input pandas model. With the following step, we can import the data sets and apply into use.

```
!pip install -U -q PyDrive
from pydrive.auth import GoogleAuth
from pydrive.drive import GoogleDrive
from google.colab import auth
from oauth2client.client import GoogleCredentials
import os
import zipfile
import pandas as pd

# Authenticate and create the PyDrive client.
# This only needs to be done once per notebook.
auth.authenticate_user()
gauth = GoogleAuth()
gauth.credentials = GoogleCredentials.get_application_default()
drive = GoogleDrive(gauth)

# Download a file based on its file ID.
#
# A file ID looks like: laggVyWshwcyP6kEI-y_W3P8D26sz
file_id = '1RayMjAmn0PhXM0I-bypjFBwU2RtgDwSw'
downloaded = drive.CreateFile({'id': file_id})

downloaded.GetContentFile('data.zip')


path = os.listdir('.')
print (path)


with zipfile.ZipFile("data.zip","r") as datazip:
  datazip.extractall("")


data = pd.read_csv("data.csv")
```

Moreover, in order to use keras as our network foundation, we also need to import the keras model and function into training our LSTM network.

1) . In order to use data frame pattern of data sets, we import pandas model; in order to use array function, we also import numpy model.

```
import pandas as pd
import numpy as np
```

2) .Normally we use import Sequential as keras model and in order to have more effective functions, we import dense model to create full connection at the end of our network; we import dropout model to avoid over-fitting; we use activation function to add non-linear function; we import flatten model to translate matrix into vector and finally import LSTM as our main model structure.

```
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten, LSTM
```

3) we use Adam as optimizer to find the local solution with gradient descent method. Besides, we add EarlyStopping as callback method in order to calculate and compare the loss after each epoch. If the loss can't be reduced after certain number of epochs, we will automatically end the

training for efficiency and avoidance of over-fitting. Finally we import matplotlib.pyplot model in order to output the plot for better evaluation.

```
from keras.optimizers import Adam
from keras.callbacks import EarlyStopping
import matplotlib.pyplot as plt
```

3.3.2 Normalization

Before adding the feature of normalization, the training result is not that satisfactory and in order to gain better efficiency, we come up with this step of normalization.

Normalization is used to turn different ranges of data into a proper range and the mean value a proper value. In our model, since some values are quite large and unstable, we use normalization to change the range into (-1,1) and try to adjust the mean value of all data close to 0.

1). We first choose some part of data since the total number of 1700000+ data is too large. We choose the 100000 data here and we can easily change the number of data here by changing the value in the following line of code.

```
data=data.iloc[:100000]
```

2) . Then we use data.drop function to eliminate unusable features: 'UpdataTime' and 'UpdataMillisec' since we use LSTM structure to rank the feature data in time order instead of actual time. After gaining the feature and related label, we can shuffle data sets in random order for better training efficiency.

```
data=data.drop(['UpdateTime', 'UpdateMillisec'],axis=1)
```

3) . After that, we first calculate the value of actual data minus actual mean value and then divide them by the range length of actual value to normalize all actual data value into range of (-1,1) and make the normalized mean value close to 0.

```
data_norm=(data - data.mean()) / (data.max() - data.min())
```

3.3.3 Split and build training data

We use three functions to clearly establish the function of splitting, shuffling and building training data. Before establishing the function of these three function, we first pre-define some parameters which we can change easily in adjusting and revising procedure.

```
# some parameters
n_samples=len(data)
ratio=0.3
past_n=50
future_n=10
# droplist=['midPrice', 'AskPrice1','BidPrice1','AskVolume1','BidVolume1']
```

In the above definition,

Ratio represents the percentage of test data sets over total chosen data sets. Here we choose 30% to be our test data sets and 70% of data sets to be our training data sets.

Past_n represents the number of sample before current tick we use as input to predict the result. Here we choose 50 samples as input.

Future_n represents the value of tick we choose to predict after which we get the result of difference of midPrice. Here we choose to predict the difference of midPrice 10 ticks after current tick.

1) . **BuildTrain**

In buildTrain function, x_set represent the input data while y_set represent the output data. N_samples is the number of our chosen data, which is 100000.

We use “for” circulation to divide these whole 100000 data into 100000-50-10 data sets of 50 samples, which can be represented as “n_samples-past_n-future_n” data sets of “past_n” samples. In each for condition, we use append function to add to and create the input datasets and the output sets.

```
def buildTrain(data, past_n, future_n):
  X_set=[]
  Y_set=[]
  n_samples=len(data)
  for i in range(0, n_samples-past_n-future_n, 1): #step=1
    X_set.append(np.array(data.iloc[i:i+past_n]))
    Y_set.append(np.array(data.iloc[i+past_n+future_n-1]["midPrice"]-data.iloc[i+past_n-1]["midPrice"]))
  return np.array(X_set), np.array(Y_set)
```

```
X_set, Y_set=buildTrain(data_norm, past_n, future_n)
```

2) . Shuffle

We use shuffle function to shuffle the data sets in random order. Since X set is the data set of 3 dimensions, where first dimension is the number of data sets “n_samples-past_n-future_n” , the second dimension is the value of input samples and the third dimension is the number of features. Therefore we only need to random the first dimension of x set “X.shape[0]” and return the shuffled data sets.

```
def shuffle(X, Y):
  np.random.seed(6)
  randomList = np.arange(X.shape[0])
  np.random.shuffle(randomList)
  return X[randomList], Y[randomList]
```

```
X_set, Y_set=shuffle(X_set, Y_set)
```

3) . Test & Train split

We define tt_split function to split test and train data sets according to the pre-defined ratio. We separately split x_set and y_set into train and test sets.

```
def tt_split(X, Y, ratio):
  X_train = X[int(X.shape[0]*ratio):]
  Y_train = Y[int(Y.shape[0]*ratio):]
  X_test = X[:int(X.shape[0]*ratio)]
  Y_test = Y[:int(Y.shape[0]*ratio)]
  return X_train, Y_train, X_test, Y_test
```

```
X_train, Y_train, X_test, Y_test = tt_split(X_set, Y_set, ratio)
```

3.3.4 Model and training

According to LSTM structure, since it needs to rank the data according to the time order, we first use “sequential()” function to ensure the sequence of our model.

```
model = Sequential()
```

After that, we add two layers of LSTM, where the first layer is LSTM layer and the second layer is dense layer.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                (None, 100)               95200
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 101
=================================================================
Total params: 95,301
Trainable params: 95,301
Non-trainable params: 0
_________________________________________________________________
```

For the first layer, we use “LSTM(100, input_shape=(X_train.shape[1], X_train.shape[2]))” to input our data sets with sample=50 and feature=108 (X_train.shape[1]=50, X_train.shape[2]=108) and to assign the output dimension of this layer to be 100.

Besides, fer the second layer, we use “Dense(1)” to create full connection to turn 100 outputs from the first layer into 1 final output which we need as our result.

```
model.add(LSTM(100, input_length=X_train.shape[1], input_dim=X_train.shape[2]))
model.add(Dense(1))
```

After that, we use “mse” to calculate the loss and “adam” as optimizer model to train our model and use model.summary to output the test result.

3.3.5 Questions and Solution

In the process of constructing this LSTM model, we faced three questions and solved them to create a satisfied model.

Q1. Since the value of different feature vary largely, if we directly combine them together as input set, the output effects is not quite well.

A1. In order to control all data among the similar range, we import normalization method to change all data into range (-1,1). Finally the result effects is quite satisfactory.

Q2. How to split our data sets in a much more effective way?

A2. XXXXX

Q3. At first, we didn’t know how to construct a better model of LSTM.

A3. XXXXX

2.4 Test Results and Evaluation

2.4.1 Test Results

The result is quite ideal and beyond our expectation. Since we have 100 epochs, we only show 10 epochs’ results of loss at the beginning, in the middle and at the end of the training for saving space.

At the beginning

```
Train on 69958 samples, validate on 29982 samples
Epoch 1/100
69958/69958 [==============================] - 53s 755us/step - loss: 1.3135e-04 - val_loss: 2.9551e-05
Epoch 2/100
69958/69958 [==============================] - 49s 695us/step - loss: 2.5905e-05 - val_loss: 1.7643e-05
Epoch 3/100
69958/69958 [==============================] - 48s 693us/step - loss: 2.1476e-05 - val_loss: 2.0123e-05
Epoch 4/100
69958/69958 [==============================] - 47s 670us/step - loss: 1.9652e-05 - val_loss: 1.5038e-05
Epoch 5/100
69958/69958 [==============================] - 48s 690us/step - loss: 2.2024e-05 - val_loss: 1.9749e-05
Epoch 6/100
69958/69958 [==============================] - 47s 670us/step - loss: 1.9017e-05 - val_loss: 2.2234e-05
Epoch 7/100
69958/69958 [==============================] - 50s 710us/step - loss: 1.7443e-05 - val_loss: 1.4469e-05
Epoch 8/100
69958/69958 [==============================] - 49s 700us/step - loss: 1.6982e-05 - val_loss: 1.5763e-05
Epoch 9/100
69958/69958 [==============================] - 47s 670us/step - loss: 1.6646e-05 - val_loss: 1.4663e-05
Epoch 10/100
69958/69958 [==============================] - 48s 688us/step - loss: 1.5750e-05 - val_loss: 1.5566e-05
```

In the middle

```
Epoch 46/100
69958/69958 [==============================] - 49s 699us/step - loss: 9.2572e-06 - val_loss: 9.9810e-06
Epoch 47/100
69958/69958 [==============================] - 48s 680us/step - loss: 8.8536e-06 - val_loss: 8.6847e-06
Epoch 48/100
69958/69958 [==============================] - 46s 661us/step - loss: 8.4873e-06 - val_loss: 8.9796e-06
Epoch 49/100
69958/69958 [==============================] - 48s 681us/step - loss: 8.5953e-06 - val_loss: 9.0154e-06
Epoch 50/100
69958/69958 [==============================] - 46s 663us/step - loss: 8.4671e-06 - val_loss: 8.9495e-06
Epoch 51/100
69958/69958 [==============================] - 47s 678us/step - loss: 8.3387e-06 - val_loss: 8.7980e-06
Epoch 52/100
69958/69958 [==============================] - 47s 679us/step - loss: 8.1883e-06 - val_loss: 8.6529e-06
Epoch 53/100
69958/69958 [==============================] - 48s 685us/step - loss: 8.2319e-06 - val_loss: 8.2349e-06
Epoch 54/100
69958/69958 [==============================] - 47s 679us/step - loss: 8.0410e-06 - val_loss: 8.6198e-06
Epoch 55/100
69958/69958 [==============================] - 46s 661us/step - loss: 7.9505e-06 - val_loss: 9.0039e-06
```

At the end

```
Epoch 91/100
69958/69958 [==============================] - 46s 661us/step - loss: 5.1869e-06 - val_loss: 5.6113e-06
Epoch 92/100
69958/69958 [==============================] - 49s 702us/step - loss: 5.4746e-06 - val_loss: 7.1990e-06
Epoch 93/100
69958/69958 [==============================] - 48s 686us/step - loss: 5.0939e-06 - val_loss: 5.6928e-06
Epoch 94/100
69958/69958 [==============================] - 46s 662us/step - loss: 4.9484e-06 - val_loss: 5.7874e-06
Epoch 95/100
69958/69958 [==============================] - 47s 678us/step - loss: 5.0235e-06 - val_loss: 5.3977e-06
Epoch 96/100
69958/69958 [==============================] - 46s 662us/step - loss: 4.7969e-06 - val_loss: 5.8874e-06
Epoch 97/100
69958/69958 [==============================] - 48s 681us/step - loss: 4.8600e-06 - val_loss: 6.4842e-06
Epoch 98/100
69958/69958 [==============================] - 48s 689us/step - loss: 4.9549e-06 - val_loss: 5.4634e-06
Epoch 99/100
69958/69958 [==============================] - 48s 684us/step - loss: 4.7131e-06 - val_loss: 5.9498e-06
Epoch 100/100
69958/69958 [==============================] - 47s 677us/step - loss: 4.7582e-06 - val_loss: 5.8292e-06
```

From the above record of results, we can find that the loss of our model on training data (the former one) is gradually decreasing with the increase of epochs, which means our training model is effective. Moreover, the loss of our final model is in the unit of 10^{-6},which is actually quite small and ideal. **Therefore, we can make a conclusion that the effects of using LSTM method is quite good and the loss of our model is small enough.**

Besides, in this training, since the loss is continually decreasing, the function of early-stopping isn't used.

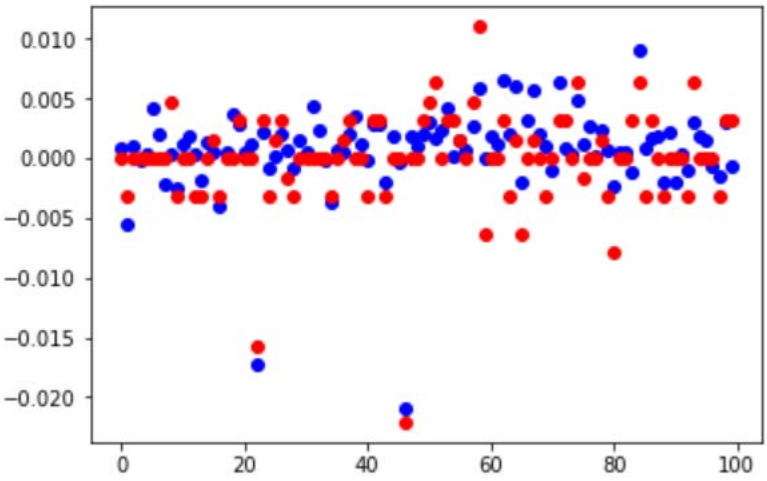
2.4.2 Evaluation

We use two methods for evaluation.

The first is to plot each estimated and true value of difference of midPrice.

```
predict_number=100
Y_esitmated=model.predict(X_test[0:predict_number])
fig=plt.figure()
plt.plot(range(predict_number),Y_esitmated,'bo')
plt.plot(range(predict_number),Y_test[0:predict_number],'ro')
plt.show()
```

In the following figure, we can see red dots and blue dots, where red dots represent the true values of results in Y_test set and blue dots represent the estimated values according to our prediction.



According to the figure, the estimated and true value are quite similar, which means our training model is quite efficient.

The second evaluation method is to use keras method to calculate the lose. It turns out that the loss of our final model is $5.829239 * 10^{-6}$ and is quite small.

https://en.wikipedia.org/wiki/Long_short-term_memory

https://zhuanlan.zhihu.com/p/32085405

Project Report: Task 2

Group member: He Yunfan, Liu Ziwei, Chen Jisen

1. Abstract

This task involves unsupervised learning to generate effective features using algorithms. Task 2 requires you to add generated features to the model of task 1, and test whether the model performance is improved in testing set.

We choose LSTM autoencoder (A deep learning framework for financial time series using auto-encoders and long-short term memory) to generate the feature of the feature of the stock information. We divide the result (the difference between MidPrice with time difference **future_n** ticks) into 8 parts: $y<-3$, $-3 \leqslant y<-2$, $-2 \leqslant y<-1$, $-1 \leqslant y<0$, $0 \leqslant y<1$, $1 \leqslant y<2$,$2 \leqslant y<3$, $y>3$. We test 5 different groups of hyperparameters to get better feature results.

The generated features improve the model performance in testing set significantly. We use a neural network to classify different parts of results. The accuracy of model using original features is **65.7%** while the one using autoencoder-generated features is **83.2%**. The performance has improved by over **18%**!

2. Model

LSTM Autoencoders

Autoencoder is an unsupervised learning method (more like self-supervised). Since our task is a sequence problem. Recurrent neural networks, such as the LSTM are very fit for sequences of data. LSTM can learn the ordering features. In LSTM autoencoders, the data are decoded, encoded and recreated. The performance of the autoencoder depends on the ability to recreate the input data. When the autoencoder has been trained, we can divide the autoencoder into encoder and decoder and use the output of the encoder to generate the compressed representation of the sequence.

Neural Networks

Neural networks can be trained as a classification model. In multi-layer NN, the ReLU activation function is often used, which is easy to calculate and has a good performance. Using dropout layer is an effective way to avoid overfitting.

3. Feature generation

1. Load the dataset

```
# Install the PyDrive wrapper & import libraries.
# This only needs to be done once per notebook.
!pip install -U -q PyDrive
from pydrive.auth import GoogleAuth
from pydrive.drive import GoogleDrive
from google.colab import auth
from oauth2client.client import GoogleCredentials
import os
import zipfile
import pandas as pd

# Authenticate and create the PyDrive client.
# This only needs to be done once per notebook.
auth.authenticate_user()
gauth = GoogleAuth()
gauth.credentials = GoogleCredentials.get_application_default()
drive = GoogleDrive(gauth)

# Download a file based on its file ID.
#
# A file ID looks like: laggVyWshwcyP6kEI-y_W3P8D26sz
file_id = '1RayMjAmnOPhXMOI-bypjFBwU2RtgDwSw'
downloaded = drive.CreateFile({'id': file_id})

downloaded.GetContentFile('data.zip')


path = os.listdir('.')
print (path)


with zipfile.ZipFile("data.zip","r") as datazip:
  datazip.extractall("")
```

We load the dataset which we have uploaded on google drive.

2. Import packages

```
import pandas as pd
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten, LSTM
from keras.layers import RepeatVector, TimeDistributed
from keras.layers.normalization import BatchNormalization
from keras.optimizers import Adam
from keras.callbacks import EarlyStopping
import matplotlib.pyplot as plt
%matplotlib inline
```

```
Using TensorFlow backend.
```

3. Normalization

Deprecate 'UpdateTime', 'UpdateMillisec' and normalize the data with Z score

```
data=data.iloc[100000:150000]
```

```
data=data.drop(['UpdateTime', 'UpdateMillisec'],axis=1)
data_norm=(data - data.mean()) / data.std()
```

```
data_norm=data_norm.drop(['midPrice', 'LastPrice',\
       'Volume', 'LastVolume', 'Turnover', 'LastTurnover', 'AskPrice5',\
       'AskPrice4', 'AskPrice3', 'AskPrice2', 'AskPrice1', 'BidPrice1',\
       'BidPrice2', 'BidPrice3', 'BidPrice4', 'BidPrice5', 'AskVolume5',\
       'AskVolume4', 'AskVolume3', 'AskVolume2', 'AskVolume1', 'BidVolume1',\
       'BidVolume2', 'BidVolume3', 'BidVolume4', 'BidVolume5', 'OpenInterest',\
       'UpperLimitPrice', 'LowerLimitPrice'],axis=1)
```

4. Build the data

```
[ ]  # some parameters
     n_samples=len(data_norm)
     ratio=0.3
     past_n=60
     future_n=40 # label值分的不是很开 提高 future_n?
     # droplist=['midPrice', 'AskPrice1','BidPrice1','AskVolume1','BidVolume1']


     def buildTrain(data,past_n,future_n, data_original):
       X_set=[]
       Y_set=[]
       n_samples=len(data)
       for i in range(0,n_samples-past_n-future_n,1): #step=1
         X_set.append(np.array(data.iloc[i:i+past_n]))
         Y_set.append(np.array(data_original.iloc[i+past_n+future_n-1]["midPrice"]-data_original.iloc[i+past_n-1]["midPrice"]
       return np.array(X_set), np.array(Y_set)


     def shuffle(X,Y):
       np.random.seed(6)
       randomList = np.arange(X.shape[0])
       np.random.shuffle(randomList)
       return X[randomList], Y[randomList]

     def tt_split(X,Y,ratio):
       X_train = X[int(X.shape[0]*ratio):]
       Y_train = Y[int(Y.shape[0]*ratio):]
       X_test = X[:int(X.shape[0]*ratio)]
       Y_test = Y[:int(Y.shape[0]*ratio)]
       return X_train, Y_train, X_test, Y_test

     X_set, Y_set=buildTrain(data_norm, past_n, future_n, data)

     X_set, Y_set=shuffle(X_set, Y_set)

     X_train, Y_train, X_test, Y_test = tt_split(X_set, Y_set, ratio)
```

Divide the dataset into the size which LSTM needs.

```
print(Y_train[(Y_train>3)].size)
print(Y_train[(Y_train>2)&(Y_train<=3)].size)
print(Y_train[(Y_train>1)&(Y_train<=2)].size)
print(Y_train[(Y_train>0)&(Y_train<=1)].size)
print(Y_train[(Y_train>-1)&(Y_train<=0)].size)
print(Y_train[(Y_train>-2)&(Y_train<=-1)].size)
print(Y_train[(Y_train>-3)&(Y_train<=-2)].size)
print(Y_train[(Y_train<=-3)].size)
```

```
2201
1819
3754
6215
7459
6501
3700
3281
```

```
# def f(x):
# return(10*x)
# a=[1,2,3]
# b=[f(t) for t in a]
# print(b)
def getPatternValue(x):
  if (x>3):
    return 0
  elif (x>2 and x<=3):
    return 1
  elif (x>1 and x<=2):
    return 2
  elif (x>0 and x<=1):
    return 3
  elif (x>-1 and x<=0):
    return 4
  elif (x>-2 and x<=-1):
    return 5
  elif (x>-3 and x<=-2):
    return 6
  else:
    return 7


pattern_value=[getPatternValue(t) for t in Y_test]
print(len(pattern_value))
```

```
14970
```

Divide the result (the difference between MidPrice with time difference future_n ticks) into 8 parts: y<-3, -3≤y<-2, -2≤y<-1, -1≤y<0, 0≤y<1, 1≤y<2 ,2≤y<3, y>3.

5. Build the model

```
model = Sequential()
model.add(LSTM(100,name='mid', input_shape=(X_train.shape[1], X_train.shape[2])))
model.add(RepeatVector(X_train.shape[1]))
model.add(LSTM(100, return_sequences=True))
model.add(TimeDistributed(Dense(X_train.shape[2])))
model.compile(loss="mse", optimizer="adam")
model.summary()
```

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/tensorflow/python/f
Instructions for updating:
Colocations handled automatically by placer.
```

Layer (type)	Output Shape	Param #
mid (LSTM)	(None, 100)	83600
repeat_vector_1 (RepeatVecto	(None, 60, 100)	0
lstm_1 (LSTM)	(None, 60, 100)	80400
time_distributed_1 (TimeDist	(None, 60, 108)	10908

```
Total params: 174,908
Trainable params: 174,908
Non-trainable params: 0
```

Here is our LSTM autoencoder structure.

6. Train the model

We train the autoencoder with 200 epochs and with early stopping=10.

```
callback = EarlyStopping(monitor="loss", patience=10, verbose=1, mode="auto")
model.fit(X_train, X_train, epochs=200, batch_size=128, validation_data=(X_test, X_test), callbacks=[callback])
```

```
34930/34930 [==============================] - 77s 2ms/step - loss: 0.3983 - val_loss: 0.4091
Epoch 193/200
34930/34930 [==============================] - 77s 2ms/step - loss: 0.3992 - val_loss: 0.4029
Epoch 194/200
34930/34930 [==============================] - 77s 2ms/step - loss: 0.3994 - val_loss: 0.4047
Epoch 195/200
34930/34930 [==============================] - 77s 2ms/step - loss: 0.3989 - val_loss: 0.4054
Epoch 196/200
34930/34930 [==============================] - 77s 2ms/step - loss: 0.3993 - val_loss: 0.4057
Epoch 197/200
34930/34930 [==============================] - 77s 2ms/step - loss: 0.3987 - val_loss: 0.4050
Epoch 198/200
34930/34930 [==============================] - 77s 2ms/step - loss: 0.3985 - val_loss: 0.4035
Epoch 199/200
34930/34930 [==============================] - 78s 2ms/step - loss: 0.3972 - val_loss: 0.4040
Epoch 200/200
34930/34930 [==============================] - 77s 2ms/step - loss: 0.3989 - val_loss: 0.4021
<keras.callbacks.History at 0x7fbaf9f5ce48>
```

We test 5 different groups of hyperparameters, the loss still a little bit high

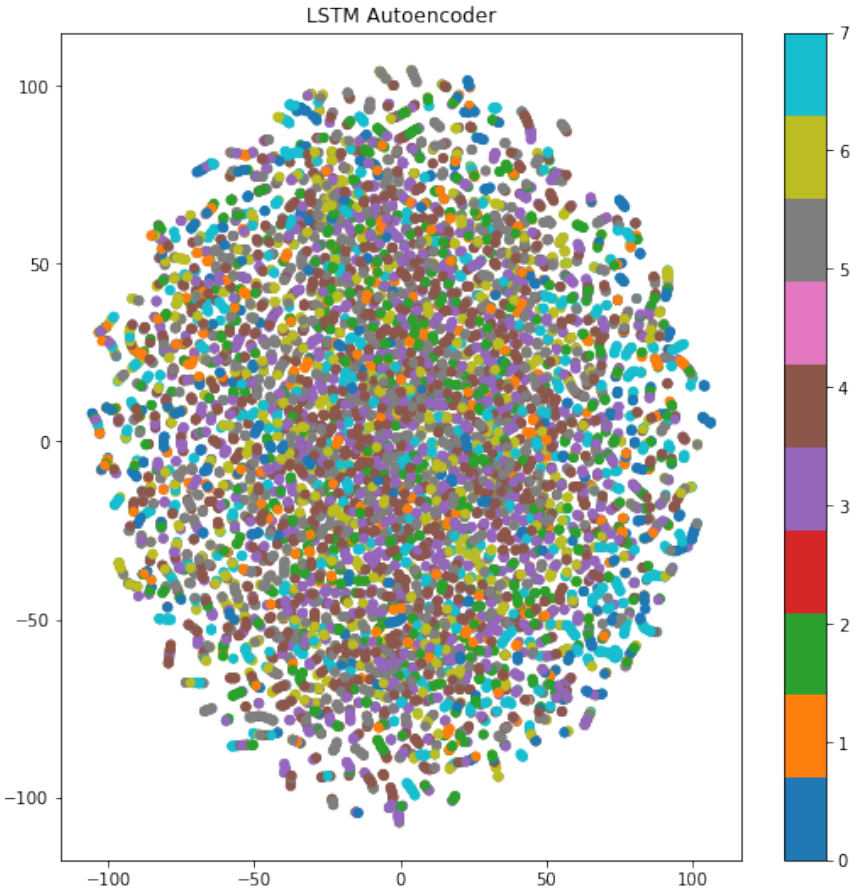
7. Feature generation

```
import sklearn
from sklearn.manifold import TSNE
from keras.models import Model

mid_model = Model(inputs=model.input, outputs=model.get_layer('mid').output)

pattern_raw=mid_model.predict(X_test)


tsne=TSNE(n_components=2,init='pca',random_state=0)
X_tsne=tsne.fit_transform(pattern_raw)
```

```
plt.figure(figsize=(9, 9))
plt.scatter(X_tsne[:, 0], X_tsne[:, 1], c=pattern_value, cmap='tab10',linewidths=0.00000001)
plt.colorbar()
plt.title("LSTM Autoencoder")
plt.show()
```

We use keras.Model to take out the midlayer of the autoencoder as encoder. And use test set data to generate the pattern to check the feature classification in 2D graph.

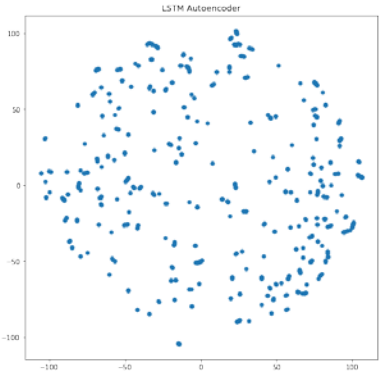
All eight catagories:



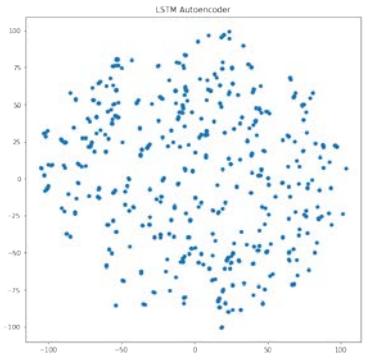
The result may not be that clear, so we divide the pattern of each categories.

```
k=0
a=np.array(pattern_value)
b=np.argwhere(a==k)
#print(b)
#print(X_tsne[10])
t=X_tsne[b]
# print(X_tsne)
# print(t)
# print(t[0,0,0])
plt.figure(figsize=(9, 9))
plt.scatter(t[:,0, 0], t[:,0, 1],linewidths=0.00000001)
#plt.colorbar()
plt.title("LSTM Autoencoder")
plt.show()
```

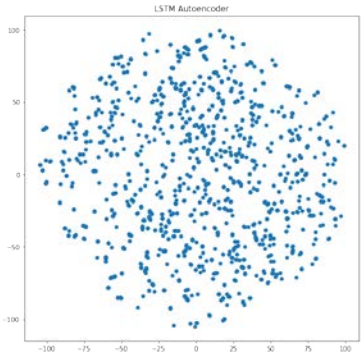
We use this code with different k to generate the pattern of each category.



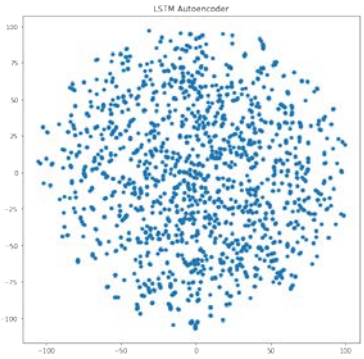
k=0



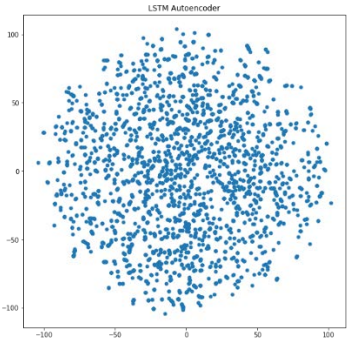
k=1



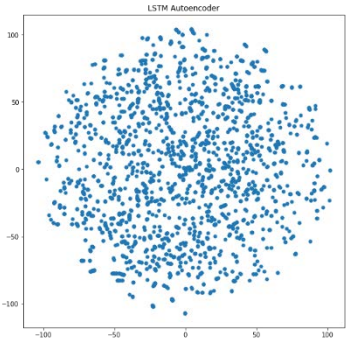
k=2



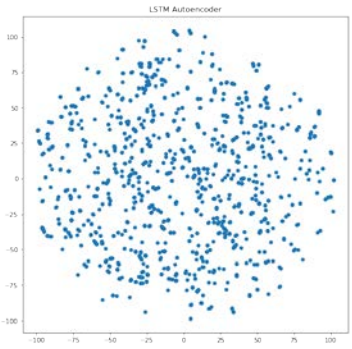
k=3



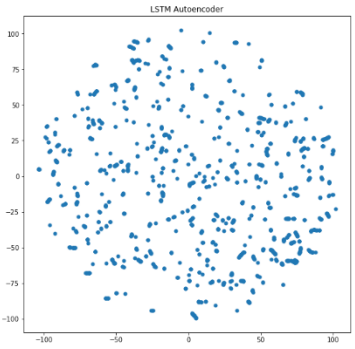
k=4



k=5



k=6



k=7

4. Classification

Model with original features

We use a neural network to classify different parts of results. Without using generated features, we use past 60 lines and each line has 108 indicators, and these 6480 numbers are the input of model. The output is a number varying from 0 to 7.
We use a three-layer NN with two drouput layers.

```
[ ]  model13 = Sequential()
     model13.add(Flatten(input_shape=(X_train.shape[1], X_train.shape[2])))
     model13.add(Dense(2048, activation='relu' ))
     model13.add(Dropout(0.5))
     model13.add(Dense(2048, activation='relu' ))
     model13.add(Dropout(0.5))
     model13.add(Dense(8, activation='softmax'))
     model13.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
     model13.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_9 (Flatten)          (None, 6480)              0
_________________________________________________________________
dense_25 (Dense)             (None, 2048)              13273088
_________________________________________________________________
dropout_9 (Dropout)          (None, 2048)              0
_________________________________________________________________
dense_26 (Dense)             (None, 2048)              4196352
_________________________________________________________________
dropout_10 (Dropout)         (None, 2048)              0
_________________________________________________________________
dense_27 (Dense)             (None, 8)                 16392
=================================================================
Total params: 17,485,832
Trainable params: 17,485,832
Non-trainable params: 0
_________________________________________________________________
```

After fitting, the final best accuracy in test set is **65.7%**.

```
Epoch 00025: val_acc did not improve from 0.65478
Epoch 26/100
34930/34930 [==============================] - 3s 83us/step - loss: 0.5561 - acc: 0.7892 - val_loss: 1.1644 - val_acc: 0.6509

Epoch 00026: val_acc did not improve from 0.65478
Epoch 27/100
34930/34930 [==============================] - 3s 84us/step - loss: 0.5556 - acc: 0.7925 - val_loss: 1.1658 - val_acc: 0.6523

Epoch 00027: val_acc did not improve from 0.65478
Epoch 28/100
34930/34930 [==============================] - 3s 84us/step - loss: 0.5590 - acc: 0.7891 - val_loss: 1.1567 - val_acc: 0.6572

Epoch 00028: val_acc improved from 0.65478 to 0.65718, saving model to model13.ckpt
Epoch 29/100
34930/34930 [==============================] - 3s 84us/step - loss: 0.5523 - acc: 0.7917 - val_loss: 1.1502 - val_acc: 0.6502

Epoch 00029: val_acc did not improve from 0.65718
Epoch 30/100
34930/34930 [==============================] - 3s 84us/step - loss: 0.5498 - acc: 0.7918 - val_loss: 1.1798 - val_acc: 0.6476

Epoch 00030: val_acc did not improve from 0.65718
```

Model with generated features

We have trained an autoencoder before, therefore we reuse it in the following model. The model is a combination of autoencoder and neural network. The "mid_model" is the first layer of autoencoder, and we set it to be not trainable. Its output is the input of neural network classification model.

```
x = Dense(2048, activation='relu')(mid_model.output)
x = Dropout(0.5)(x)
x = Dense(2048, activation='relu')(x)
x = Dropout(0.5)(x)
x = Dense(8, activation='softmax')(x)

model2 = Model(inputs=mid_model.input, outputs=x)


for layer in mid_model.layers:
    layer.trainable = False


model2.compile(optimizer='adam', loss='sparse_categorical_crossentropy'
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
mid_input (InputLayer)       (None, 60, 108)           0
_________________________________________________________________
mid (LSTM)                   (None, 100)               83600
_________________________________________________________________
dense_29 (Dense)             (None, 2048)              206848
_________________________________________________________________
dropout_11 (Dropout)         (None, 2048)              0
_________________________________________________________________
dense_30 (Dense)             (None, 2048)              4196352
_________________________________________________________________
dropout_12 (Dropout)         (None, 2048)              0
_________________________________________________________________
dense_31 (Dense)             (None, 8)                 16392
=================================================================
Total params: 4,503,192
Trainable params: 4,419,592
Non-trainable params: 83,600
_________________________________________________________________
```

After fitting, the final best accuracy in test set is **83.2%**, which is improved significantly.

```
Epoch 00093: val_acc did not improve from 0.83140
Epoch 94/100
34930/34930 [==============================] - 4s 109us/step - loss: 0.2364 - acc: 0.9011 - val_loss: 0.5687 - val_acc: 0.8309

Epoch 00094: val_acc did not improve from 0.83140
Epoch 95/100
34930/34930 [==============================] - 4s 110us/step - loss: 0.2317 - acc: 0.9041 - val_loss: 0.5744 - val_acc: 0.8301

Epoch 00095: val_acc did not improve from 0.83140
Epoch 96/100
34930/34930 [==============================] - 4s 111us/step - loss: 0.2338 - acc: 0.9054 - val_loss: 0.5691 - val_acc: 0.8300

Epoch 00096: val_acc did not improve from 0.83140
Epoch 97/100
34930/34930 [==============================] - 4s 111us/step - loss: 0.2251 - acc: 0.9072 - val_loss: 0.5831 - val_acc: 0.8289

Epoch 00097: val_acc did not improve from 0.83140
Epoch 98/100
34930/34930 [==============================] - 4s 111us/step - loss: 0.2285 - acc: 0.9062 - val_loss: 0.5738 - val_acc: 0.8324

Epoch 00098: val_acc improved from 0.83140 to 0.83240, saving model to model2.ckpt
Epoch 99/100
34930/34930 [==============================] - 4s 111us/step - loss: 0.2247 - acc: 0.9063 - val_loss: 0.5767 - val_acc: 0.8310

Epoch 00099: val_acc did not improve from 0.83240
Epoch 100/100
34930/34930 [==============================] - 4s 113us/step - loss: 0.2210 - acc: 0.9090 - val_loss: 0.5863 - val_acc: 0.8307

Epoch 00100: val_acc did not improve from 0.83240
```

5. Discussion and conclusion

The visualization of feature generation result is not very ideal. We can conclude from the figure that the point that the less the midPrice changes, the closer the data points come to the origin point.

However, we are glad to see that the LSTM autoencoders generated good features. The accuracy of generated features is **83.2%** which is far more better than the original accuracy **65.7%**.

Reference:
https://machinelearningmastery.com/lstm-autoencoders/

Project Report: Task 3

Trading strategy based on reinforcement learning

Group member: He Yunfan, Liu Ziwei, Chen Jisen

1. Abstract

This task requires us to use reinforcement learning to control the machine to buy and sell stock to get profits as much as possible. This task sets that each tick has at most 5 hand long positions and 5 hand short positions. Long positions and short positions cannot be held at the same time. A tick can only have one action at a time. Positions can be increased or decreased (with unit equals one hand) through buying and selling, and the absolute value of change in the number of positions of one action cannot exceed one hand.

The advantages of DQN are experiment replay and fix Q targets. Experiment replay means that it can train itself again and again. Fix Q target can cut down the relevance, which will make training result better.

2. Methodology

a. Choose the reinforcement algorithm

Nowadays, there dozens of algorithms for RL, such as Q-learning, DQN, DDPG, A2C, PPO2 and so on. After discussion, we finally choose DQN (Deep Q network), which is easy to implement and better than Q-learning which can only solve small number of states problem, while stock market has infinite states. So, we need to use neural network to generate the Q value.

b. Define reward and action space

The target of the task is to earn money as much as possible, so we need to define the reward of the algorithm, which is the most important part in DQN.

Initial:

We first have 5 stocks with current value. We make a variable to record the position of the stock and another variable to record the average value of the stocks.

Action and Reward:

Action: Buy (2), Sell (1), Hold (0). For stock position=9, when we decide to buy, we will make it hold and give reward=0. For stock position=0, when we decide to sell, we will make it hold and give reward=0. When action=2 (buy), the reward will be 0 and update the average price. When action=1 (sell), reward=current price-average price.

```
reward=0
        # Action: Buy(2),Sell(1),Hold(0)
        if action==1 and len(agent.position)>0:
                pop_price=agent.position.pop(0)
                real_price=avgPrice
                # reward=max(bidTrain[t]-real_price,0)
                reward=bidTrain[t]-real_price
                profit+=(bidTrain[t]-real_price)
                print("Time:%10d Type:%6s Position:%4d Price:%8.1f Profit:%6
.1f Total profit:%16.1f"%(t,"SELL", len(agent.position)-
5,bidTrain[t],bidTrain[t]-real_price,profit))
        elif action==2 and len(agent.position)<10:
```

```
                reward=avgPrice-askTrain[t]
                totalValue=avgPrice*len(agent.position)
                totalValue+=askTrain[t]
                agent.position.append(askTrain[t])
                avgPrice=totalValue/len(agent.position)
                print("Time:%10d Type:%6s Position:%4d Price:%8.1f"%(t,"BUY"
, len(agent.position)-5,askTrain[t]))
```

c. DQN agent

Here is our code

```
import numpy as np
import pandas as pd
import random

import keras
from keras.models import Sequential
from keras.models import load_model
from keras.layers import Dense
from keras.optimizers import Adam

from collections import deque


# State:
# [stock_owned=(-5,5),stock_price=midPrice,cash=0]

# Action:
# Buy(2),Sell(1),Hold(0)

class MyAgent:
    def __init__(self, state_num,is_Eval=False, load_model_name=""):
        self.state_num=state_num
        self.action_num=3
        self.position=[]

        self.is_Eval=is_Eval

        self.Memory=deque(maxlen=1000)
        self.gamma=0.9
        self.epsilon = 1.0
        self.epsilon_min = 0.01
        self.epsilon_decay = 0.995
        self.learning_rate = 0.005
```

```
34.
35.         if is_Eval:
36.             self.model=load_model(load_model_name)
37.         else:
38.             self.model=self.build_model()
39.
40.
41.     # LSTM
42.     # def build_model(self):
43.     #     model=Sequential()
44.     #     model.add()
45.
46.
47.     # MLP
48.     def build_model(self):
49.         model=Sequential()
50.         model.add(Dense(64, input_dim=self.state_num, activation="relu"))
51.         model.add(Dense(64, activation="relu"))
52.         model.add(Dense(32, activation="relu"))
53.         model.add(Dense(8, activation="relu"))
54.         model.add(Dense(self.action_num, activation="linear"))
55.         model.compile(loss="mse", optimizer=Adam(lr=self.learning_rate))
56.
57.         return model
58.
59.     def act(self, state):
60.         """
61.         The maximum position is 5
62.         """
63.         if not self.is_Eval:
64.             if np.random.rand()<self.epsilon:
65.                 return random.randrange(self.action_num)
66.         act_values=self.model.predict(state)
67.         return np.argmax(act_values[0])
68.
69.     def replay(self,batch_size):
70.         mini_batch=random.sample(self.Memory, batch_size)
71.
72.         for state, action, reward, next_state, done in mini_batch:
73.             # print(next_state.shape)
74.             # print(next_state)
75.             if not done:
76.                 target=reward+self.gamma * np.amax(self.model.predict(next_s
    tate)[0])
```

```
            else:
                target=reward
            label = self.model.predict(state)
            label[0][action] = target
            self.model.fit(state, label, epochs=1, verbose=0)
        self.epsilon_update()
        return

    def epsilon_update(self):
        if self.epsilon>self.epsilon_min:
            self.epsilon*=self.epsilon_decay
        return

    def remember(self,state,action,reward,next_state,done):
        self.Memory.append((state,action,reward,next_state,done))
        return

```

We use the most common mlp model to generate the q value
The hyperparameters are:
Memory length=1000. (Remember the latest 1000 state, action, reward, next state and done and use them in replay function)
Gamma=0.9

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

d. Total training process

The total episode number is 100.

First, get the state, and set it as the current state.

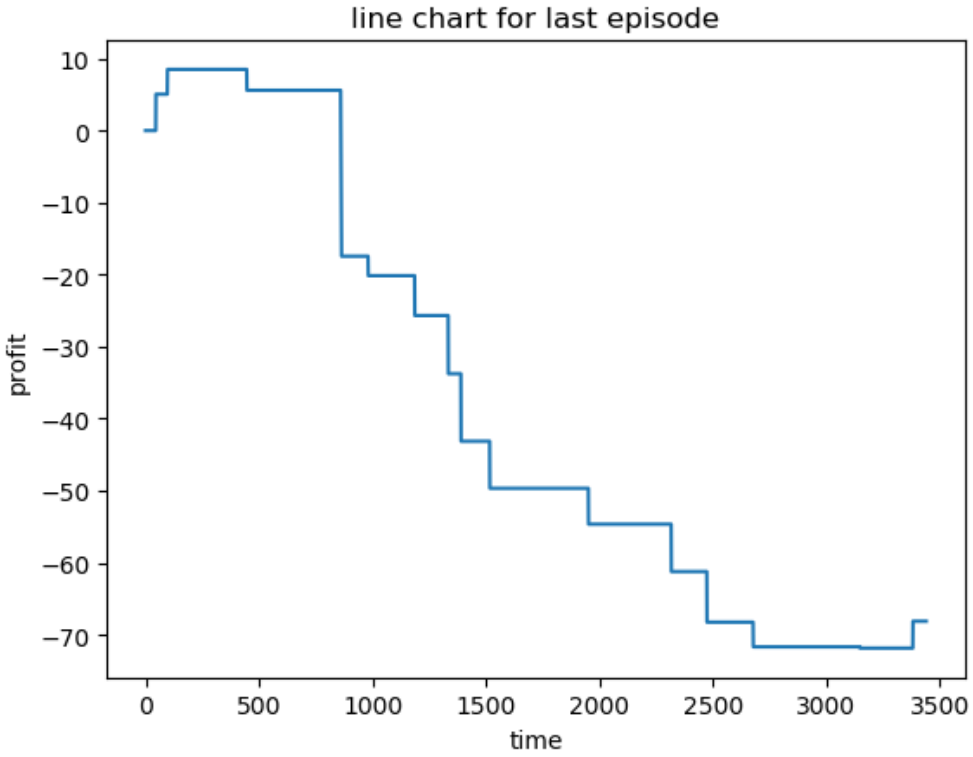
Second, get the action of the agent according to the current state and update the next state.

Third, set the initial reward to be zero. Act according to the action of the agent and the maximum and minimum position, and get the reward.

Then, write the (state,action,reward,next_state,done) into the memory of agent. Update the current state with the next state. Make the agent replay.

Save the model every 30 episode.

Finally, print the result on the screen.



3. Conclusion and discussion

Reinforcement learning is a heated topic now, and it is hard to learn all the algorithms of RL, so we choose DQN, which is easy to implement. The advantages of DQN are experiment replay and fix Q targets. Experiment replay means that it can train itself again and again. Fix Q target can cut down the relevance, which will make training result better

We can conclude from the figure that we first makes some money, while after about 500 ticks, the profit goes down. The reason of this result may be the simple mlp neural network cannot generate the good Q value.

For further improvement, we will use LSTM as the neural network of DQN

Reference:
https://github.com/edwardhdlu/q-trader