Imperial College London

Imperial College London

Department of Mathematics

Predicting High-Frequency Stock Market by Neural Networks

Author: Yuchen Tu (CID: 01219050)

A thesis submitted for the degree of

MSc in Mathematics and Finance, 2019-2020

Declaration

The work contained in this thesis is my own work unless otherwise stated.

Signature: Yuchen Tu

Date: 08/09/2020

Acknowledgements

I would like to thank Dr Mikko S. Pakkanen for his patient support in the past four years. I highly appreciate his insightful suggestions and advice on my thesis.

I also want to thank my family and my girlfriend for their love.

Abstract

In this thesis, two tasks are investigated in the high-frequency stock trading market. The first is predicting the direction of the next mid-price change. The second task focuses on forecasting when the change would occur. To accomplish the two tasks, I introduced four types of neural network models: FNN without lagged features, FNN with lagged features, many-to-one RNN with LSTM layers, many-to-many RNN with LSTM layers. Multiple configurations of models are tested for these tasks in order to locate the best-performed models. After intensive model training and parameters tuning, the result shows that the many-to-many Recurrent Neural Network with LSTM layers outperforms all the other models in both tasks. In the meantime, We would find the ability of the many-to-many RNN with LSTM layers in capturing the long-term dependence between the labels and the past limit order book (LOB) information. Moreover, while dealing with these two tasks with stock-specific models, I also tried to use limited data to build universal models that can work for every stock in the market, and the universal model built in the first task demonstrates good accuracy.

Contents

List of Figures

List of Tables

Introduction

In 1974, W.A. Little [1] described a type of neural network, and the idea was further developed by Hopfield [2], who formulated a type of RNN called Hopfield network. Hopfield network was designed to resemble the memory system in the brain, and it is fully connected with a single layer, inspired by theoretical neuroscience. In 1986, David Rumelhart [3] published a paper describing the details of the backpropagation algorithm in multi-layer neural networks. The paper also demonstrated that multi-layer networks trained with backpropagation could turn the units in hidden layers into some crucial features, which further prospered the research of RNN in many aspects such as image recognition. Under the support of the prior researches, the previously unsolvable task was solved by Schmidhuber [4] in 1993, where he utilised an RNN unfolded in time with more than 1000 layers.

However, the success rate of training an RNN with backpropagation was not satisfying at the beginning [5]. In particular, RNNs often 'forget' what happened too long ago. Hochreiter [6] identified the reason, which is now called the vanishing and exploding gradient problem. More precisely, the problem describes a situation where gradients vanish or explode at some layers, which makes the training process extremely slow or unstable. Researchers developed different solutions to overcome this problem, for instance, Long short-term memory (LSTM), Residual networks and Multi-level hierarchy. Out of these solutions, LSTM is regarded as a popular solution. In 1997, the conception of LSTM was formally published by Sepp Hochreiter and Jürgen Schmidhuber [7]. Compared with classical RNNs, LSTM solves the vanishing and exploding gradient problem by changing the dynamics of gradient calculation: many multiplicative operations are switched to additive operations, which makes the gradient much less likely to vanish or explode. The invention of LSTM gave us a brand new way to implement RNN, six years later, a group of researchers [8] pointed out that LSTM could be advantageous in speech recognition tasks. In 2007, a revolution [9] of speech recognition was triggered by LSTM, a newly designed discriminative keyword spotting system based on LSTM out vastly outperformed a traditional system based on hidden Markov models in a keyword spotting task. Two years later, an LSTM based model [10] outperforms all the candidates in the 2007 ICDAR Arabic recognition contest, which demonstrated the power of LSTM in handwriting recognition. In 2015, Google announced its updated acoustic models [11] based on LSTM, which was claimed to be more accurate and efficient in voice search.

So far, we have seen numerous achievements that LSTM has made since its invention, but these achievements are concentrated in areas such as speech recognition and pattern recognition. However, in the stock markets of the world, high-frequency stock trading data is generated at almost every moment, most of the data is in the form of time series with a long history. Besides, high-frequency stock trading data is well known for its nonlinearity and low signal-to-noise ratio. These properties make high-frequency stock trading data perfect in deep learning studies. In recent years, there are many successful attempts in applying LSTM in the stock market. For example, in 2020, Jiayu Qiu and his team [12] showed that LSTM with denoised input stock trading data could quite accurately predict stock prices. Besides, a profound result [13] was established by Justin Sirignano and Rama Cont, by using LSTM on trading data for over 1000 stocks, they found the existence of universal price formation mechanism. In the same paper, they also built stock-specific models that consistently outperformed the corresponding linear (VAR) models.

This thesis begins with an introduction of the limit order book (LOB) and a description of the high-frequency stock trading data, followed by an initial feature selection using the sequential forward selection method. Then I want to discuss two tasks of neural networks in high-frequency stock trading data: predicting the direction of the next mid-price change and predicting when the next mid-price change will occur. The procedures of tackling these two tasks are the same: I start from using four different types of neural networks, including FNN without lagged features, FNN

with lagged features, many-to-one RNN with LSTM layers and many-to-many RNN with LSTM layers. Then for each type of the neural networks, different configurations are tried to find its best performance. In the end, the best performance of different types of neural networks are compared with each other to find out which type of model is the best for the task. It is interesting to see that different types of neural networks can behave quite distinctively. As a result, the many-to-many RNN with LSTM layers is proved to be the most powerful model out of the four candidates in both tasks. Moreover, following the idea of Justin Sirignano and Rama Cont's paper, I would also try to build two naive universal models with a limited amount of data for the two tasks separately.

Throughout the paper, the many-to-many RNN with LSTM layers always outperforms the FNNs in both tasks, which indicates the existence of dependence between the labels and the past LOB information, and also proves the many-to-many RNN with LSTM layers is better at catching the dependence between the labels and the past LOB information.

Chapter 1

Data Preparation

1.1 The limit order book

When trading with a limit order book, traders often place two types of orders in the market: limit order and market order. A buy limit order allows a trader to set the maximum size and the maximum price at which the trader is willing to buy. Similarly, a sell limit order allows a trader to set the maximum size and the minimum price at which the trader is willing to sell. A limit order is executed only if there are market orders satisfying requirements set by one or more limit orders, and it can be partially executed. Therefore, limit orders are sometimes not executed immediately. The definition of the market order is much simpler, a market order allows a trader to buy or sell a specific size of an asset at the best possible prices, and it is usually executed immediately and completely, but the price is not guaranteed when the market order is placed.

The limit order book is nowadays widely used as a record of limit orders of sellers and buyers in the market, and they are mainly managed by exchanges, where order matching systems are applied on the limit order book data to decide whether and how an order should be executed. For limit orders, the order matching systems always give execution priority to limit orders with the highest bid price or lowest ask price, apart from that, public orders have higher priority over hidden orders. In most stock markets, for limit orders at the same price level, limit orders that placed earlier gain priority over limit orders placed later. However, in most option and commodity markets, pro-rata rule is more common, that is, limit orders at the same price level are executed in proportion to their sizes, no matter which limit order was placed first.

Usually, a limit order book is presented in the form of Table 1.1. It allows traders to see the price levels and sizes of outstanding limit orders. Note that it is possible to see more levels of a limit order book, but for demonstration, five levels should be adequate. In a full-level limit order book, outstanding limit orders of all levels are available, it is available with additional fees in some markets .

ask price 5	ask size 5
ask price 4	ask size 4
ask price 3	ask size 3
ask price 2	ask size 2
ask price 1	ask size 1
bid price 1	bid size 1
bid price 2	bid size 2
bid price 3	bid size 3
bid price 4	bid size 4
bid price 5	bid size 5

Table 1.1: A typical 5-level limit order book

At each price level of a limit order book, the total number of outstanding limit orders is displayed, so that if one wants to place a market order, he/she would be able to calculate how many price levels are going to be covered by the market order. When there is a change in the limit order book, the system creates one or more new rows of data to record the updated status of the limit order book, this kind of data forms the data used to train the models in the thesis. Besides,

I am going to focus on the stock market, and it is standard to use the mid-price to track the trend of the stock price.

Definition 1.1.1 (mid-price)**.** Here we define the mid-price as:

$$\text{mid-price} = \frac{\text{ask price 1} + \text{bid price 1}}{2}.$$

1.2 Data description

The data used in the thesis is stock tick data from the LOBSTER database [14], which consists of every change in the limit order book and the basic information from price level 1 to price level 5. The data are from three stocks: WMT, GOOG, JPM. The three stocks are selected from different industries. The corresponding periods are from 01-05-2020 to 10-05-2020 for training and from 12-05-2020 to 13-05-2020 for testing. After some simple manipulation of the raw data, I formed the original data frame (see Figure 1.1).

	0	1	2	3	...	17	18	19	time
0	865300	100	862800	43	...	1000	861900	25	34200.062173
1	865300	100	862800	43	...	1000	861900	25	34200.062211
2	869300	2	862800	43	...	100	861900	25	34200.062230
3	869300	2	862800	43	...	100	861900	25	34200.062256
4	869300	2	862800	43	...	600	861900	25	34200.063099
5	869300	2	862800	43	...	600	861900	25	34200.063141
6	869300	102	862800	43	...	600	861900	25	34200.091973
7	869300	102	862800	43	...	100	861900	25	34200.187320
8	869300	102	862500	20	...	100	861000	17	34200.196515
9	869300	102	863100	43	...	100	861900	25	34200.198767
10	869300	2	863100	43	...	100	861900	25	34200.297845
11	869300	2	863100	43	...	100	861900	25	34200.413762
12	869300	2	863100	43	...	100	861900	25	34200.413773
13	869300	2	863100	43	...	100	861900	25	34200.413967
14	869300	2	863100	43	...	100	861900	25	34200.413970
15	869300	2	863100	43	...	100	861900	25	34200.414361
16	869300	2	863100	43	...	600	861900	25	34200.414366
17	869300	2	863100	43	...	600	861900	25	34200.479768
18	869300	2	862500	20	...	600	861700	20	34200.479815
19	869300	2	862500	20	...	100	861700	20	34200.480955
20	869300	2	862500	20	...	100	861700	20	34200.480963
21	869300	2	862500	20	...	100	861700	20	34200.482233
22	869300	2	862500	20	...	600	861700	20	34200.482238
23	869300	2	862900	43	...	600	861900	25	34200.483984
24	869300	2	862500	20	...	600	861700	20	34200.548909
25	869300	2	862800	43	...	600	861900	25	34200.549989

Figure 1.1: Screenshot of the dataframe.

In Figure 1.1, columns with numbers as headers are bid prices, bid sizes, ask prices and sizes. For example, 0, 1, 2, 3, 4, represents ask price 1, ask size 1, bid price 1, bid size 1, respectively. We can see that sometimes the mid-price remains unchanged, so there are three types of direction in the original data: '-1', '0', '1', representing a drop in mid-price, unchanged mid-price, a rise in mid-price. As in this thesis, the main focus would be on how the mid-price goes up and down, so I prefer to construct a binary classification problem in Chapter 3. In Chapter 3, the predictions are made when mid-price changes, the rows with direction '0' are deleted there for simplicity. In Chapter 4, predictions are made every time the LOB information changes. Thus, rows with direction '0' are not deleted.

1.3 Data cleaning and feature scaling

First of all, the reader may observe that there are sometimes two rows with the same timestamp in Figure 1.1. This is because sometimes there are market orders that match limit orders at more than one price levels, then the limit order book will update more than once and hence creates more than one row of data, and I would only keep the last row in these cases. As for the missing values, the principle of dealing with missing values here is to delete the row with missing values, but after careful checking, I did not find any missing values in the original data.

To help the neural networks learn quickly and efficiently from the data, it is a common practice to scale the data. In this thesis, all the input data is scaled by standardization in this thesis, which is defined as:

$$a' = \frac{a - mean(a)}{std(a)},$$

where $std(a)$ is the standard deviation of a, and a is a vector.

When applying the scaling method on the training and test dataset, it would be based on the whole training dataset, that is, $mean(\cdot)$ and $std(\cdot)$ from the training dataset are used for both training data scaling and test data scaling.

1.4 Feature selection

In this section, I am going to select the features for the models in the following chapters. The selected features will be directly used in RNN models and FNN without lagged features, but note that there are also some Feedforward neural networks with lagged features in the thesis, features for those benchmark models are very similar to the features I selected here but may be adjusted accordingly.

Due to the properties of the neural network, simple features will be transformed into some complex features by the network automatically during training, and there are even people using the neural network itself as a feature selection tool. Therefore, I would not try to add too many complex features. However, I still need to decide which simple features are needed. Usually, the random forest is a common method in feature selection for neural networks. A random forest consists of multiple decision trees; it can work for both classification and regression problem, which seems to be the right choice. However, the results given by the random forest is not always satisfying. In my case, I do not find them particularly useful. In figure 1.2, a typical result of random forest feature selection based on four stock's tick data is shown.

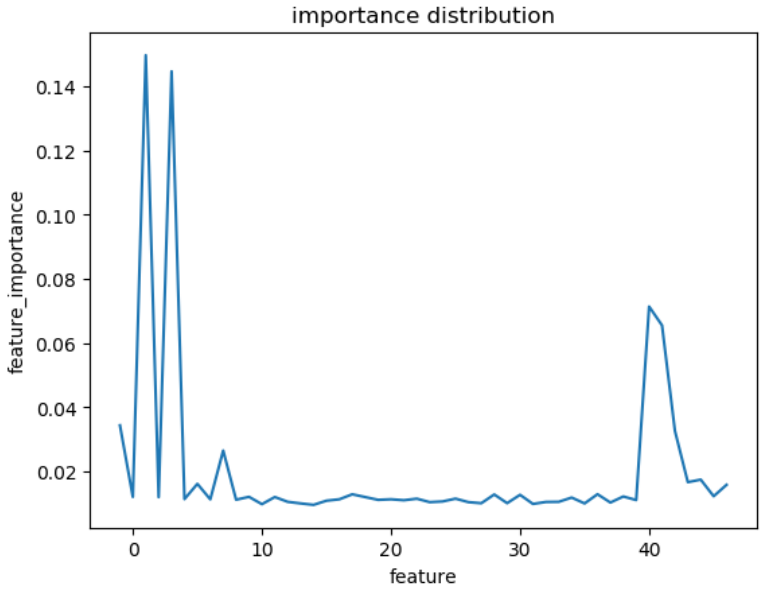

Figure 1.2: Random Forest Feature selection result for the direction of next mid-price change prediction.

In Figure 1.2, Index -1 is time, indices 0 to 39 are the 40 columns of bid-ask prices/sizes, the order is ask price 1, ask size 1, bid price 1, bid size 1 then ask price 2, ask size 2, and so on, index 41 is the time from the last mid-price change, indices 40, 42, 43, 44, 45 are the directions of the last five mid-price changes(larger index indicates further away from the current change).

It is worth noting that bid-ask prices in the Figure 1.2 are not given enough feature importances, almost none of them are selected by the method. However, in practical experiments, a model that uses bid-ask prices data would very often beat a model that does not use bid-ask prices. The

following example can explain the main reason of this phenomenon: suppose at some moment, the bid-ask prices in the limit order book of a stock look like the table 1.2, then it is intuitive to predict the mid-price would go down in the next moment due to the lack of support from the bid prices quoted within five levels.

ask price 5	20.04
ask price 4	20.03
ask price 3	20.02
ask price 2	20.01
ask price 1	20.00
bid price 1	19.99
bid price 2	19.92
bid price 3	19.87
bid price 4	19.85
bid price 5	19.80

Table 1.2: An example of limit order book of a stock

Therefore we need to find an alternative method that is simple and efficient. After some attempts, I found Sequential Forward Selection(SFS) equipped with linear regression works quite well, SFS algorithm can be summarised in the following step:

(Suppose in the algorithm below, all the available features are stored in the set N. Further, we set a number k, which is less or equal to the number of elements in the set N. We call the label in a prediction task Y)

1. Begin with an empty set S.
2. If $k > 0$, select a feature X_i from N that maximises R^2 of the linear regression with regressors X_i and dependent variable Y. Add this X_i into the set S and remove X_i from the set N.
3. If $k > 1$, select a feature X_j from N that maximises R^2 of the linear regression with regressors X_j together with all the features in S. The dependent variable is still Y. Add this X_i into the set S and remove X_i from the set N.
4. If $k > 1$, repeat step 3 until the number of elements in N reaches k.

I implemented the SFS with the mlxtend library in Python, please refer to [15] for further details.

1.4.1 Feature selection for the direction of the next mid-price change prediction

The drawback of this type of method is apparent: I am also selecting features for recurrent neural networks, but the SFS method itself is not recurrent, in other words, the SFS method is not able to catch the recurrent pattern in the time series, but it is very commonly known that high-frequency stock data is correlated in time. However, this correlation is ignored in the method. Therefore, when it comes to the actual application, I choose to ask SFS to select several features, then I would adjust these features manually by testing on their alternatives, and make decisions based on the results of empirical trials and logical reasoning, in this way the selected features are more effective.

Now we can start the feature selection process, and I would first select features for the direction of the next mid-price change prediction. The available features for selection are the following:

- Bid price 1 to bid price 5,
- Bid size 1 to bid size 5,
- Ask price 1 to ask price 5,
- Ask size 1 to ask size 5,
- Direction of the current mid-price change,
- Time from the last mid-price change

As predictions are to be made once the mid-price changes, and the mid-price change happened at the current time step is called the current mid-price change. The last two features here are not directly shown in Figure 1.1, they are added according to my personal experience. In the stock market, it is often observed that directions of two consecutive mid-price changes are correlated. Besides, the time between current mid-price change and the last mid-price change would indicate the frequency of trading, if the trading activity is more frequent, then the features at levels closer to the level 1 might be given more importance. Therefore, these features are added as candidates for feature selection. The total number of candidates is now 22, as a general principle, roughly half of the selected features should be selected, so I have set the SFS to select 11 features. The result of SFS is:

- Ask size 1, bid size 1, bid price 1.
- Ask size 2, bid size 2, bid price 2.
- bid size 3. bid size 4, ask size 5, bid size 5.
- Direction of the current mid-price change.

The SFS ignores 'ask price 1' and 'ask price 2' but selects 'bid price 1' and 'bid price 2', this is not reasonable, due to the random nature of the stock market, and if we assume we are neutral about the future stock price, the bid side and the ask side should be given the same importance. So I would add the 'ask price 1' and 'ask price 2'. For the same reason, ask size 3 is added. 'Time from the last mid-price change' is not selected by the SFS and also in the empirical tests, and I do not see any noticeable performance improvements after adding the time from the last mid-price change, so I have abandoned it here. Also, bid size 4, ask size 5 and bid size 5 are abandoned since I don't see significant performance drop while not using them, and since they are relatively far away from the mid-price, the effect from them on the mid-price is usually limited. The final set of features selected are the following:

- Bid price 1, bid size 1, ask size 1, ask price 1.
- Ask price 2, bid size 2, bid price 2, ask size 2.
- Ask size 3, bid size 3,
- Direction of the current mid-price change,

1.4.2 Feature selection for prediction of time to the next mid-price change

To select features for predicting time to the next mid-price change. All the available features are the following:

- Bid price 1 to bid price 5.
- Bid size 1 to bid size 5.
- Ask price 1 to ask price 5.
- Ask size 1 to ask size 5.
- Time.
- Time from the last mid-price change.
- Time gaps between the last two mid-price changes.

It is doubtful that the direction of the last mid-price change can have any significant effect on when the next mid-price change would happen, so it is not considered in this task. However, in this task, predictions are made every time the LOB data updates, sometimes they are between the last mid-price change and the next mid-price change. So the time from the last mid-price change should be a proper feature candidate because it is more likely to see a mid-price change when time goes further away from the last mid-price change. Besides, the Autoregressive Conditional Duration model [16] suggests that two consecutive intertrade duration are correlated in some way.

Hence I believe that time between the last two mid-price changes are indicative for predicting when the next mid-price change would occur. The total number of candidates is 23, and I firstly use SFS to select 11 features, which are the following:

- Time,
- Ask size 1, bid price 1.
- Ask price 2, bid price 2.
- Ask size 4, bid size 4.
- Ask price 5, bid price 5,
- Time from the last mid-price change.
- Time between the last two mid-price changes.

This time the SFS pays more attention to price factors, that is reasonable, as the price could also have an impact on the mid-price moves, just as what I explained for Table 1.2. To balance the bid side and the ask side, I added 'ask price 1', 'bid size 1'. I also replaced 'ask size 4' and 'bid size 4' with 'ask price 3' and 'bid price 3', which are closer to the mid-price. Hence, the final result of feature selection becomes:

- Time,
- Bid price 1, bid size 1, ask size 1, ask price 1
- Ask price 2, bid price 2,
- Ask price 3, bid price 3,
- Ask price 5, bid price 5,
- Time from the last mid-price change.
- Time between the last two mid-price changes.

Chapter 2

Recurrent neural network and LSTM

2.1 Structure of RNN

A recurrent neural network is particularly suitable for sequential input, it uses internal states to store memories throughout sequential input. There are various ways to construct an RNN, the main difference would exist in how layers are connected with each other. For example, all the output of a layer is used as the input for the next layer in some cases, but sometimes only a part of the output becomes the input of the next layer. There are many types of RNNs, here I would only introduce the most common type, for further details, please refer to Goodfellow's Book [17, page 367-415]. To interpret the RNN model more easily, I made a diagram of a vanilla RNN for classification (see Figure 2.1), it demonstrates how the sequential data is processed in the RNN,

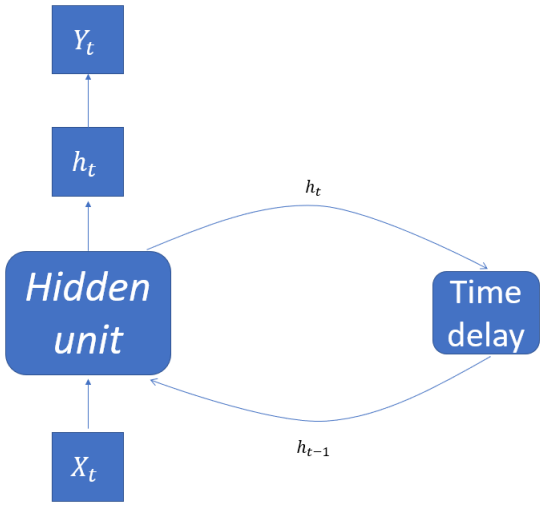

Figure 2.1: A vanilla RNN

There are three types of weight matrices in this RNN: input to hidden connections weight matrix U, hidden-to-hidden recurrent connections weight matrix W, and hidden-to-output connections weight matrix V. Suppose the activation function in the hidden unit is the hyperbolic tangent function, and the activation function for output is the softmax function. Then the following equations illustrate forward propagation of the vanilla RNN:

$$\boldsymbol{h}_t = \tanh\left(\boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}_{t-1} + \boldsymbol{U}\boldsymbol{X}_t\right),$$
$$\hat{\boldsymbol{Y}}_t = softmax(\boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}_t),$$

where the meaning of matrices U, V, and W have been introduced in the paragraph above, $\boldsymbol{b}$ and

$\boldsymbol{c}$ are bias vectors. Besides, $S(\cdot)$ is the softmax activation function, definitions of $S(\cdot)$ and $\tanh(\cdot)$ are the following:

$$S(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}},$$

where $i = 1, \ldots, M$ and $\mathbf{x}$ is an M-dimensional vector in $\mathbb{R}^M$. The $\tanh(\cdot)$ is

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} = \frac{e^x - e^{-x}}{e^x + e^{-x}},$$

where $x \in \mathbb{R}$.

2.2 Training of RNN

The training of the vanilla RNN usually involves a gradient-based technique, in which we have to compute all the necessary gradients in the network. For recurrent neural networks, a useful method called Backpropagation through time (BPTT) is often applied to compute the gradients. Let us briefly walk through it here. More details can be found in Gang Chen's paper [18].

We first suppose that the Loss function is set as cross entropy, which is defined as

$$L = -\sum_t Y_t \cdot \log \hat{Y}_t,$$

where Y_t is the label that we are trying to predict, $\hat{Y}_t$ is the prediction we made, as softmax activation function is used here, the $\hat{Y}_t$ and Y_t would be vectors of real numbers in $[0, 1]$.

Recall that Our target is to determine the parameters that minimise L. Then we need to obtain the derivative: $\frac{\partial L}{\partial m_t}$, where m_t is $c + Vh_t$, from Gang Chen's paper we can see:

$$\frac{\partial L}{\partial m_t} = \hat{Y}_t - Y_t$$

Then we can start finding derivatives with respect to W, as RNN uses the same W at every time step. If we only consider the derivative of the objective function at time step $t+1$, which is called L_{t+1}, and is defined by $-Y_{t+1} \log \hat{Y}_{t+1}$ then

$$\begin{aligned} \frac{\partial L_{t+1}}{\partial W} &= \frac{\partial L_{t+1}}{\partial \hat{Y}_{t+1}} \frac{\partial \hat{Y}_{t+1}}{\partial \mathbf{h}_{t+1}} \frac{\partial \mathbf{h}_{t+1}}{\partial W}. \\ &= \sum_{k=1}^{t} \frac{\partial L_{t+1}}{\partial \hat{Y}_{t+1}} \frac{\partial \hat{Y}_{t+1}}{\partial \mathbf{h}_{t+1}} \frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_k} \frac{\partial \mathbf{h}_k}{\partial W}, \end{aligned} \tag{2.2.1}$$

where we used backpropagation through time (BPTT) for the second equality. Moreover, summing up derivative at all the time steps can lead to the derivative with respect to W,

$$\frac{\partial L}{\partial W} = \sum_{t=1}^{N-1} \sum_{k=1}^{t+1} \frac{\partial L_{t+1}}{\partial \hat{Y}_{t+1}} \frac{\partial \hat{Y}_{t+1}}{\partial \mathbf{h}_{t+1}} \frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_k} \frac{\partial \mathbf{h}_k}{\partial W}.$$

where we assumed that $t = 1, ..., N$. Then we can focus on finding the derivative with respect to U. Similarly, we start with the derivative the last time step:

$$\frac{\partial L_{t+1}}{\partial U} = \frac{\partial L_{t+1}}{\partial \mathbf{h}_{t+1}} \frac{\partial \mathbf{h}_{t+1}}{\partial U} = \sum_{k=1}^{t+1} \frac{\partial L_{t+1}}{\partial \mathbf{h}_{t+1}} \frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_k} \frac{\partial \mathbf{h}_k}{\partial U}.$$

Summing up the derivatives at all the previous time steps:

$$\frac{\partial L}{\partial U} = \sum_{t=1}^{N-1} \sum_{k=1}^{t+1} \frac{\partial L_{t+1}}{\partial z_{t+1}} \frac{\partial z_{t+1}}{\partial \mathbf{h}_{t+1}} \frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_k} \frac{\partial \mathbf{h}_k}{\partial U}. \tag{2.2.2}$$

The derivatives with respect to c and V are relatively simple:

$$\frac{\partial L}{\partial c} = \sum_{t=1}^{N-1} \frac{\partial L}{\partial \hat{Y}_t} \frac{\partial \hat{Y}_t}{\partial c},$$

$$\frac{\partial L}{\partial V} = \sum_{t=1}^{N-1} \frac{\partial L}{\partial \hat{Y}_t} \frac{\partial \hat{Y}_t}{\partial V}.$$

2.2.1 Vanishing and exploding gradient

RNN is designed for sequential input, and it is expected to learn long-term dependence in the input sequence. While training RNN using BPTT with excessively long sequence, it is not surprising to encounter the vanishing or exploding gradients problem. This problem makes tuning the parameters of the earlier layers very difficult. To give a gentle explanation of the problem, we can think that computing gradients in BPTT require multiplication of many weight matrices because the same weight matrices are used at every time step, which is mainly from $\frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_k}$ in (2.2.2). To illustrate the problem further, suppose W is one of these weight matrices and it is diagonalizable, then it can be decomposed as:

$$\boldsymbol{W}^n = \left(\boldsymbol{Q}\,\mathrm{diag}(\boldsymbol{\lambda})\boldsymbol{Q}^{-1}\right)^n = \boldsymbol{Q}\,\mathrm{diag}(\boldsymbol{\lambda})^n\boldsymbol{Q}^{-1}.$$

where $\boldsymbol{\lambda}$ is a vector of eigenvalues.

Hence, if some of the eigenvalues λ_i are smaller than 1, the vanishing gradients problem appears and slows down the training process. If some of the eigenvalues λ_i are larger than 1, exploding gradients problem appears and make the training process unstable. Apart from that, the use of hyperbolic tangent activation function can lead to vanishing gradient problems as well. For more details, the reader may refer to Y.Bengio's paper [19]. and Goodfellow'book [20, page 396-399].

2.3 Structure of LSTM

Unlike a standard RNN unit, an LSTM unit often includes three gates: input gate, output gate and forget gate. With these gates, LSTM becomes capable of selecting what to input, what to output and what to forget, which is also one of the reasons why LSTM is better at recording long term memory. LSTM units are usually connected as an LSTM layer, the number of units in a layer is also the length of the input sequence. There could be several ways to control the output of an LSTM layer. Usually, people can choose to output only the hidden state at the last time step or the hidden states at every time step. These two patterns of output are called many-to-one and many-to-many, which can lead to a distinctive impact on the performance of models with LSTM layers, and it is investigated later in this chapter.

Let us firstly have a look at a standard version of LSTM unit [21]

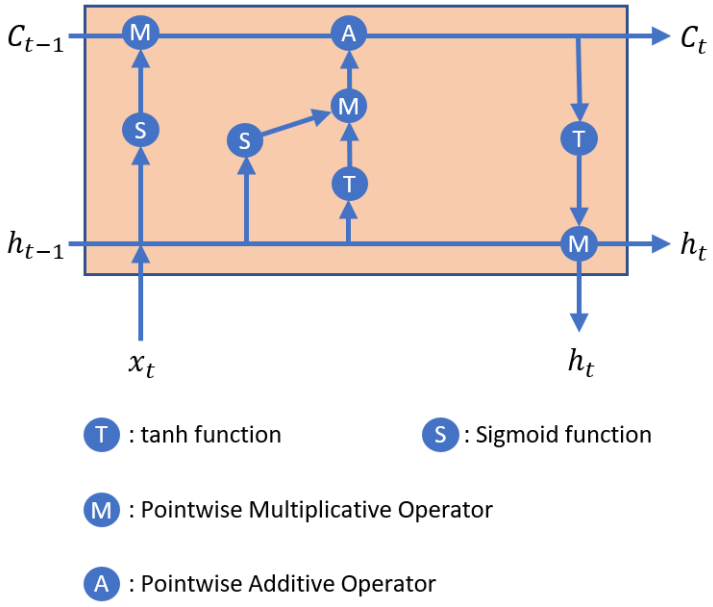

Figure 2.2: Structure of an LSTM unit.

The gates in Figure 2.2 can be described mathematically using the following equations:

$$\begin{aligned} i_t &= sigmoid\,(B_i + U_i x_t + V_i h_{t-1}) \\ f_t &= sigmoid\,(B_f + U_f x_t + V_f h_{t-1}) \\ o_t &= sigmoid\,(B_o + U_o x_t + V_o h_{t-1}) \end{aligned} \tag{2.3.1}$$

where sigmoid function is defined by:

$$sigmoid(a) = \frac{1}{1+e^{-a}} = \frac{e^a}{e^a+1}, \text{ where a} \in \mathbb{R}.$$

In the equations (2.3.1), f_i is the forget gate, which controls what is to be forgotten from the last cell state, i_t is the input gate, which determines what to be used from the input x at time step t, and o_i is the output gate, controlling the information passed from the cell state to hidden state. And B_i, B_f, B_o are bias vectors, U_i, U_f, U_o are the weight matricies connecting gates and the input x_t, V_i, V_f, V_o are the weight matrices connecting gates and hidden states.

In addition, the updating process of cell states and hidden states is the following:

$$\begin{aligned} \tilde{c}_t &= \tanh\,(U_c x_t + V_c h_{t-1} + B_c) \\ c_t &= f_t \circ c_{t-1} + i_t \circ \tilde{c}_t \\ h_t &= o_t \circ \tanh\,(c_t) \end{aligned} \tag{2.3.2}$$

where c_t is the cell state, which will usually not be the output of the LSTM unit directly, h_t is the hidden sates, though it is called hidden state, but normally we regard it as the output of the LSTM units. And $\circ$ is the element-wise multiplication.

Now we can describe the algorithm in an LSTM unit shown in Figure 2.2 by words: at time t, the unit input hidden state h_{t-1} and the external input x_t, then the unit computes the input gate, the output gate, and the forget gate based on h_{t-1} and x_t. On the other side, when the cell state c_{t-1} steps into the unit, the unit will use the computed forget gate to select the essential information it needs. And a new temporary cell state $\tilde{c}_t$ is computed based on x_t and h_{t-1}, then the temporary cell state $\tilde{c}_t$ will be passed through the input gate, then it meets the filtered c_{t-1}, after an addition, they combined into a new cell state c_t. Together with the output gate, the c_t will be used to compute the new hidden state h_t.

In Keras, LSTM is trained by a three-dimensional input, let us call the shape of input (x, y, z), x is the number of samples in the input, a sample has dimension (y, z), where y is the time steps contained in a sample, and z is the number of features fed into one unit of the first layer.

2.4 Training of LSTM

Training of LSTM requires backpropagation through time (BPTT) again, as more parameters are used in LSTM units, we would expect the backpropagation to be much more complicated. There are four groups of parameters that we need to update:

- input gate parameters: B_i, U_i, V_i.
- output gate parameters: B_o, U_o, V_o.
- forget gate parameters: B_f, U_f, V_f.
- hidden states and cell states related parameters: U_c, V_c, B_c.

To make the problem more practical, we can build a vanilla RNN with one LSTM layer and one output layer with softmax activation on the hidden states. In this case, two more equations need to be considered:

$$z_t = V_z h_t + B_z,$$

and

$$\hat{Y}_t = softmax(z_t),$$

where V_z and B_z are two more parameters needing to be updated. To trigger the backpropagation and make the derivation smoother, we need to define the following:

$$\begin{aligned}
m_{c,t} &= U_c x_t + V_c h_{t-1} + B_c, \\
m_{i,t} &= U_i x_t + V_i h_{t-1} + B_i, \\
m_{o,t} &= U_o x_t + V_o h_{t-1} + B_o, \\
m_{f,t} &= U_f x_t + V_f h_{t-1} + B_f.
\end{aligned}$$

we further suppose the loss function to be $L = -\sum_t Y_t \log \hat{Y}_t$. From previous section on RNN's backpropagation, we have learnt the derivation of softmax fucntion, which gives us:

$$\begin{aligned}
\frac{\partial L}{\partial z_t} &= \hat{Y}_t - Y_t, \\
\frac{\partial L}{\partial V_{z,t}} &= \frac{\partial L}{\partial z_t} \cdot \frac{\partial z_t}{\partial V_{z,t}} = \frac{\partial L}{\partial z_t} \cdot h_t^T, \\
\frac{\partial L}{\partial B_{z,t}} &= \frac{\partial L}{\partial z_t} \cdot \frac{\partial z_t}{\partial B_{z,t}} = \frac{\partial L}{\partial z_t}.
\end{aligned}$$

Computing gradient with respect to input gate parameters:

Since both L and i_t are dependent on c_t

$$\begin{aligned}
\frac{\partial L}{\partial i_t} &= \frac{\partial L}{\partial c_t} \cdot \frac{\partial c_t}{\partial i_t} = \frac{\partial L}{\partial c_t} \circ \hat{c}_t, \\
\frac{\partial L}{\partial m_{i,t}} &= \frac{\partial L}{\partial i_t} \cdot \frac{\partial i_t}{\partial m_{i,t}} = \frac{\partial L}{\partial c_t} \circ \hat{c}_t \circ \frac{\partial i_t}{\partial m_{i,t}} = \frac{\partial L}{\partial c_i} \circ \hat{c}_t \circ i_t \left(1 - i_t\right), \\
\frac{\partial L}{\partial U_{i,t}} &= \frac{\partial L}{\partial m_{i,t}} \cdot \frac{\partial m_{i,t}}{\partial U_{i,t}} = \frac{\partial L}{\partial m_{i,t}} \cdot x_t^T, \\
\frac{\partial L}{\partial V_{i,t}} &= \frac{\partial L}{\partial m_{i,t}} \cdot \frac{\partial m_{i,,t}}{\partial V_{i,t}} = \frac{\partial L}{\partial m_{i,t}} \cdot h_{t-1}^T, \\
\frac{\partial L}{\partial B_{i,t}} &= \frac{\partial L}{\partial m_{i,t}} \cdot \frac{\partial m_{i,t}}{\partial B_{i,t}} = \frac{\partial L}{\partial m_{i,t}},
\end{aligned}$$

where I used the fact that the derivative of sigmoid function $S(x)$ is $S(x)(1 - S(x))$

Computing gradient with respect to forget gate parameters:

Since both L and f_t are dependent on c_t

$$\begin{aligned}
\frac{\partial L}{\partial f_t} &= \frac{\partial L}{\partial c_t} \cdot \frac{\partial c_t}{\partial f_t} = \frac{\partial L}{\partial c_t} \circ c_{t-1}, \\
\frac{\partial L}{\partial m_{f,t}} &= \frac{\partial L}{\partial f_t} \cdot \frac{\partial f_t}{\partial m_{f,t}} = \frac{\partial L}{\partial c_t} \circ c_{t-1} \circ \frac{\partial f_t}{\partial m_{f,t}} = \frac{\partial L}{\partial c_t} \circ c_{t-1} \circ f_t \left(1 - f_t\right), \\
\frac{\partial L}{\partial U_{f,t}} &= \frac{\partial L}{\partial m_{f,t}} \cdot \frac{\partial m_{f,t}}{\partial U_{f,t}} = \frac{\partial L}{\partial m_{f,t}} \cdot x_t^T, \\
\frac{\partial L}{\partial V_{f,t}} &= \frac{\partial L}{\partial m_{f,t}} \cdot \frac{\partial m_{f,t}}{\partial V_{f,t}} = \frac{\partial L}{\partial m_{f,t}} \cdot h_{t-1}^T, \\
\frac{\partial L}{\partial B_{f,t}} &= \frac{\partial L}{\partial m_{f,t}} \cdot \frac{\partial m_{f,t}}{\partial B_{f,t}} = \frac{\partial L}{\partial m_{f,t}}.
\end{aligned}$$

Computing gradient with respect to output gate parameters:

Since both L and i_t are dependent on h_t,

$$\begin{aligned}
\frac{\partial L}{\partial o_t} &= \frac{\partial L}{\partial h_t} \cdot \frac{\partial h_t}{\partial o_t} = \frac{\partial L}{\partial h_t} \circ \tanh(c_t), \\
\frac{\partial L}{\partial m_{o,t}} &= \frac{\partial L}{\partial o_t} \cdot \frac{\partial o_t}{\partial m_{o,t}} = \frac{\partial L}{\partial h_t} \circ \tanh(c_t) \circ \frac{\partial o_t}{\partial m_o} = \frac{\partial L}{\partial h_t} \circ \tanh(c_t) \circ o_t \left(1 - o_t\right), \\
\frac{\partial L}{\partial U_{o,t}} &= \frac{\partial L}{\partial m_{o,t}} \cdot \frac{\partial m_{o,t}}{\partial U_{o,t}} = \frac{\partial L}{\partial m_{o,t}} \cdot x_t^T, \\
\frac{\partial L}{\partial V_{o,t}} &= \frac{\partial L}{\partial m_{o,t}} \cdot \frac{\partial m_{o,t}}{\partial V_{o,t}} = \frac{\partial L}{\partial m_{o,t}} \cdot h_{t-1}^T, \\
\frac{\partial L}{\partial B_{o,t}} &= \frac{\partial L}{\partial m_{o,t}} \cdot \frac{\partial m_{o,t}}{\partial B_{o,t}} = \frac{\partial L}{\partial m_{o,t}}.
\end{aligned}$$

The situation for cell states and hidden states parameters are slightly different: For $t < T$,

$$\begin{aligned}
\frac{\partial L}{\partial c_t} &= \frac{\partial L}{\partial h_t} \cdot \frac{\partial h_t}{\partial c_t} + \frac{\partial L}{\partial c_{t+1}} \frac{\partial c_{t+1}}{\partial c_t} = \frac{\partial L}{\partial h_t} \circ o_t \circ \left(1 - \tanh\left(c_t\right)^2\right) + \frac{\partial L}{\partial c_{t+1}} f_{t+1}, \\
\frac{\partial L}{\partial h_t} &= \frac{\partial L}{\partial o_{t+1}} \frac{\partial o_{t+1}}{\partial h_t} + \frac{\partial L}{\partial i_{t+1}} \frac{\partial i_{t+1}}{\partial h_t} + \frac{\partial L}{\partial f_{t+1}} \frac{\partial f_{t+1}}{\partial h_t} + \frac{\partial L}{\partial \hat{c}_{t+1}} \frac{\partial \hat{c}_{t+1}}{\partial h_t} + \frac{\partial L}{\partial z_t} \frac{\partial z_t}{\partial h_t}, \\
&= o_{t+1} \left(1 - o_{t+1}\right) V_{o,t+1} \frac{\partial L}{\partial o_{t+1}} + i_{t+1} \left(1 - i_{t+1}\right) V_{i,t+1} \frac{\partial L}{\partial i_{t+1}} + f_{t+1} \left(1 - f_{t+1}\right) V_{f,t+1} \frac{\partial L}{\partial f_{t+1}}, \\
&+ \frac{\partial L}{\partial \hat{c}_{t+1}} \frac{\partial \hat{c}_{t+1}}{\partial h_t} + V_{z,t} \frac{\partial L}{\partial z_t}.
\end{aligned}$$

For $t = T$,

$$\begin{aligned}
\frac{\partial L}{\partial h_t} &= \frac{\partial L}{\partial z_t} \frac{\partial z_t}{\partial h_t} = V_z \frac{\partial L}{\partial z_t}, \\
\frac{\partial L}{\partial c_t} &= \frac{\partial L}{\partial h_t} \cdot \frac{\partial h_t}{\partial c_t} = \frac{\partial L}{\partial h_t} \circ o_t \circ \left(1 - \tanh\left(c_t\right)^2\right),
\end{aligned}$$

where I used the fact that the first order derivative of $f(x) = \tanh(x)$ is $1 - \tanh(x)^2$.

So now we have

$$\begin{aligned}
\frac{\partial L}{\partial \hat{c}_t} &= \frac{\partial L}{\partial c_t} \cdot \frac{\partial c_t}{\partial \hat{c}_t} = \frac{\partial L}{\partial c_t} \circ i_t, \\
\frac{\partial L}{\partial m_{c,t}} &= \frac{\partial L}{\partial \hat{c}_t} \cdot \frac{\partial \hat{c}_t}{\partial m_{c,t}} = \frac{\partial L}{\partial c_t} \circ i_t \circ \frac{\partial \hat{c}_t}{\partial m_{c,t}}, \\
&= \frac{\partial L}{\partial c_c} \circ i_t \circ \left(1 - \tanh\left(m_c\right)^2\right) = \frac{\partial L}{\partial c_c} \circ i_t \circ \left(1 - \hat{c}_t^2\right), \\
\frac{\partial L}{\partial U_{c,t}} &= \frac{\partial L}{\partial m_{c,t}} \cdot \frac{\partial m_{c,t}}{\partial U_{c,t}} = \frac{\partial L}{\partial m_{c,t}} \cdot x_t^T, \\
\frac{\partial L}{\partial V_{c,t}} &= \frac{\partial L}{\partial m_{c,t}} \cdot \frac{\partial m_{c,t}}{\partial V_{c,t}} = \frac{\partial L}{\partial m_{c,t}} \cdot h_{t-1}^T, \\
\frac{\partial L}{\partial b_{c,t}} &= \frac{\partial L}{\partial m_{c,t}} \cdot \frac{\partial m_{c,t}}{\partial b_{c,t}} = \frac{\partial L}{\partial m_{c,t}}.
\end{aligned}$$

Note that the backpropagation steps shown above are just for one time step, to calculate the gradients throughout the time series, we need to aggregate all the gradients, for example:

$$\frac{\partial L}{\partial V_o} = \sum_t \frac{\partial L}{\partial V_{o,t}}, \quad \frac{\partial L}{\partial U_i} = \sum_t \frac{\partial L}{\partial U_{i,t}}.$$

The gradients calculation for the other parameters follows the same pattern, which will not be repeated here.

2.5 Optimizers

Through backpropagation, we can compute gradients of the loss function with respect to parameters, but to make use of these gradients, we need an optimiser. An optimiser is an algorithm that

determines how the computed gradients are used to change the parameters in the neural networks, there are many commonly used optimisers such as Momentum, SGD, RMSprop, Adam, Adadelta, Adagrad, Adamax, Nadam, Ftrl. In this part, I would like to give some introduction to some of these optimisers and explain why I would choose to use Adam as the optimiser for the models in the thesis.

2.5.1 Gradient Descent

Gradient decent is the most basic optimizer, it uses the first order derivative of the loss function to help it reach a local minima, Since gradient descent is very basic, we only need one equation to define it:

$$\boldsymbol{\alpha} = \boldsymbol{\alpha} - \boldsymbol{\eta} \cdot \nabla \mathbf{L}(\boldsymbol{\alpha}) = \boldsymbol{\alpha} - \frac{\eta}{N} \sum_{i=1}^{N} \nabla \mathbf{L}_i(\boldsymbol{\alpha}),$$

where $\boldsymbol{\alpha}$ is the weight parameter, η is the learning rate and $\mathbf{L}$ is the loss function, and $\mathbf{L}_i(\boldsymbol{\alpha})$ is the value of the loss function calculated on the i -th sample. By looking at the definition, it is not difficult to find why it is the basic, it uses constant learning rate, with a simple linear pattern for parameters update. This setting makes gradient descent approachable and fast, but the drawbacks are evident, due to its constant learning rate, it often stops at a local minimum, and the algorithm is applied on the whole training dataset. Therefore, it could be extremely time-consuming when the training dataset is large.

2.5.2 Stochastic Gradient Descent

Stochastic gradient descent is a modification of the naive gradient descent algorithm. The only difference is now gradients are updated after calculating the gradients based on one single sample.

$$\boldsymbol{\alpha} = \boldsymbol{\alpha} - \eta \cdot \nabla \mathbf{L}_i(\boldsymbol{\alpha}),$$

where $\boldsymbol{\alpha}$ is the weight parameter, η is the learning rate and $\mathbf{L}$ is the loss function, and $\mathbf{L}_i(\boldsymbol{\alpha})$ is the value of the loss function calculated on the i-th sample. Since now the parameters are updated for every sample, the frequency of updating is much higher, which makes it quicker to reach convergence, but the parameters may experience higher variances.

2.5.3 Mini-Batch Gradient Descent

Mini-batch gradient descent algorithm is developed to combine the advantages of the stochastic gradient descent and original gradient descent. It firstly divides all the samples into several batches, then updates parameters after calculating gradients based on each batch,

$$\boldsymbol{\alpha} = \boldsymbol{\alpha} - \eta \cdot \nabla \mathbf{L}_i(\boldsymbol{\alpha}),$$

where $\boldsymbol{\alpha}$ is the weight parameter, η is the learning rate and $\mathbf{L}$ is the loss function, and $\mathbf{L}_i(\boldsymbol{\alpha})$ is the value of the loss function calculated on the i-th batch. Although this algorithm has many advantages, its drawback is still evident: all the parameter are sharing the same learning rate, which is not preferable as some parameters might need to change more drastically than others and it is sometimes difficult to find a suitable learning rate.

2.5.4 Momentum

In the momentum optimisation algorithm, we are not only using the current gradient but also the gradient computed at the last iteration. The algorithm can be explained in two equations:

$$\begin{aligned} \boldsymbol{\alpha} &= \boldsymbol{\alpha} - \boldsymbol{\delta}_t \\ \boldsymbol{\delta}_t &= \gamma \boldsymbol{\delta}_{t-1} + \eta \nabla \mathbf{L}_i(\boldsymbol{\alpha}) \end{aligned}$$

where $\boldsymbol{\alpha}$ is the weight parameter, η is the learning rate and $\mathbf{L}_i(\boldsymbol{\alpha})$ is the value of the loss function calculated on the i-th sample. The name of momentum optimiser is coming from using the gradient of the previous iteration, which would accelerate the process of approaching the minima. Parameters trained with momentum optimiser have lower variance, but since there is one more hyperparameter here now, it might be difficult to set the value of γ.

2.5.5 Adagrad

Adagrad is a more advanced algorithm as it uses variable learning rate, it is suitable for both convex and non-convex problems. To describe the algorithm we need two equations:

$$G = \sum_{i=1}^{t} g_i g_i^\top$$

$$\boldsymbol{\alpha}_j := \boldsymbol{\alpha}_j - \frac{\eta}{\sqrt{G_{j,j}}} g_j$$

where $g_i = \nabla \mathbf{L}_i(w)$ is the gradient computed at iteration i. G is a matrix that sums up all the outer products of the gradient at each iteration. Furthermore, $\boldsymbol{\alpha}$ is the weight parameter, η is the learning rate. Since the actual learning rate is controlled by G, which varies at each epoch, the learning rate changes after every epoch, besides, the learning rate in Adagrad differs for different parameters as well, parameters that get more updates will usually have a lower learning rate.

2.5.6 Adam

The full name of Adam is Adaptive Moment Estimation [22], To implement Adam algorithm, we need first two quantities: decaying avearage of gradients $m^{(t+1)}$ and decaying average of squared gradients $v^{(t+1)}$

$$\begin{aligned} m^{(t+1)} &= \beta_1 m^{(t)} + (1-\beta_1)\, \nabla_{\boldsymbol{\alpha}} \mathbf{L}^{(t)} \\ v^{(t+1)} &= \beta_2 v^{(t)} + (1-\beta_2) \left(\nabla_{\boldsymbol{\alpha}} \mathbf{L}^{(t)} \right)^2 \end{aligned} \tag{2.5.1}$$

After obtaining the $m^{(t+1)}$ and $v^{(t+1)}$, we can move forward to adjusting them, then use the adjusted $\hat{m}$ and $\hat{v}$ to write down the recursive equation for $\boldsymbol{\alpha}$ updates:

$$\begin{aligned} \hat{m} &= \frac{m^{(t+1)}}{1-\beta_1^{t+1}} \\ \hat{v} &= \frac{v^{(t+1)}}{1-\beta_2^{t+1}} \\ \boldsymbol{\alpha}^{(t+1)} &= \boldsymbol{\alpha}^{(t)} - \eta \frac{\hat{m}}{\sqrt{\hat{v}} + \epsilon} \end{aligned} \tag{2.5.2}$$

where $\boldsymbol{\alpha}$ is the weight parameter, η is the learning rate and $\mathbf{L}$ is the value of the loss function calculated at iteration t, and β_1, β_2, ϵ are hyperparameters that are used to tune the learning rate. The paper written by Diederik and Jimmy [22] suggests that β_1 can be set as 0.9, β_2 can be set as 0.999 and ϵ can be set as 10^8. As the full name of Adam indicates, it uses not only gradients but also the second moment of gradients. According to the paper [22], we can see that Adam has various advantages: it is fast and efficient, it requires a little memory to run, and every parameter has its customized learning rate. In particular, Diederik and Jimmy also demonstrated the better performance of Adam in comparison to most of the other optimizers in the same paper. And because of the superiority of Adam, it is used as the optimiser for all the models in the thesis.

Chapter 3

Predicting the direction of the next mid-price change

3.1 Problem description

In this chapter, the main task is to predict the direction of the next change of mid-price (which will also be called as a label in the rest of the chapter). This is a basic research in market microstructure, which helps us understand the price formation mechanism and the dependence between the labels and the past LOB information in the stock market. In this task, I would examine the performance of four types of models: Feedforward Neural Network(FNN) with lagged features, FNN without lagged features, many-to-one RNN with LSTM layers, many-to-many RNN with LSTM layers. The main purpose of building Feedforward neural networks(FNNs) with four dense layers is to create a benchmark, though these FNN models are not recurrent, which might be a huge disadvantage for them, but via adjusting the structure of the input data, I partially brought some of the recurrent information into the modified FNN models to enhance its performance. More details on the input data reconstruction can be found in the following sections.

The models are trained by high-frequency data of three stocks ('GOOG', 'JPM', 'WMT') from 01/05/2020 to 10/05/2020 and tested by data of the same stocks from 12/05/2020 to 13/05/2020, more details about the data can be found in chapter 1. I only used data of a few days, but the data is high-frequency trading data, and the data of a single stock in one day could contain more than 100,000 rows, so the reader should not be worried about the sample size.

As we are tackling a binary classification task, the most important thing is to check whether the ground-truth labels and the predicted labels are the same. Furthermore, after checking the data, I find the 0's and 1's have almost equal proportion in the training and test dataset. The performance measure that I would use is the accuracy on the test set, which is defined as:

$$\text{test\_accuracy} = \frac{\text{number of correct predictions}}{\text{total number of prediction}}.$$

A common loss function for binary classification is the binary cross entropy loss, minimising binary cross entropy loss is equivalent to finding the maximum likelihood. The maximum likelihood estimators are often favoured due to its excellent asymptotic properties, such as consistency and asymptotic normality. Hence, the Loss function selected for this task is binary cross entropy loss:

$$\text{BCL} = -\frac{1}{N}\sum_{i=1}^{N} Y_i \cdot \log \hat{Y}_i + (1 - Y_i) \cdot \log\left(1 - \hat{Y}_i\right),$$

where Y is the actual label and $\hat{Y}$ is the predicted label, and N is the number of samples.

To examine the difference among networks with different configurations, **I would like to introduce the 'units', inherited from Keras, it usually appears in the tables in the thesis.** ,

Remark 3.1.1 ('units')**.** 'units' has different meanings at different layers, but it always represents the number features used for prediction, and please note it is not the 'unit' in 'LSTM unit' or 'Dense unit'. In LSTM layers, 'units' is the dimensionality of the output space of an LSTM unit, in dense layers, 'units' would be the number of dense units in the dense layer.

I would set "units" to be 20, 35, 50, which could potentially lead to a quite different result, simply because the complexity of networks is not at the same level. People might think larger 'units' should always be better, but in reality, while training the model, we might experience overfitting which leads the performance on the test set undesirable, and another problem is lack of computational power, which makes the training process nearly endless. So there is a tradeoff between time spent on training and actual performance. I am aiming to find a proper configuration that works both fast and accurately.

The other important variable throughout the chapter is called 'time steps', which is the length of the input sequence for the RNN models (for FNN, it is slightly different). As claimed by Sirignano and Cont [13], there exists long-term dependence between the labels and the past LOB information, which means LOB data recorded a long time ago could affect the current trading activity significantly, in our task, this should be shown by better performance in longer time steps and better performance while using RNN. Therefore, I would test models (if applicable) with four types of input with at least three different time steps: 20, 50, 100. Hopefully, we would find out whether the models can dig out the dependence between the labels and the past LOB information.

I also want to introduce the definitions of the universal model and stock-specific model:

Definition 3.1.2 (Universal model)**.** Universal models are models trained based on trading data across multiple stocks, and the more stocks are used to train the data, the more universal the models are. Moreover, in theory, a universal model can work for any stocks.

Definition 3.1.3 (Stock specific model)**.** Stock specific models are models trained based on trading data of a single stock, which will only work for the stock used for training the models.

By these two definitions, all the models used in sections 3.2, 3.3, 3.4 are stock-specific models, but in section 3.5, we are going to try a universal model to check if there exists a universal price formation mechanism in the stock market by using limited data.

3.2 Feedforward neural network

The structure of Feedforward neural networks are relatively straightforward. It usually consists of three types of layers: input layer, hidden layer, and output layer. More details of FNN can be found in Goodfellow's book [23, page 164-223]. Here I plotted a simple figure to illustrate the structure of a vanilla FNN in Figure 3.1.

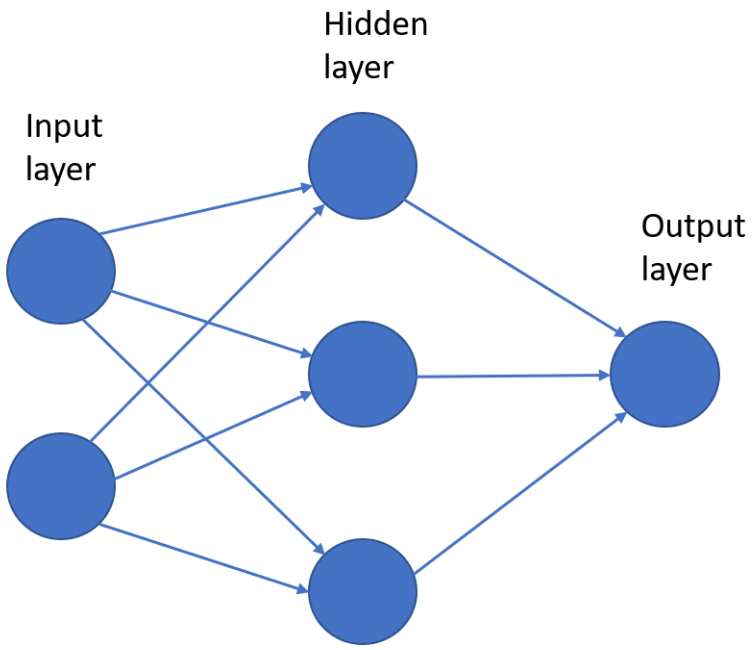

Figure 3.1: A three-layer FNN.

The Figure 3.1 is only indicative, where activations and number of units per layer are not specified, hence the actual structure of the FNN in the task is shown in the table 3.1, note x is a variable that will be set below in specific tasks. Note that 'units' would be the number of dense units in the dense layer.

	units	Type of layer
layer 1	x	Dense with ReLU activation
layer 2	x	Dense with ReLU activation
layer 3	x	Dense with ReLU activation
layer 4	1	Dense with sigmoid activation

Table 3.1: FNN structure for chapter 3

3.2.1 FNN without lagged features

I would first try to build a basic model, which entirely relies on the most recently available information. The features used in the model are the following:

- Bid price 1, bid size 1, ask size 1, ask price 1.
- Ask price 2, bid size 2, bid price 2, ask size 2.
- Ask size 3, bid size 3,
- Direction of the current mid-price change,

The purpose of building such a model is to check whether the mid-price changes are in a markovian style, that is, any current mid-price changes only depends on things events from the previous time

step. If that is true, the performance of this model should be the same as models that use past limit order book information. The structure of the FNN is the same as Table 3.1, and the results are shown in Table 3.2.

Stock / units	GOOG	WMT	JPM
20	66.9%	62.6%	66.5%
35	66.9%	62.6%	66.5%
50	67.2%	62.6%	66.7%

units	Average
20	65.3%
35	65.3%
50	65.5%

Table 3.2: Test accuracy of FNN without lagged features

Note that 'units' is the number of dense units in the dense layer. According to the large-tick effect [24], as the share price for GOOG is quite large, which is over $1000 in the last three months, so there is less useful information concentrated in the first five price levels. Hence it should be harder to predict. However, that is not the case in this model, and it may be because GOOG has a higher dependence between the features so that only using current features would already give us a good result. Probably this model is not the best, and more tests will be done in the following parts.

3.2.2 FNN with lagged features

In the last subsection, I tried FNN without lagged features, the FNN does seem to perform well in the task, but now I want to add some lagged features into the training process of FNN. Furthermore, see if it is possible to catch the dependence between the labels and the past LOB information with the modified features.

Since FNN does not work in a recurrent style, some adjustments are made on the features to incorporate historical limit order book information into the model training. Note that in an RNN with LSTM layers, only the most recent direction of mid-price change is used at each time step, but due to the recurrent nature of RNN, every output is computed based on directions from all the previous time steps within the same sample. However, FNN does not have this advantage as it is not recurrent, and the input for training FNN is two-dimensional with shape (x, y), where x is the number of samples in the input, and y is the number of features in each sample, where a sample is one-dimensional. In contrast, the samples for RNN are two-dimensional, so that the previous directions are stored in one sample but at different time steps.

To make the FNN comparable to RNN, the length of FNN input is extended in the following way: Set the value of 'time steps' as N at first, then the direction of mid-price changes in the last $N - 1$ time steps are added to the one-dimensional sample. The reason for doing so is that the direction of recent mid-price changes is often considered as the most important feature in this kind of task. Hopefully, this will make the FNN model more competitive and help us catch the dependence the labels and the past LOB information. As a result, the following features are selected for this FNN:

- Bid price 1, bid size 1, ask size 1, ask price 1.
- Ask price 2, bid size 2, bid price 2, ask size 2.
- Ask size 3, bid size 3,
- direction of the current mid-price change,
- Directions of the last $N - 1$ mid-price changes,

The experiments are done with three stocks with four different time steps and three different model configurations. All the performance is in the table below:

(Note that in table 3.3, 3.4, 3.5, the value of 'time steps' means the number of previous directions of mid-price change included, which is also the N above, and 'units' is the number of units in the dense layer.)

Time steps \ Stock	GOOG	WMT	JPM
5	75.0%	63.8%	66.7%
20	76.1%	63.8%	68.2%
50	76.1%	63.6%	67.8%
100	75.7%	63.4%	67.9%

Time steps	Average
5	68.5%
20	69.4%
50	69.2%
100	69.0%

Table 3.3: Accuracy on test set of FNN with lagged features (units=20)

Time steps \ Stock	GOOG	WMT	JPM
5	75.3%	63.9%	67.1%
20	76.2%	63.8%	68.0%
50	75.9%	63.7%	67.9%
100	75.8%	63.4%	67.5%

Time steps	Average
5	68.8%
20	69.3%
50	69.2%
100	68.9%

Table 3.4: Accuracy on test set of FNN with lagged features (units=35)

Time steps \ Stock	GOOG	WMT	JPM
5	75.2%	64.0%	67.2%
20	76.1%	63.9%	67.7%
50	76.1%	63.4%	67.6%
100	75.7%	63.4%	67.8%

Time steps	Average
5	68.8%
20	69.2%
50	69.0%
100	69.0%

Table 3.5: Accuracy on test set of FNN with lagged features (units=50)

From tables 3.3, 3.4, 3.5, we can firstly say the average test accuracy is consistently above 68%, the best average test accuracy is found at $x = 20$ and 'time steps'=20, which is 69.4%. So the overall performance is better than the FNN without lagged features, especially when 'time steps' is 20. Besides, the performance difference among models with a different number of units in each layer is bearly noticeable. The performance changes as 'time steps' changes, the best performance is achieved at 'time steps' equal to 20 in most of the cases. When 'time steps' increases to 50, we will usually see a slight drop in test accuracy (this is how I call the accuracy on the test dataset.), this phenomenon reflects that FNN is not very good at dealing with very long input. Another thing to notice is that GOOG maintains a quite high test accuracy, while WMT is relatively not easy to predict.

Furthermore, due to the performance improvement compared to FNN without lagged features, the FNN with lagged features does catch some of the dependence. However, at the moment, we are still not knowing how good the improvement is, we will further investigate it by using RNN with LSTM layers, as they are often claimed to be very good at capturing dependence in time series.

3.3 RNN with LSTM layers (many to one)

In Keras setting, we can select whether to return sequences for each LSTM layer. To connect two LSTM layers, as a common practice, the input of the second layer would be the same as the output of the first layer. Hence I choose to return sequences for the first two LSTM layers in my networks, but the third LSTM layer needs to pass its output to a dense layer, which accepts input of any length. After stacking the first three LSTM layers, the output of the third LSTM layer leads to two types of RNN network: many-to-many RNN and many-to-one RNN.

In the many-to-one RNN model, the third LSTM layer only passes the output at the last time step to the dense layer. Then the output was activated by a dense layer with 1 unit, output of the dense layer would be the final output. In other words, if the sequence has T time steps, the network is only outputting the prediction at time step T. To illustrate how the LSTM units and dense units are connected in the many-to-one RNN model, I plotted Figure 3.2.

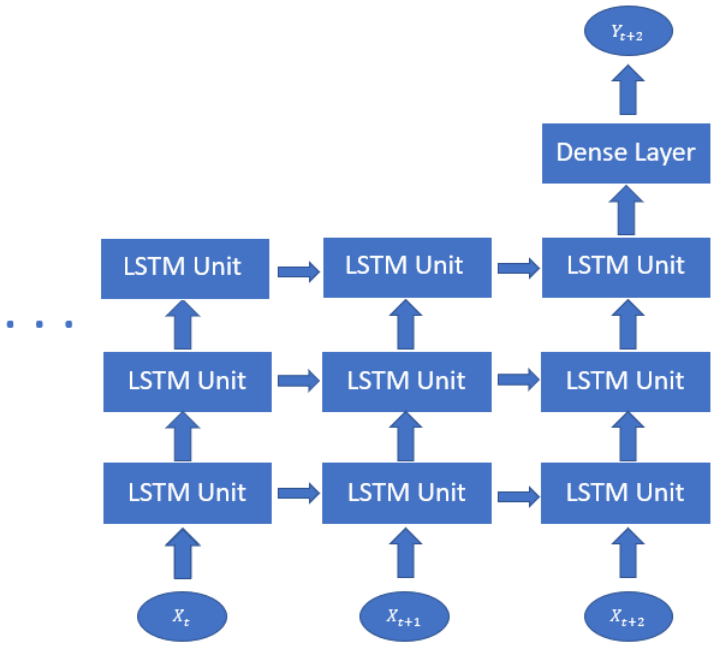

Figure 3.2: Many-to-one RNN with LSTM layers

To be more precise, the actual configuration and structure of the RNN are described by the table 3.6, where x is a variable that would be set when needed. Moreover, note that in LSTM layers, 'units' is the dimensionality of the output space of an LSTM unit, in normal dense layers, 'units' is the number of dense units in the dense layer.

	units	type of layer
layer 1	x	LSTM
layer 2	x	LSTM
layer 3	x	LSTM
layer 4	1	dense with sigmoid activation

Table 3.6: Structure of RNN with LSTM layers(many-to-one)

This time I am dealing with RNN, the input for RNN is three-dimensional with a shape (x, y, z), where x is the number samples, y is the number of time steps in a sample, z is the number of features used at every time step. Hence, I only need to put the features at the corresponding time step and let the RNN itself learn the dependence among these time steps. Consequently, the features used at each time step of the many-to-one RNN is the following:

- Bid price 1, bid size 1, ask size 1, ask price 1.
- Ask price 2, bid size 2, bid price 2, ask size 2.
- Ask size 3, bid size 3,
- Direction of the current mid-price change.

Now we can start the experiments to test the performance of the many-to-one RNN, and the three-layer structure is inherited from the paper [13], which has been proved to be desirable in solving such problems. The performances are recorded by three tables, each of them represents the model performance with a specific 'units' value. All the performances are recorded in the tables 3.7, 3.8, 3.9. (Note that in LSTM layers, 'units' is the dimensionality of the output space of an LSTM unit,

in normal dense layers, 'units' is the number of dense units in the dense layer. And 'time steps' is the length of the input sequence of the RNN.)

Time steps \ Stock	GOOG	WMT	JPM
5	67.2%	48.6%	50.4%
20	71.9%	63.2%	64.7%
50	72.5%	64.0%	65.8%
100	70.8%	63.4%	66.7%

Time steps	Average
5	55.4%
20	66.6%
50	67.4%
100	67.0%

Table 3.7: Accuracy of RNN with LSTM layers (many-to-one, units=20) on test set

Time steps \ Stock	GOOG	WMT	JPM
5	67.4%	50.1%	50.4%
20	71.9%	62.5%	65.3%
50	73.2%	63.7%	66.9%
100	74.9%	63.9%	67.7%

Time steps	Average
5	56.0%
20	66.6%
50	67.9%
100	68.8%

Table 3.8: Accuracy of RNN with LSTM layers (many-to-one, units=35) on test set

Time steps \ Stock	GOOG	WMT	JPM
5	66.1%	49.3%	50.4%
20	73.7%	63.2%	66.8%
50	73.2%	64.2%	67.4%
100	74.8%	63.5%	62.1%

Time steps	Average
5	55.3%
20	67.9%
50	68.3%
100	66.8%

Table 3.9: Accuracy of RNN with LSTM layers (many-to-one, units=50) on test set

The best average performance of the many-to-one RNN with LSTM layers happens when 'units' is 35, and 'time steps' is 100, where the accuracy on the test set reaches 68.8%. But the trend appears when 'units'=35 makes want to know what can happen if I extend the time steps further. After an extra round of test, I found if 'units' is 35 and 'time steps' is 125, 150, 200, respectively. Then the average test accuracies are 66.3%, 67.2%, 67.4%, lower than the previous record. In comparison, the best average accuracy on test set achieved by the FNN with lagged features is 69.4%, which is higher than this model. If we take a look at the performance for these two models at every different 'units' and 'time steps', we can see the FNN with lagged features is always the winner. Hence I think the many-to-one RNN is slightly worse than FNN with lagged features in predicting the direction of the next mid-price change and capturing the dependence between the labels and the past LOB information. Moreover, the training process for the many-to-one RNN is more time consuming, that would be another disadvantage of the many-to-one RNN.

3.4 RNN with LSTM layers (many to many)

In the many-to-many RNN, the third LSTM layer only passes the output at every time step to the dense layer. The output at every time step is activated by the same dense layer separately, the output of the dense layer would still be the final output. The many-to-many structure is believed to be more efficient than the many-to-one structure, as it makes use of the label at every time step while training with a sample. In contrast, the many-to-one model only uses the label at the last time step while training with a sample. To illustrate the pattern of how the layers are connected within the many-to-many RNN model, I made Figure 3.3.

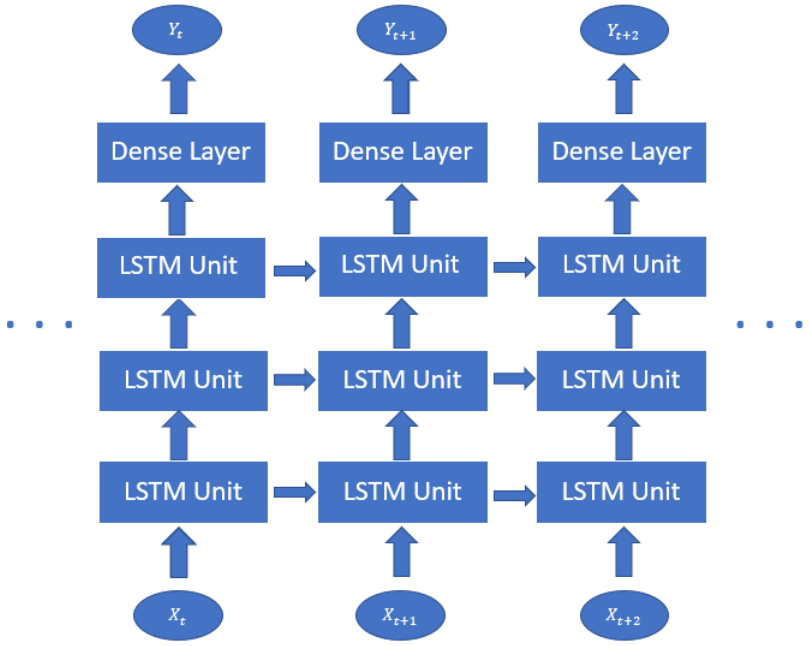

Figure 3.3: Many-to-many RNN with LSTM layers.

The table 3.10 provides another angle to view the structure of the many-to-many RNN.

	units	type of layer
layer 1	x	LSTM
layer 2	x	LSTM
layer 3	x	LSTM
layer 4	1	Dense with sigmoid activation

Table 3.10: Structure of RNN with LSTM layers(many-to-many)

The features used in the many-to-many RNN are the same as those used in the many-to-one RNN, so they are not repeated here. The results are in the tables 3.11, 3.12, 3.13 below:

Time steps \ Stock	GOOG	WMT	JPM
5	73.8%	63.9%	66.7%
20	76.0%	64.4%	68.4%
50	76.7%	64.6%	68.3%
100	77.1%	64.8%	68.3%

Time steps	Average
5	68.1%
20	69.6%
50	69.9%
100	70.1%

Table 3.11: Accuracy of RNN with LSTM layers (many-to-many, units=20) on test set

Time steps \ Stock	GOOG	WMT	JPM
5	74.1%	63.9%	67.5%
20	76.4%	64.4%	68.1%
50	76.9%	64.8%	68.2%
100	77.2%	64.8%	68.2%

Time steps	Average
5	68.5%
20	69.6%
50	70.0%
100	70.1%

Table 3.12: Accuracy of RNN with LSTM layers (many-to-many, units=35) on test set

Stock / Time steps	GOOG	WMT	JPM
5	74.0%	64.0%	67.0%
20	76.4%	64.6%	68.2%
50	76.9%	64.7%	68.3%
100	77.4%	64.8%	68.5%

Time steps	Average
5	68.3%
20	69.7%
50	70.0%
100	70.2%

Table 3.13: Accuracy of RNN with LSTM layers (many-to-many, units=50) on test set

Apparently, the many-to-many RNN is the winner out of the four candidates, it has the best average accuracy on the test set, the highest average accuracy on the test set was achieved when 'units' is 50, 'time steps' is 100, and it is 70.2%. The superior performance might benefit from more labels are used to tune parameters at each sample. As the performance seems to get better when the values of'time steps' and 'units' increases, The reader might wonder what would happen if I extend the time steps and the units further, I tried it and summarized the result in Table 3.14:

time steps / units	150	200
75	70.2%	70 %
100	70.1%	70.1%

time steps / units	125	150
50	70.2%	70.2 %

Table 3.14: Average Accuracy on test set(extra test)

From Table 3.14 we can see there is no improvement in performance when I increase 'units' to 75, 100 and increase 'time steps' to 150, 200, the best average test accuracy is still 70.2%, if I extend the time steps only and keep 'units' as 50, both average test accuracies are 70.2%. Hence the best average test accuracy for this model should be very close to 70.2%.

3.5 Universality test

In the paper [13], Justin Sirignano and Rama Cont discovered that there exists a universal price formation mechanism, which is supported by their universal model, the universal model consistently beats stock-specific models built in the paper. However, the universal model was trained by 500 stocks data over a long range of time with a superior computation power, which is not feasible for every researcher. So I am going to build a similar but less complicated model with less data, and see if the simpler model can achieve a similar result. As it has been proved in the previous sections that many-to-many models are the most efficient and accurate, the universal model is also many-to-many here, the structure is the same as to Figure 3.3. The actual structure is showing in the table 3.15 below.

	units	type of layer
layer 1	x	LSTM
layer 2	x	LSTM
layer 3	x	LSTM
layer 4	1	Dense with sigmoid activation

Table 3.15: Structure of RNN with LSTM layers(many-to-many, universal)

To start the training process, I gathered nine stocks' data collected from serveral trading days, The nine stocks are:

'ACN', 'ADBE', 'F', 'FTI', 'GE', 'IVZ', 'KIM', 'LIN', 'UNH'.

The corresponding trading dates are:

'02/01/2020', '03/02/2020', '02/03/2020', '01/06/2020', '01/06/2020', '01/04/2020', '01/07/2020',

'01/04/2020', '01/07/2020'.

The trading data of 9 stocks is the training data for the universal model. For the test, there will be 3 stocks in total: 'GOOG', 'WMT', 'JPM', and all the test data is collected from '12/05/2020' and '13/05/2020'. In addition, the structure of the data used to train and validate the models is the same as described in Chapter 1.

Stock / Time steps	GOOG	WMT	JPM
5	64.2%	62.2%	66.3%
20	67.9%	61.7%	66.1%
50	70.2%	61.9%	66.3%
100	64.2%	61.8%	67.0%

Time steps	Average
5	64.2 %
20	65.2 %
50	66.1 %
100	64.3%

Table 3.16: Accuracy of RNN with LSTM layers (universal, units=20) on test set

Stock / Time steps	GOOG	WMT	JPM
5	60.2%	62.4%	66.7%
20	62.5%	61.6%	65.9%
50	71.1%	62.1%	66.4%
100	73.8%	62.0%	65.7%

Time steps	Average
5	63.1%
20	63.3 %
50	66.5%
100	67.2%

Table 3.17: Accuracy of RNN with LSTM layers (universal, units=35) on test set

Stock / Time steps	GOOG	WMT	JPM
5	59.2%	62.1%	66.8%
20	69.3%	61.8%	66.5%
50	65.7%	62.1%	66.5%
100	69.2%	62.0%	66.4%

Time steps	Average
5	62.7 %
20	65.9 %
50	64.8 %
100	65.9 %

Table 3.18: Accuracy of RNN with LSTM layers (universal, units=50) on test set

As a result, the average test accuracy of the universal model is always lower than that of the stock-specific many-to-many models with the same configurations, which is as expected, because I only used the one-day data of 9 stocks. Moreover, these nine stocks used to train the universal model do not include the three stocks used for the test, so the test process is out-of-sample. I noticed that when 'units'=35, 'time steps'=100, the model reaches its best performance, and it seems that increasing 'time steps' will help us get better performance. After trying 'time steps'=110, 120, 130, I find the corresponding testing accuracies to be 65.2%, 66.2%, 66.3%. Hence I think 67.2% is close to the best performance of the model. Although the results are not as excellent as the results in the paper, given that the data I used here is very limited, the universal model does show us some predictive power in this task. So I would still conclude that the universal price formation mechanism is partially recovered by the simpler universal model based on less data.

3.6 Discussion on the results

- To compare performances of these models, I am going to find out the configurations that provide the models with their best performances, and then compare the highest test accuracies achieved by these models. The best configurations for the FNN without lagged features is 'units' = 50, which gives an average test accuracy of 65.5%. The best configurations for the FNN with lagged features is 'units' = 20 and 'time steps' = 20, which gives an average test accuracy of 69.4%. The best configurations for the many-to-one RNN with LSTM layers is 'units' = 35 and 'time steps' = 100, which gives an average test accuracy of 68.8%. The best configurations for the many-to-many RNN with LSTM layers is 'units' = 50 and 'time

steps' = 100, which gives an average test accuracy of 70.2%. Hence we can conclude that the many-to-many RNN is the best model in this chapter.

- In general, there exists dependence between the labels and the past LOB information. There are two facts to support this. First, for the FNN with lagged features, many-to-many RNN and the many-to-one RNN, input with 20 time steps always outperform input with 5 time steps. Nevertheless, when the length of input sequence extends further to 50 time steps, the FNN with lagged features would often experience a drop in test accuracy, this might be because FNN is not good at dealing with very long input. Besides, the RNN models performance does not increase linearly as 'Time steps' increases, this might be because the longer input would lead to fewer samples for training when the training data remains the same, it may also be caused by higher complexity of the network and potential overfitting problems when more time steps are taken into consideration. The second fact that supports the existence of dependence between the labels and the past LOB information is the performance gap between FNN without lagged features and many-to-many RNN. Moreover, by comparing the performances of many-to-many RNN and FNN with lagged features, we can see many-to-many RNN with LSTM is better at catching the dependence between labels and the past LOB information.

- Another thing to notice is the large-tick effect [24], that is, the direction of next mid-price change for small-tick stocks is harder to predict than large-tick stocks. This is because for large-tick stocks have more useful information is concentrated on the first few price levels, and the models in the thesis are only based on information within 5 price levels, which makes it impossible to catch information outside these price levels. When stocks prices differ a lot, this effect becomes more evident. In May 2020, the share price of JPM is around \$90, the share price of GOOG is over \$ 1000, and the share price of WMT is around \$120. Therefore we should expect to see the test accuracy is highest for JPM and the lowest for GOOG. However, the large-tick effect is not obvious in my tests. This may be because other factors that could affect the prediction quality are more significant in this model.

Chapter 4

Predicting time to the next mid-price change

4.1 Problem description

In this chapter, the main task is to predict when the next change of mid-price would occur. The structures of neural networks are the same as the ones in the last chapter, the only difference is that now this has become a regression problem, with a continuous variable as the label. In this problem, the labels are positive real numbers with no upper bound, but the sigmoid function is bounded. Hence, for the activation function at the last layer in the following neural networks, I would instead use an activation function called rectified linear unit (ReLU), which is defined as $ReLU(x) = \max(0, x)$, where $x \in \mathbb{R}$. I would perform the task with four types of neural networks: FNN without lagged features, FNN with lagged features, many-to-many RNN with LSTM layers, many-to-one RNN with LSTM layers.

Another change in this chapter is the performance metric and loss function, we are going to use the coefficient of determination(R^2) as the performance metric and Mean Absolute Error (MAE) as the loss function. Let us recall their definitions:

Definition 4.1.1.

$$R^2 := 1 - \frac{\sum_i \left(Y_i - \hat{Y}_i\right)^2}{\sum_i \left(Y_i - \bar{Y}\right)^2} = 1 - \frac{MSE}{\text{Sample Variance}}.$$

Definition 4.1.2.

$$\text{MAE} := \frac{\sum_{i=1}^{N} \left|Y_i - \hat{Y}_i\right|}{N},$$

where Y is the actual label and $\hat{Y}$ is the predicted label, and N is the number of labels.

In this sepical task, I would define two more customised R^2 for performance assessment: R^2 for test set, R_t^2, and benchmark R^2, R_b^2:

Definition 4.1.3.

$$R_t^2 := 1 - \frac{\sum_i \left(Y_i - \hat{Y}_i\right)^2}{\sum_i \left(Y_i - \bar{Y}\right)^2} = 1 - \frac{\text{MSE based on estimates}}{\text{Sample Variance on test set}},$$

where Y_i is the actual label in test set, $\hat{Y}_i$ is the corresponding predicted label.

Definition 4.1.4.

$$R_b^2 := 1 - \frac{\sum_i \left(Y_i - (\bar{K}_i)\right)^2}{\sum_i \left(Y_i - \bar{Y}\right)^2} = 1 - \frac{\text{MSE based on mean of training set labels}}{\text{Sample Variance on test set}}.$$

where $\bar{K}_i$ is the mean of the training set labels, Y_i is the actual label in test set, $\hat{Y}_i$ is the corresponding predicted label.

R^2 can be regarded as the proportion of variance that can be explained by the model. Hence larger R^2 is usually better. However, Mean Squared Error can be heavily affected by extreme values, especially when the sample size is not large enough. Therefore, R^2 can be affected by extreme values as it is constructed directly by MSE. Since the values of R_b^2 does not vary as I change models, 'time steps' or 'units'. So there are only three R_b^2 values for all sections except for the last section, the three R_b^2 values are listed below:

GOOG	WMT	JPM	average
-0.023	-0.067	-0.004	-0.031

Table 4.1: R_b^2 for three stocks for section 4.1 to 4.4

However, only looking at the R^2 may not be sufficient, I also want to have a look from another dimension. The Absolute Percentage Error is regarded as a good option.

Definition 4.1.5. The definition of absolute percentage error is

$$APE = \left| \frac{Y - \hat{Y}}{Y} \right|$$

To assist the performance analysis, I plotted the percentiles of Absolute Percentage Error and recorded its key point after every trail. An example of the percentile plot of the absolute percentage error is shown below:

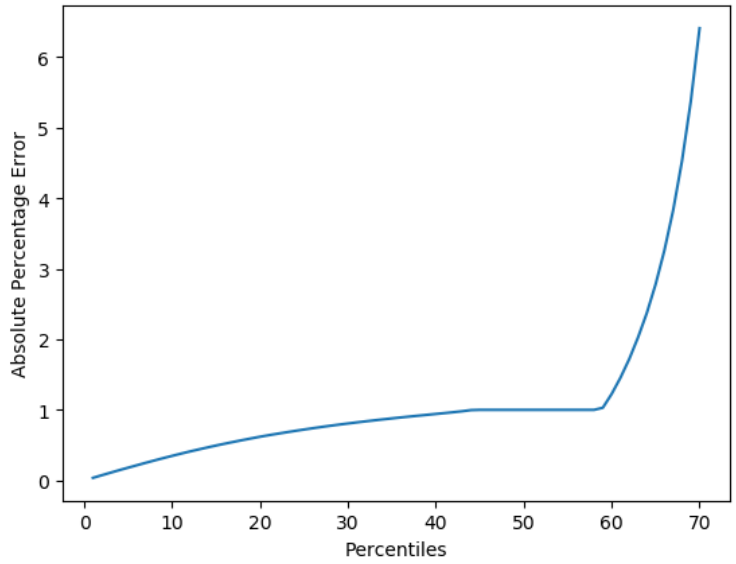

Figure 4.1: An example of the percentile plot of the absolute percentage error

Definition 4.1.6. After defining APE, I would call a prediction with APE less than 1 an **effective prediction**.

The pattern of the percentiles plot can almost always be divided as three parts for any models in this chapter: a relatively flat curve from APE=0 to APE=1, a horizontal line staying at APE=1, a much steeper curve for APE larger than one. In these three parts, the connection between the first two parts is the key point in the Percentiles plot, which is when the APE first time becomes an exact '1'. By examining the empirical results, I found a '1' in APE almost always indicates the prediction made is just 0, which can be treated as ineffective or passive predictions in practical. For those predictions that make APE even higher than 1, I would treat them as ineffective predictions, and I think they are not equivalently important to those predictions with APE less than 1

because a prediction with APE larger than 1 is more than 90% of the time also has an APE lager than 2, which means they could often lead to more damage in real-world trading activity. Thus I recorded the proportion of predictions with APE less than 1 as a performance measure at another dimension, and it will be named as the **proportion of effective predictions**. It gives us a more thorough understanding of the performance.

However, I would still treat R_t^2 as the primary measure of performance because R_t^2 is more powerful and focuses on the actual quality of each prediction, and since the sample size is quite large, the effect of extreme values on R_t^2 is largely reduced. But the proportion of effective predictions might be pointless in some cases, for example, if a model only predicts something very small, for example, 10^{-5} of the minimum of the labels in the training dataset, then its APE would almost always be less than 1, but this model is not meaningful.

4.2 Feedforward neural network

In this section, I attempt to predict when the next mid-price change would occur with two types of Feed-forward neural networks(FNNs), one of which uses lagged features, the other does not. By this experiment, we can also see how good the FNN is at capturing the dependence between the labels and the past LOB information. Both FNNs share the same structure, which can be described by Table 4.2.

	units	Type of layer
layer 1	x	Dense with ReLU activation
layer 2	x	Dense with ReLU activation
layer 3	x	Dense with ReLU activation
layer 4	1	Dense with ReLU activation

Table 4.2: Structure of FNN for time to next mid-price change prediction

Please note that 'units' represents the number of units per layer for the first three layers.

4.2.1 FNN without lagged features

Similarly, I am going to build a simple FNN which assumes the occurrence of mid-price change is markovian. So I have to abandon any lagged features and only use the most recently available features, hence the new features used to train the model are the following:

- Time,
- Bid price 1, bid size 1, ask size 1, ask price 1
- Ask price 2, bid price 2,
- Ask price 3, bid price 3,
- Ask price 5, bid price 5,
- Time from the last mid-price change.
- Time between the last two mid-price changes.

With the naive assumption, I have done a formal experiment, which includes training the model with 3 different configuration: 'unit' = 20, 'unit' = 35, 'unit' = 50. The test results are described in the following types of tables: R_t^2 table and proportion of effective predictions table, see Tables 4.3, 4.4 for the results.

Stock / units	GOOG	WMT	JPM
20	0.096	0.086	0.005
35	0.081	-0.125	-0.100
50	0.159	-0.125	-0.035

x	Average
20	0.062
35	-0.048
50	0.000

Table 4.3: R_t^2 of FNN without lagged features

Stock / units	GOOG	WMT	JPM
20	13%	2 %	0%
35	12%	0 %	0%
50	13%	0%	1%

x	Average
20	5.0%
35	4.0%
50	4.7%

Table 4.4: Proportion of effective predictions of FNN without lagged features

where the effective prediction and units have been defined previously.

As we can see from the results above, more than half of the R_t^2s above are positive, but most of them are not far away from the R_t^2. The reader may also notice that the '-0.125' appears twice, which is because the training processes stuck in very similar places. The proportion of effective predictions is not very high and often 0 in the table, but due to the high ratio of positive R_t^2 values, I would not say the FNN without using lagged features is completely useless in predicting when the next mid-price change would occur. Certainly, we need to see some more results from other models.

4.2.2 FNN with lagged features

Similar to the last chapter, I build an FNN with lagged features as a benchmark in the task, the FNN should be able to catch some of the recurrent information in the directions of mid-price changes, the features used in the FNN are the following:

- Time,
- Bid price 1, bid size 1, ask size 1, ask price 1
- Ask price 2, bid price 2,
- Ask price 3, bid price 3,
- Ask price 5, bid price 5,
- The last N 'time from the last mid-price change'.
- The last N 'time diffrences between the two consecutive mid-price changes'.

Since a sample in FNN training process is one-dimensional, lagged features are added to these samples to catch some recurrent information in the past limit order book data. The lagged features added here are "The last N 'time from the last mid-price change' ". and "The last N 'time differences between the two consecutive mid-price changes'", where N is treated as the 'time steps'. Let us check the performance of the modified FNN model in Tables 4.5, 4.6, 4.7, 4.8, 4.9, 4.10.

Stock / Time steps	GOOG	WMT	JPM
20	0.127	-0.039	-0.012
50	0.111	-0.054	-0.156
100	0.092	0.025	0.045

Time steps	Average
20	0.025
50	-0.033
100	0.054

Table 4.5: R_t^2 of FNN with lagged features (units=20)

Time steps \ Stock	GOOG	WMT	JPM
20	34%	3%	22%
50	17%	4%	0%
100	32%	7%	43%

Time steps	Average
20	20.0%
50	7.0%
100	27.3%

Table 4.6: Proportion of effective predictions of FNN with lagged features (units=20)

Time steps \ Stock	GOOG	WMT	JPM
20	0.124	-0.004	-0.071
50	0.070	0.003	0.044
100	0.081	-0.046	-0.154

Time steps	Average
20	0.016
50	0.039
100	-0.040

Table 4.7: R_t^2 of FNN with lagged features (units=35)

Time steps \ Stock	GOOG	WMT	JPM
20	34%	26%	5%
50	13%	3%	20%
100	30%	7%	0%

Time steps	Average
20	21.7%
50	12.0%
100	12.3%

Table 4.8: Proportion of effective predictions of FNN with lagged features (units=35)

Time steps \ Stock	GOOG	WMT	JPM
20	0.105	0.028	-0.155
50	0.081	-0.151	-0.156
100	-0.042	-0.154	-0.042

Time steps	Average
20	-0.007
50	-0.075
100	-0.079

Table 4.9: R^2 of FNN with lagged features (units=50)

Time steps \ Stock	GOOG	WMT	JPM
20	26%	10%	0%
50	19%	0%	0%
100	1%	0%	6%

Time steps	Average
20	12.0%
50	6.3%
100	2.3%

Table 4.10: Proportion of effective predictions of FNN with lagged features(units=50)

The performance of the modified FNN model is still not satisfying, because more than half of the average R_t^2 values are negative, and for those positive R_t^2 values, most of them are still at a low level and not far away from their corresponding R_b^2 values. The highest R_t^2 achieved is on GOOG when 'units'=20 and 'time steps'=20, with a R_t^2 of 0.127, the corresponding proportion of effective predictions is 34%. The 0.127 is not better than the best record of FNN without lagged features. This model also loses when comparing the best average R_t^2 with the FNN without lagged features. However, in terms of the proportion of effective predictions, the FNN with lagged features is much better. However, we need to see the performances of all other models to make the final comment.

4.3 RNN with LSTM layers (many to one)

In Chapter 3, I used many-to-one RNN to tackle a classification problem, which was outperformed by the FNN with lagged features. In the current chapter, the features used are still formed by limit order book time series, and I would expect similar performance relative to FNN with lagged

features here. The connection structure of the many-to-one RNN can be found in Figure 3.2, Table 4.11 is made to offer some further details of the layers in the many-to-one RNN.

	Units	Type of layer
layer 1	x	LSTM
layer 2	x	LSTM
layer 3	x	LSTM
layer 4	1	Dense with relu activation

Table 4.11: Structure of RNN with LSTM layers(many to one)

The features used in the many-to-one model are selected in Chapter 1. As the input sample of LSTM layers are two dimensional, where the first dimension is 'time steps', the second is 'features'. I only need to place the original features at each of these time steps without adding any lagged features, and the RNN will make predictions based on information from all previous time steps in the same sample. The features at each time step are:

- Time,
- Bid price 1, bid size 1, ask size 1, ask price 1,
- Ask price 2, bid price 2,
- Ask price 3, bid price 3,
- Ask price 5, bid price 5,
- Time from the last mid-price change,
- Time diffrences between the last two mid-price changes.

The performances at different configuration are described by two dimensions: R_t^2 and the proportion of effective predictions. See the tables below for the experiment results.

Time steps \ Stock	GOOG	WMT	JPM
20	0.036	-0.082	0.028
50	0.007	-0.024	-0.010
100	-0.003	-0.129	0.002

Time steps	Average
20	-0.006
50	-0.009
100	-0.043

Table 4.12: R_t^2 of many-to-one RNN (units=20)

Time steps \ Stock	GOOG	WMT	JPM
20	23%	5%	26%
50	23%	13%	31%
100	18%	0%	32%

Time steps	Average
20	18.0%
50	22.3%
100	16.7%

Table 4.13: Proportion of effective predictions of many-to-one RNN (units=20)

Time steps \ Stock	GOOG	WMT	JPM
20	0.020	-0.031	-0.045
50	-0.008	-0.041	0.007
100	-0.004	-0.109	-0.023

Time steps	Average
20	-0.019
50	-0.014
100	-0.045

Table 4.14: R_t^2 of many-to-one RNN (x=35)

Time steps \ Stock	GOOG	WMT	JPM
20	27%	10%	13%
50	19%	8%	27%
100	19%	0%	19%

Time steps	Average
20	16.7%
50	18.0%
100	12.7%

Table 4.15: Proportion of effective predictions of many-to-one RNN (units=35)

Time steps \ Stock	GOOG	WMT	JPM
20	0.027	-0.018	-0.130
50	0.006	-0.021	-0.058
100	0.004	-0.040	-0.099

Time steps	Average
20	-0.040
50	-0.024
100	-0.045

Table 4.16: R^2 of many-to-one RNN (x=50)

Time steps \ Stock	GOOG	WMT	JPM
20	24%	10%	1%
50	19%	9%	0%
100	23%	12%	2%

Time steps	Average
20	11.7%
50	9.3%
100	12.3 %

Table 4.17: Proportion of effective predictions of many-to-one RNN (units=50)

The results above are not surprising to me, because when I test the many-to-one model in Chapter 3, the FNN with lagged features already beats the many-to-one model once. We can see from the results that all the average R_t^2 values are less than 0, some of them are slightly higher than the average R_b^2, but still not very meaningful. The largest R_t^2 is achieved when units=20 and 'time steps'=20 for GOOG, at which the R_t^2 reaches 0.036, which is still a very small number. This is worse than the best record in the FNN with lagged features test. Even when comparing the best average proportion of effective predictions, the best average proportion of effective predictions for many-to-one RNN is 22.3%, lower than the 27.3% from FNN with lagged features. Therefore, I think the many-to-one model fails to compete with FNN with lagged features in this task.

4.4 RNN with LSTM layers (many to many)

Finally, the many-to-many model is coming to the stage, the connection structure of many-to-many RNN with LSTM layers can be referred to 3.3, here I would use a table to describe the details in each layer:

	units	type of layer
layer 1	x	LSTM
layer 2	x	LSTM
layer 3	x	LSTM
layer 4	1	Dense with ReLU activation

Table 4.18: Structure of RNN with LSTM layers(many to many), Chapter 4

Although many-to-many RNN has longer output, the input structure is still the same as many-to-one RNN. Hence, the features used in the many-to-many model is the same as those in section 4.3 and will not be repeatedly listed here. The performance is tested at different configurations ('units per LSTM layer', 'time steps') and is again measured by the same measures (R_t^2 and proportion of effective predictions). See Tables 4.19, 4.20, 4.21, 4.22, 4.23, 4.24 for the experiment results.

Time steps \ Stock	GOOG	WMT	JPM
20	0.122	-0.020	0.000
50	0.139	0.061	0.023
100	0.172	0.086	0.021

Time steps	Average
20	0.034
50	0.074
100	0.093

Table 4.19: R_t^2 of many-to-many RNN (units=20)

Time steps \ Stock	GOOG	WMT	JPM
20	41%	1%	18%
50	27%	13%	34%
100	34%	40%	4%

Time steps	Average
20	20.0%
50	24.7%
100	26.0%

Table 4.20: Proportion of effective predictions of many-to-many RNN (units=20)

Time steps \ Stock	GOOG	WMT	JPM
20	0.127	-0.099	0.067
50	0.146	0.070	0.024
100	0.105	0.087	-0.125

Time steps	Average
20	0.032
50	0.080
100	0.022

Table 4.21: R_t^2 of many-to-many RNN (units=35)

Time steps \ Stock	GOOG	WMT	JPM
20	25%	0%	41%
50	42%	23%	43%
100	23%	36%	0%

Time steps	Average
20	22.0%
50	36.0%
100	19.7%

Table 4.22: Proportion of effective predictions of many-to-many RNN (units=35)

Time steps \ Stock	GOOG	WMT	JPM
20	0.160	0.002	0.019
50	0.132	0.070	-0.071
100	0.108	0.092	0.041

Time steps	Average
20	0.060
50	0.044
100	0.080

Table 4.23: R_t^2 of many-to-many RNN (units=50)

Time steps \ Stock	GOOG	WMT	JPM
20	28%	5%	42%
50	30%	37%	0%
100	33%	29%	42%

Time steps	Average
20	25.0%
50	22.3%
100	34.7%

Table 4.24: Proportion of effective predictions of many-to-many RNN (units=50)

Finally we can see something acceptable, first of all, all the average R_t^2 values are positive now, and the highest stock-specific R_t^2 is recorded for 'GOOG' when 'units'=20 and 'time steps'=100, which reaches a R_t^2 of 0.172, which is the highest among all the models, and the corresponding proportion of effective predictions is 34%, which is also nice. The performance on average also beats all the other models at all these configurations tried within this chapter, on average the best configuration would be 'units'=20, 'time steps' = 100 if we only consider R_t^2. The best

average proportion of effective predictions of this model is the highest(36.0%) at 'units'=35, 'Time steps'=50, which is also the best over all the models. The average performance boost at 'units'=20 and 'time steps'=100 for 'GOOG' might be due to increasing in the complexity of the model, and I further guess that extending time steps could probably improve the performance further, so I have done some more tests and see if the performance will continue increasing in the following experiment, which gives us the results below:

time steps / stock	108	117	125	150
GOOG	0.141	0.148	0.167	0.124

Table 4.25: R_t^2 of many-to-many RNN (units=20),extra tests

time steps / stock	108	117	125	150
GOOG	40%	36%	29%	29%

Table 4.26: Proportion of effective predictions of many-to-many RNN (units=20),extra tests

The R_t^2 does not increase as 'time steps' value increases further to 108, 118, 125 and 150. Hence I would say 'units'=20 and 'time steps'=100 is quite close to the best configuration for this model.

To examine the actual error distribution, I selected a case for GOOG with 50 time steps and 20 units, which represents the distribution of error from a relatively good model, I plotted the distribution of error as a truncated histogram in Figure 4.2.

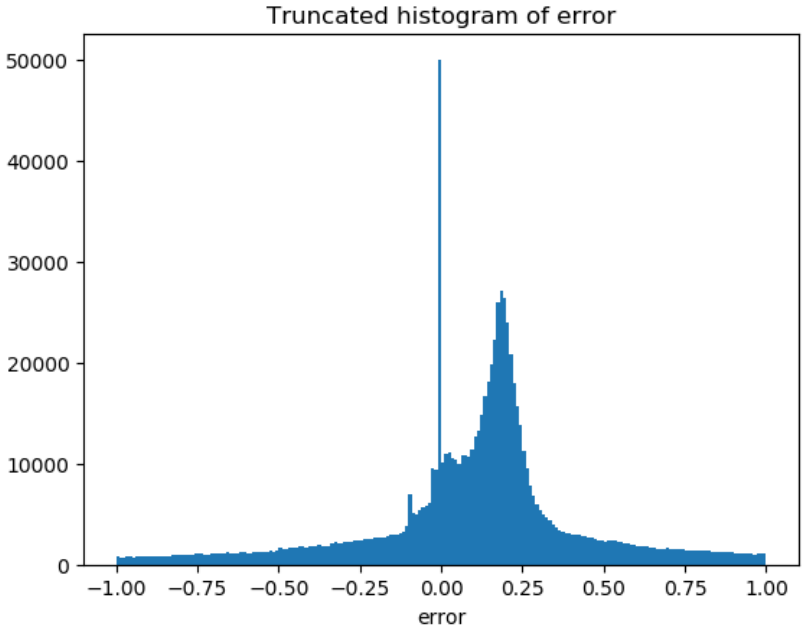

Figure 4.2: Distribution of the error

Note that, in this case, the median of labels is 0.07s, and the mean of labels is 0.7s. Furthermore, the error of a prediction is $\hat{Y}_i - Y_i$. From the mean and median number, we can see how large the extreme values are, that is also why I use the truncated distribution. Besides, Figure 4.2 above shows us many predictions are simply 0, which is passive, though not necessarily bad. We can also see the model tends to overestimate the labels, as most of the weights are on the positive side. However, the good thing is that most of the errors are on the scale of the mean of the labels. So the model knows the suitable scale of predictions.

4.5 Universality test

Similar to the last section in the universality test, I would like to discuss if it is possible to build a 'universal' model with little multi-stock high-frequency data that has similar predictive power as the corresponding stock-specific models. To start the training process, I used the same stocks' data as in section 3.5 again, the dates of the data are the same as well, so the test dataset and the training dataset are just copies from section 3.5, except that the labels and some features in the datasets are adjusted to suit the current task. The features used are the same as the features used in the last section. The results are recorded in Table 4.27, 4.28, 4.29, 4.30, 4.31, 4.32.

Time steps \ Stock	GOOG	WMT	JPM
20	-0.141	-2.151	-10.073
50	-0.162	-2.347	-26.688
100	-0.161	-5.205	-33.875

Time steps	Average
20	-4.122
50	-9.732
100	-13.080

Table 4.27: R_t^2 of the universal RNN (units=20)

Time steps \ Stock	GOOG	WMT	JPM
20	0%	2%	2%
50	0%	2%	1%
100	0%	1%	1%

Time steps	Average
20	1.3%
50	1.0%
100	0.7%

Table 4.28: Proportion of effective predictions of the universal RNN (units=20)

Time steps \ Stock	GOOG	WMT	JPM
20	-0.122	-0.283	-13.338
50	-0.161	-3.288	-36.031
100	-0.168	-0.124	-64.270

Time steps	Average
20	-4.581
50	-13.160
100	-21.521

Table 4.29: R_t^2 of the universal RNN (units=35)

Time steps \ Stock	GOOG	WMT	JPM
20	0%	1%	0%
50	0%	1%	2%
100	1%	0%	1%

Time steps	Average
20	0.3%
50	1.0%
100	0.7%

Table 4.30: Proportion of effective predictions of the universal RNN (units=35)

Time steps \ Stock	GOOG	WMT	JPM
20	-0.156	-0.121	-0.186
50	-0.161	-0.125	-12.596
100	-0.171	-0.124	-6.028

Time steps	Average
20	-0.154
50	-4.294
100	-2.108

Table 4.31: R_t^2 of the universal RNN (units=50)

Time steps \ Stock	GOOG	WMT	JPM
20	0%	0%	15%
50	7%	0%	13%
100	0%	0%	0%

Time steps	Average
20	5.0%
50	6.7%
100	0.0%

Table 4.32: Proportion of effective predictions of the universal RNN (units=50)

As all the R_t^2 values are negative in the results, and some of then are way below 0, so I can conclude that, unlike the model in Chapter 3, the universal model completely failed to predict the occurrence of the next mid-price change. There might not exist a universal model that can outperform the corresponding stock-specific models in predicting when the next mid-price change would occur. However, due to the simplicity of my naive universal model and the limited amount of data used for training, it might be because we need to build a more complex model with much more data and training, which could be a potential point of further research.

4.6 Discussion on the results

- While using R_t^2 as the performance measure, the best average R_t^2 achieved by FNN without lagged features, FNN with lagged features, many-to-one RNN with LSTM layers, many-to-many RNN with LSTM layers are 0.062, 0.054, -0.006, 0.093, respectively. While using the average proportion of effective predictions(POEP) as the performance measure, the best average POEP achieved by FNN without lagged features, FNN with lagged features, many-to-one RNN with LSTM layers, many-to-many RNN with LSTM layers are 5.0%, 27.3%, 22.3%, 36.0%, respectively. The many-to-many model is the winner in both criteria, which demonstrates the ability of the many-to-many model in learning regression problems and catching dependence between the labels and past LOB information.

- This is a relatively hard task if we only look at the R_t^2 values, many of them are even less than their corresponding R_b^2 values (apart from the many-to-many RNN case), none of them is above 0.5, no matter which model is applied. That shows us the difficulty of this task, but it is somehow reasonable, since the direction of next mid-price prediction is already quite accurate, especially for 'GOOG', thus if it is so easy to predict when it would have, then the likelihood of arbitrage would be considerably high. That might be one of the reasons why this task is so hard.

- The large-tick effect again does not exist, as the large-tick stock 'JPM' is not easier to predict than 'GOOG', which might be because there are some other more important factors affecting the prediction quality, or the large-tick effect has to be applied on stocks with smaller stock prices to be effective.

Conclusion

In the thesis, two tasks are performed by four different types of neural networks. The first task is to predict the next direction of stock mid-price change. Out of the four models, the many-to-many RNN with LSTM layers outperformed all the other candidate models and achieved the highest average test accuracy of 70.2%, which is relatively successful. The second task is to predict when the mid-price change would occur. Although the performance of all these models are not very satisfying if we use R_t^2 as the performance measure, we can still see the leading position of many-to-many RNN with LSTM layers compared with other candidate models in the task. Moreover, while assessing the model performance by using the proportion of effective predictions, the performance of many-to-many RNN with LSTM layers is indeed acceptable. Besides, both tasks demonstrate the ability of many-to-many RNN with LSTM layers in catching the long-term memory of the input sequence.

The universal models built in Chapter 3 and Chapter 4 exhibits quite distinctive performances, the universal model in Chapter 3 shows us the existence of universality in the price formation mechanism in the stock market. However, the universal model in Chapter 4 failed to demonstrate any predictive power, no matter which measure we use to measure its performance. The potential reasons are discussed in the chapter.

Throughout the paper, it is not hard to see the stock 'GOOG' seems to be more predictable than the other two stocks, and I think there could be multiple reasons: 1. the trading activities on 'GOOG' is more frequent, which results in more data accumulated for the same period for 'GOOG', and hence training of 'GOOG' has more data support. 2. 'GOOG' is the stock of a large technology company, which means it is famous and popular in the current market, there might be more retail investors trading the shares of 'GOOG', and the behaviours of retail investors are relatively easier to predict. 3. Other participants may also play an important role, due to the popularity of 'GOOG', there might be more large high-frequency trading companies participating the trading of 'GOOG' shares, and these companies may use similar algorithms while trading, which makes the mid-price movement more predictable.

For further researches, there are two directions that I am interested in. The first is the profitability of these models, the models built here are mainly used to predict how and when the mid-price of stock would change, but in reality, even if the prediction is correct, it is often not profitable due to existence of bid-ask spread. Maybe we can do more researches on how to combine different models and create a system that can consistently make profitable trading decisions. The second direction is on the structure of neural networks. Due to the complexity and the flexibility of RNN, we may investigate on more advanced neural networks structure in the future, together with more powerful computation tools, I believe we will reach the new limit shortly.

Bibliography

[1] W.a. Little. The existence of persistent states in the brain. *Mathematical Biosciences*, 19(1-2):101–120, 1974.

[2] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79(8):2554–2558, 1982.

[3] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[4] Jürgen Schmidhuber. Habilitation thesis: System modeling and optimization, 1993. Page 150, link: ftp://ftp.idsia.ch/pub/juergen/habilitation.pdf.

[5] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks, Nov 2012.

[6] Sepp Hochreiter. Untersuchungen zu dynamischen neuronalen netzen, diploma thesis, tu munich, 1991.

[7] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[8] Alex Graves, Douglas Eck, Nicole Beringer, and Juergen Schmidhuber. Biologically plausible speech recognition with lstm neural nets. *Biologically Inspired Approaches to Advanced Information Technology Lecture Notes in Computer Science*, page 127–136, 2004.

[9] Santiago Fernández, Alex Graves, and Jürgen Schmidhuber. An application of recurrent neural networks to discriminative keyword spotting. *Proceedings of the 17th International Conference on Artificial Neural Networks – ICANN 2007*, pages 220–229, 2007.

[10] Schmidhuber Jürgen Alex Graves. Offline handwriting recognition with multidimensional recurrent neural networks, 2009. Neural Information Processing Systems (NIPS) Foundation, page 545–552.

[11] Haşim Sak, Andrew Senior, Kanishka Rao, Françoise Beaufays, and Johan Schalkwyk. Google voice search: faster and more accurate, Sep 2015.

[12] Jiayu Qiu, Bin Wang, and Changjun Zhou. Forecasting stock prices with long-short term memory neural network based on attention mechanism. *Plos One*, 15(1), 2020.

[13] Justin Sirignano and Rama Cont. Universal features of price formation in financial markets: Perspectives from deep learning. *SSRN Electronic Journal*, 2018.

[14] output. `https://lobsterdata.com/info/DataStructure.php`. Accessed: 2020-08-01.

[15] Sebastian Raschka. Mlxtend: Providing machine learning and data science utilities and extensions to python's scientific computing stack. *The Journal of Open Source Software*, 3(24), April 2018.

[16] Robert F. Engle and Jeffrey R. Russell. Autoregressive conditional duration: A new model for irregularly spaced transaction data. *Econometrica*, 66(5):1127–1162, 1998.

[17] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*, pages 367–415. MIT Press, 2016. http://www.deeplearningbook.org.

[18] Gang Chen. A gentle tutorial of recurrent neural network with error backpropagation. *CoRR*, abs/1610.02583, 2016.

[19] Y. Bengio, P. Frasconi, and P. Simard. The problem of learning long-term dependencies in recurrent networks. In *IEEE International Conference on Neural Networks*, pages 1183–1188 vol.3, 1993.

[20] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*, pages 396–399. MIT Press, 2016. http://www.deeplearningbook.org.

[21] Felix A. Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. *Neural Computation*, 12(10):2451–2471, 2000.

[22] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 12 2014.

[23] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*, pages 164–223. MIT Press, 2016. http://www.deeplearningbook.org.

[24] Martin Gould and Julius Bonart. Queue imbalance as a one-tick-ahead price predictor in a limit order book. *Market Microstructure and Liquidity*, 12 2015.