University of Birmingham

Stock market prediction of the CSI 300 using artificial neural networks

Author

Andy Chen Xia

Supervisor

Dr. Syed Shabi-Ul-Hassan

MSc Investments (2017/2018)

Student ID: 1458480

Dissertation Submitted in Partial Fulfilment of the degree of

MSc in Investments (2017/2018)

The word count: 9563

Abstract

Forecasting stock price is an important financial problem that has always caught the attention of researchers. In the last few years, machine learning models have gained their fame on solving complicated non-linear problems, leading to the development of several neural network models and hybrid models to predict the price movement of stocks. This paper evaluates the predictive power of three different machine learning models, the Multilayer Perceptron (MLP) neural network, the Convolutional Neural Network (CNN) and the Long Short-Term Memory (LSTM) neural network to forecast the Chinese Security Index (CSI) 300 price change. Moreover, the predictive power of them are evaluated using the Mean Square Error (MSE), the Mean Absolute Deviation (MAD), the Mean Absolute Percentage Error (MAPE) and the R^2. In the study, the results show that the three models have promising predictive power, but the MLP neural network has the highest forecasting accuracy among them.

Keywords: Artificial neural networks (ANN), Financial time series prediction, the Chinese Security Index (CSI) 300, Multilayer perceptron (MLP) neural network, Convolutional neural network (CNN), Long short-term memory (LSTM) neural network.

Contents

1 Introduction **4**

2 Literature Review **5**

2.1 What are Artificial Neural Networks (ANNs) ? 5

2.2 Background 5

2.2.1 Studies about the Multilayer Perceptron Neural Network (MLPNN) . . 6

2.2.2 Optimization of ANNs' forecasting power using fuzzy sets 7

2.2.3 Optimization of ANNs' forecasting power using swarm intelligence . . 8

2.2.4 Optimization of ANNs' forecasting power through combinations 9

2.2.5 Optimization of ANNs' forecasting power using evolutionary algorithms 10

2.2.6 Optimization of ANNs' forecasting power using Wavelet 11

2.2.7 Studies about the Long Short-Term Memory Neural Network (LSTMNN) 12

2.2.8 Studies about the Convolutional Neural Network (CNN) 12

3 Methodology **13**

3.1 Multilayer Perceptron Neural Network (MLPNN) 13

3.1.1 Information Processing 13

3.1.2 Learning Process 15

3.2 Long Short-Term Memory Neural Network (LSTMNN) 15

3.2.1 Information Processing 16

3.2.2 Learning Process 17

3.3 Convolutional Neural Network (CNN) 18

3.3.1 Information Processing 18

3.3.1.1 Convolutional layer 18

3.3.1.2 Pooling Layer 19

3.3.1.3 Fully connected neural network 19

3.3.2 Learning Process 20

4 Data, Analysis, and Results **20**

4.1 Data description 20

4.1.1 Data Normalization 21

4.1.2 Converting time series to an image 22

4.2 Analysis 23

4.2.1 Weight initialization 23

4.2.2 Hyperparameter setting for Adam optimization 24
4.2.3 MLP training 25
4.2.4 LSTM training 26
4.2.5 CNN training 26
4.3 Model performance measurents 28
4.4 Results 28

5 Conclusion & Further Work 31

A Python code for the MLPNN 39

B Python code for the LSTMNN 50

C Python code for the CNN 61

1 Introduction

Financial price forecasting is regarded as one of the hottest fields of interest for theoretical as well as practical purposes. On the one hand, financial market participants spend a lot of money and effort to collect and analyze the data using different models to increase the accuracy of the prediction to maximize their profit before making a financial decision. On the other hand, members from the academia investigate different forecasting models to see if the market can be predicted to verify different theories or to obtain a better understanding about the essence of the market.

According to Abu-Mostafa & Atiya (1996), the financial market is dynamic, non-linear, complex, chaotic and non-parametric in nature. Furthermore, time series data are noisy, multi-stationary and non-linear making many linear models to fail to capture certain patterns (Oh & Kim 2002). Additionally, macroeconomic factors such as interest rates, government policies, commodity price index, investors' expectations and the psychology of investors can impact stock prices (Wang 2002). Members of the academy and industry have performed many struggles at developing and adjusting models to obtain more accurate forecast. However, obtaining a model for 'better' prediction is still a subject matter that remains unsolved.

In this paper, we examine three different artificial neural networks (ANN), the Multilayer Perceptron (MLP) neural network, the Long Short-Term Memory (LSTM) neural network, and the Convolutional neural network (CNN). These models have shown the ability to solve problems with significant complexity in other fields (Sundermeyer et al. (2012); Lawrence et al. (1997); Orhan et al. (2011)). Hence, we exploit their ability of learning complex pattern for solving nonlinear problems and apply it to the prediction of financial time series. To my best knowledge, there is no article focused on comparing the prediction accuracy of the three models mentioned above. Therefore, in this study, we compare the performance of the three neural networks in predicting the price change of the Chinese Security Index (CSI) 300.

The performance of the analyzed models will be compared by using the Mean Square error (MSE), the Mean Absolute Percentage Error (MAPE), the Mean Absolute Deviation (MAD), and the R^2. The numerical result demonstrates that the MLP neural network is the most suitable to be used as a forecasting tool, since it achieves the highest forecasting accuracy among the three models.

This paper is organized into the following sections. Section 2 is devoted to present the current researches on implementing ANNs for financial time series forecasting. Section 3 presents the three neural network models. Section 4 describes the data, shows the results and analyzes them. Section 5 talks about the conclusion of the study and the possible future research. Additionally, the python code used to create the three ANN models are provided in the appendices.

This study will not only contribute to the research of ANNs but also to the financial implementation of market value calculation.

2 Literature Review

2.1 What are Artificial Neural Networks (ANNs) ?

ANNs are computing architectures inspired by the biological neural network to mimic the way animal brains work (Van Gerven & Bohte 2018). An ANN is composed of units called artificial neurons which are connected through ”links” to mimic the synapses process (Van Gerven & Bohte 2018). They can learn how to accomplish tasks by examining examples without being programmed beforehand with task-specific rules. For example, in image recognition, they can learn to recognize what a car is by examining the pictures labeled as a car and then use the learned ”knowledge” to recognize cars in other pictures (Van Gerven & Bohte 2018). Besides, it needs to be noted that the ANN does not have any prior knowledge. Therefore, due to its learning capability, it has been broadly implemented to perform a variety of tasks, such as computer vision, speech recognition, playing board games, medical diagnosis and owing to its ability to learn nonlinear patterns, it has also been a hot field of research in financial forecasting.

2.2 Background

Different financial theories have established the ground for a variety of forecasting models for time series data. However, these models cannot precisely predict the market price since external factors, which the models might fail to capture, profoundly influence it. De Gooijer & Hyndman (2006) examined over 940 papers on time series forecasting from 1982 to 2005 for the 25th anniversary of the International Institute of Forecasters (IIF). In their article, they revised different methods and models including exponential smoothing, seasonality, ARIMA, state space, structural models and the Kalman filter, nonlinear models, extended memory models, and ARCH-GARCH. Additionally, they discussed the benefits and drawbacks of each methodology and talked about the possible future improvements. Furthermore, they pointed out the strong forecasting ability of ANNs, but also the issues and shortcoming they have compared to some ”traditional” techniques. Moreover, researchers have not stopped exploring ways to improve the forecasting ability of ANNs.

2.2.1 Studies about the Multilayer Perceptron Neural Network (MLPNN)

Cao et al. (2005) used the CAPM, Fama and French 3-factor model, and a MLPNN to forecast the price movement of stocks traded on the Shanghai stock exchange. The numerical result shows that the ANN significantly outperforms the linear models. O’Connor & Madden (2006) implemented a MLPNN using external factors like commodity price and currency exchange rate to predict the Dow Jones Industrial Average Index (DJIA). The performance is then examined in a simulation making trades based on the ANN prediction. The ANN generated 23.5% annual return compared to 13.03% annual growth of DJIA. Dutta et al. (2006) used two MLPNNs with three hidden layers to predict the Bombay Stock Exchange (BSE) SENSEX weekly closing value. In the study, each ANN received different input, and the result shows that the ANN using a 52-week moving average of the weekly closing price, a 5-week moving average of the same, and a 10-week Oscillator as inputs performed better. Panda & Narasimhan (2006) implemented a MLPNN to predict BSE-SENSEX daily return. Moreover, to assess the performance of the model, it was compared to the Random Walk (RW) and the linear autoregressive model. The empirical result indicates a superiority of the MLPNN over the other two models.

Chen et al. (2006) compared the performance of the Support Vector Machine (SVM) and the MLPNN in predicting the price movement of six major Asian stock markets (Nikkei 225, All Ordinaries, Hang Seng, Straits Times, Taiwan Weighted and KOSPI). The results show that both models have strong forecasting ability, and the SVM is not always better than the MLPNN as mentioned in other works of literature. The MLPNN outperforms the SVM in the prediction of the Nikkei 225 and the Straits Times price. Constantinou et al. (2006) compared the performance of the Markov switching model and the MLPNN on predicting the Cyprus stock exchange. The result indicates that the Markov switching model does not generate more accurate forecast than the MLPNN. Hamzaçebi et al. (2009) examine the predictive power of ARIMA and the MLPNN using Grey Relational Analysis (GRA) as a measurement for performance and reach to the conclusion that the ANN was better than ARIMA. However, they point out that more research needs to be carried in order to generalize the conclusion. Vaisla & Bhatt (2010) compared the MLPNN with the Multiple Regression Analysis using data from the National Stock Exchange of India (NIFTY). The results show that the ANN is better than the statistical method, but it was also observed that the ANN forecasting ability decreases as the time series get more complicated. Hence, they concluded that is better to use the ANN for daily forecast. Kara et al. (2011) compared the performance of the MLPNN and the SVM in predicting the daily stock movement of the Istanbul Stock Exchange (ISE) National 100 Index. The result shows that the forecasting ability of the ANN is significantly better than the SVM.

Naeini et al. (2010) studied the performance of the MLPNN, Elman recurrent network and linear models on stocks traded on the Tehran Stock Market. The numerical results show that the ANN is better at predicting changes on stock value, but the other two can predict the direction of the change better than the MLPNN. Guresen et al. (2011) analyzed the performance between DAN2, MLPNN, NN-GARCH, and DAN2-GARCH using a real data set from the NASDAQ stock exchange. The results show that the traditional MLPNN outperformed the other models. Patel et al. (2015) compared the performance of four different machines learning techniques (MLPNN, SVM, random forest and naive-Bayes) on data from 2003 to 2012 on two stocks (Reliance Industries and Infosys Ltd.) and two stock price indices (CNX Nifty and BSE-SENSEX). The numerical result shows that the random forest is generates the most accurate prediction among the four implemented models. Song et al. (2018) applied five different ANN models (MLPNN, RBFNN, GRNN, SVMR, and LS-SVMR) to predict the price change of three Chinese stocks (Bank of China, Vanke A and Kweichou Moutai). The empirical result shows that the MLPNN's financial forecast accuracy consistently beats the other four models.

2.2.2 Optimization of ANNs' forecasting power using fuzzy sets

Huarng & Yu (2006) implemented a neural network to forecast fuzzy time series on the Taiwan Stock Exchange Capitalization Weighted Stock Index (TAIEX). The result shows that the new model outperforms the basic fuzzy time series and Chen's model. Zhang & Wan (2007) developed a new ANN architecture, the Statistical Fuzzy Interval Neural Networks to predict the exchange rate between the US dollar and other three currencies (Japanese Yen, British Pound and Hong Kong Dollar). The model returns an interval which is constituted by three values the upper bound, lower bound and mean. The interval varies according to the confidence level, the higher the degree of confidence the more extensive the interval.

Yu & Huarng (2008) proposed two bivariate model, the bivariate neural network-based fuzzy time series model, and a bivariate neural network-based fuzzy time series model with substitutes to test if bivariate models have better performance than univariate models. These were employed to forecast the price movement on the TAIEX and the Taiwan Futures Exchange (TAIFEX). Moreover, the models were compared to other univariate models such as Chen's fuzzy time series model, univariate neural network-based fuzzy time series model, and univariate neural network-based fuzzy time series model with substitutes. In the analysis, the bivariate neural network-based fuzzy time series model with substitutes outperform the other models. Wei (2016) proposed an Adaptive Network-based Fuzzy Inference System model using empirical mode decomposition to predict stock prices in the TAIEX and Hang Seng Stock Index (HSI). The performance of the model is compared to Chen's model, Yu's model, the autoregressive (AR) model, the ANFIS model, and the support vector regression (SVR) model. The empirical results show that the proposed model performed the best on predicting the price change.

2.2.3 Optimization of ANNs' forecasting power using swarm intelligence

Zhang & Wu (2009) used an improved bacterial chemotaxis optimization (IBCO) with a MLPNN on the S&P 500 achieving better accuracy, less computational complexity and training time than the traditional MLPNN. Hsieh et al. (2011) suggested a Recurrent Neural Network (RNN) based on Artificial Bee Colony (ABC) algorithm called ABC-RNN as an alternative to improve weights and biases searching. The data is preprocessed using Haar wavelet to eliminate noise before being used in the model. The model generates satisfactory forecasting performance, but the authors point out that the system organization is too complicated and further work on it needs to be done to simplify it. Shen et al. (2011) proposed a Radial Basis Function Neural Network (RBFNN) with artificial fish swarm algorithm (AFSA) to increase forecast ability. The algorithm is compared with Genetic Algorithms (GA), Particle Swarm Optimization (PSO), ARIMA, MLPNN and SVM on the Shanghai stock exchange to asses the model's performance. The result shows that the new model performed better than the classical RBFNN, but it was not the best among the models examined.

Bagheri et al. (2014) presented a new model to forecast exchange rates using Adaptive Network-based Fuzzy Inference System (ANFIS) to process the input data and Quantum-behaved Particle Swarm Optimization (QPSO) to update the ANFIS function. The performance of the proposed model is analyzed using four exchange rates (EUR/USD, USD/JPY, GBP/USD and USD/CHF). The empirical result shows that the suggested model has high predictive power and ability to capture financial patterns; hence, it could be a promising forecasting tool. Pulido et al. (2014) applied an ensemble neural network with fuzzy aggregation and PSO on the Mexican Stock Exchange. The result shows that the models' forecasting performance improved after the integration of PSO. Pradeepkumar & Ravi (2017) presented a PSO trained Quantile Regression Neural Network (PSOQRNN) to predict financial time series volatility. The model is then compared with GARCH, MLP, General Regression Neural Network (GRNN), Group Method of Data Handling (GMDH), RF, Quantile Regression Neural Network (QRNN) and

Quantile Regression Random Forest (QRRF) on forecasting the volatility of eight different financial times series data (USD/JPY, USD/GBP, USD/EUR, USD/INR, Gold Price, Crude Oil Price, S&P 500, and NSE India Stock Index). The result indicates a superiority of the suggested model over the other methodologies mentioned in the paper.

2.2.4 Optimization of ANNs' forecasting power through combinations

Roh (2007) studied three hybrid models by combining the classical MLPNN with Exponentially Weighted Moving Average (EWMA), Generalized Autoregressive Conditional Heteroscedasticity (GARCH), and Exponential GARCH (EGARCH). The models were applied on the Korea Composite Stock Price Index (KOSPI) 200 to predict price volatility. In the analysis, the NN-EGARCH model obtained the best result among the three with a hit ratio of 100% for forecasting periods of less than ten days. Khashei & Bijari (2010) suggested a hybrid model composed by ARIMA and the MLPNN to predict the price change of three datasets Wolf's sunspot data, the Canadian lynx data, and the GBP/USD exchange rate. The numerical result shows that the proposed model achieves higher accuracy than the traditional MLPNN. Maia & de Carvalho (2011) introduced a hybrid model by combining Holt's with the MLPNN to forecast interval-valued time series data, which are data collected in a chronological sequence over time. The empirical result shows that the suggested model outperforms the classical Holts and MLPNN. Wang et al. (2012) applied a hybrid model by combining the Exponential Smoothing Model (ESM), ARIMA and the MLPNN to forecast the price change of the Shenzhen Integrated Index (SZII) and DJIA. The numerical result shows that the new model outperforms the classical models ARIMA, ESM, MLPNN, the equal weight hybrid model (EWH), and RW.

Dai et al. (2012) combined nonlinear independent component analysis (NLICA) with the MLPNN to improve the MLPNN model by extracting independent variables before feeding the data to the neural network. The model is used to forecast Asian stock markets, and it is compared with the Linear independent component analysis neural network (LICA-NN), the Principal Component Analysis Neural Network (PCA-NN) and the MLPNN. The results show that the suggested model improved the prediction performance of the ANN and it also outperforms the other compared models. Kazem et al. (2013) developed a hybrid model on the foundation of chaotic mapping, firefly algorithm and support vector regression. Chaotic mapping and Firefly are used to optimize hyperparameters of the support vector regression. The model was analyzed on NASDAQ stock, and the results show that the model is appropriate to be used as a forecasting tool. Chaâbane (2014) suggested a new hybrid model exploiting the benefits of the Auto-Regressive Fractionally Integrated Moving Average (ARFIMA) model and the MLPNN. The proposed model generated more accurate forecasting results than the traditional ARFIMA and the MLPNN. Rather et al. (2015) suggested a hybrid model constituted by an autoregressive moving average model, an exponential smoothing model, and a RNN to predict the value of the NSE India Stock Index. The proposed model as expected produces more accurate forecast and it is capable of capturing patterns that were difficult to be discovered by classical models.

Kristjanpoller & Minutolo (2015) applied an ANN–GARCH model to forecast gold price volatility. The new model improved forecasting of the gold spot price volatility by 25% and the future price volatility by 38% in comparison to the classical GARCH method. Wang & Wang (2016) introduced a new model combining the MLPNN and the Elman recurrent neural networks with a stochastic time effective function to give more weight to recent information. The new model is used to forecast crude oil price, and it achieved higher accuracy than the MLPNN and the Elman recurrent neural network. Wang & Wang (2017) introduced a new model combining empirical mode decomposition with stochastic time strength neural network (STNN) to predict the NYSE, DAX, FTSE and HSI value. The empirical result shows that the model generates good performance in forecasting market price. Tseng et al. (2008) proposed to integrate Grey-EGARCH into ANN to increase forecasting accuracy. The model was analyzed on the Taiwan index option, and the performance was compared to NN-GARCH. the numerical result shows that the implementation of Grey-EGARCH in ANN produces a better result than implementing GARCH in ANN. Adhikari & Agrawal (2014) combined RW with the MLPNN as an attempt to gather the benefits of both models into a single model. The proposed model is compared with the MLPNN, the Elman ANN (EANN) and RW on the prediction of USD/INR, GBP/USD, S&P 500 and IBM stock price. The result shows the suggested model outperforms the other three models. Wang (2009) suggested a new hybrid model, the Grey-GJR-GARCH neural network to predict the TAIFEX and Taiwan stock index options. Besides, to asses the performance of the proposed model it is compared with GARCH-NN and GJR-NN. The result shows that the new model outperforms the other two models.

2.2.5 Optimization of ANNs' forecasting power using evolutionary algorithms

Hassan et al. (2007) suggested a fusion of the Hidden Markov Model (HMM), MLPNN and Genetic Algorithms (GA) to predict the stock price of Apple, Dell and IBM. The result shows that the suggested model has a forecasting power that is as good as ARIMA's. Nair et al. (2011) implemented a MLPNN with GA to forecast the following day's market value. The model with the help of genetic algorithm could tune the parameters at the end of each trade to obtain better accuracy than the traditional MLPNN. Chang et al. (2012) suggested an evolving partially connected neural network (EPCNN) with a random connection between neurons and an evolutionary algorithm to improve learning. The model was compared with the MLPNN, the TSK fuzzy system, and the multiple regression analysis. The proposed ANN showed promising results making it an up-and-coming model for forecasting. Wang et al. (2015) integrated the adaptive differential evolution (ADE) algorithm in the MLPNN to increase forecasting accuracy by optimizing weight searching of the neurons. The model outperformed the basic algorithms, but it is comparable to other hybrid models like Zhang's hybrid model and Khashei's hybrid model.

Chiroma et al. (2015) implemented an evolutionary neural network based on the MLPNN and GA to predict the West Texas Intermediate crude oil price. The performance of the proposed model was compared with ten MLPNNs based on different learning algorithms (Batch Training With Weight and Bias Learning Rules (B), Fletcher–Powel Conjugate Gradient (CGF), Resilient Backpropagation (RP), Scale Conjugate Gradient (SCG), Levenberg–Marquardt (LM), Gradient Descent with Momentum (GDM), Conjugate Gradient Backpropagation with Fletcher–Reeves Updates (CGF), BFGS Quasi-Newton Backpropagation (BFG), Polak–Ribière Conjugate Gradient (CGP), and One Step Secant (OSS)). The numerical result shows that the proposed model outperforms all the other models. Qiu & Song (2016) analyzed the performance of an optimized neural network based on GA using two types of inputs on the Nikkei 225. The results show the second type of inputs with one feature subset and nine technical indicators achieves higher precision. Khan et al. (2008) implemented a GA based MLPNN to predict the price change of Maruti stock and its performance is compared with the traditional MLPNN. The empirical result shows that the GA based model generates better accuracy than the traditional MLPNN.

2.2.6 Optimization of ANNs' forecasting power using Wavelet

Huang & Wu (2010) implemented a wavelet analysis to process the raw data of major stock indices (NASDAQ, Nikkei 225, KOSPI, CAC 40, FTSE 100, DAX 30, MIB 40, TSX 60, S&P 500 and TWSI) before inserting it in the Recurrent Self-Organizing Map (RSOM), which is used to store temporal features, then multiple kernel partial least square regressors are constructed for the final forecasting. The result shows that the new model defeats the classical MLPNN, SVM and GARCH models. Khandelwal et al. (2015) suggested a hybrid ARIMA and MLPNN model with a Discrete Wavelet Transform (DWT) that decomposes the data into linear and nonlinear components before feeding it to the ANN. The model's forecasting performance was analyzed on the prediction of four datasets (Lynx, GBP/USD, Indian mining and US temperature). The numerical result shows an improvement on the forecasting ability of the model after implementing DWT. Chandar et al. (2016) used an MLPNN with DWT to predict the price change on five different stocks (Tata steel, Wipro, SBI, TCS, and Infosys) and showed that the suggested model produces more accurate forecast than the basic MLPNN.

2.2.7 Studies about the Long Short-Term Memory Neural Network (LSTMNN)

Chen et al. (2015) used the LSTMNN to predict the China stock market. The performance was compared with the random prediction method. The LSTMNN achieved a forecasting accuracy of 27.2% compared to the 14.3% of the random prediction method. Hence, concluding that the LSTMNN is not suitable for the China stock market. Xiong et al. (2015) implemented a LSTMNN to predict the S&P 500 volatility, using mood and macroeconomic factors as inputs. The model generates a MAPE of 24.2% outperforming the linear Ridge/Lasso and the autoregressive GARCH by at least 31%. Jia (2016) explored the effectiveness of implementing the LSTMNN on stock prediction. In the study, the model showed very promising forecasting performance on the prediction of Google's stock price.

Zhuge et al. (2017) applied a LSTMNN with emotional analysis to predict the stock price of stocks traded in the Shanghai stock exchange. The model takes the Shanghai stock index price, the stock exchange data, and the emotional data as input. Furthermore, to asses the performance of the proposed model, it is compared with a LSTMNN without the emotional data. The numerical result shows that adding the emotional data greatly improves the forecasting accuracy. McNally et al. (2018) compared the prediction accuracy of the bayesian optimized recurrent neural network, the LSTMNN and the ARIMA model on the prediction of the Bitcoin price movement. The numerical results show that the LSTMNN outperforms the other two methods,

2.2.8 Studies about the Convolutional Neural Network (CNN)

Tsantekidis et al. (2017) implemented a Convolutional Neural Network (CNN) using large-scale, high-frequency time-series data from the order book of financial exchanges to predict the price movement of five different stocks (the Finnish companies Kesko Oyj, Outokumpu Oyj, Sampo, Rautaruukki and Wartsila Oyj). The model used a dataset of over 4 million trading events and generated better forecasting results than the MLPNN and the SVM. Siripurapu (2014) explore the possibility of implementing CNN to predict the S&P 500 ETF Trust (SPY). The results were not satisfactory since it could not achieve an R^2 greater than 0.

All the study mentioned above are contributing to the improvement of the forecasting ability of ANNs. However, most of them are Hybrid models or modifications of the MLPNN, and relatively fewer research about the LSTMNN and the CNN on the financial forecasting topic has been contributed.

3 Methodology

3.1 Multilayer Perceptron Neural Network (MLPNN)

The multilayer perceptron is one of the most broadly implemented neural network topologies. It is a fully connected neural network where each neuron of a layer is connected to every single neuron of the previous layer (Rosenblatt 1961). Furthermore, The MLPNN has to have three or more layers, and it is constituted by three types of layers:

- **Input layer**, only one in the ANN, where information is inserted, and it is constituted by one or more input neurons.
- **Hidden layer**, one or more in the ANN and it is constituted by hidden neurons with nonlinear activation functions.
- **Output layer**, only one in the ANN, where the result is generated, and it is constituted by one or more output neurons.

3.1.1 Information Processing

Feedforward is a way to processes the information in ANNs, in this procedure information moves in a single direction. Data enters the network through the input layer, and then the sum of the products of the weights and the inputs is calculated in each node of the following layers (Schmidhuber 2015). If the value exceeds some threshold the neuron fires generating a positive value; otherwise it takes a deactivated value which is typically zero or a negative number. Furthermore, the threshold is decided by the activation function of the neurons.

Suppose the total number of layer of our ANN is given by L, we use l to indicate a single layer. $l = 1$ is the input layer, $l = 2$, ... , $l = L-1$ indicate the hidden layer and $l = L$ indicates the output layer. Figure 1 shows an example of a MLPNN structure with four neurons in the input layer, five neurons in the hidden layer and one neuron in the output layer.

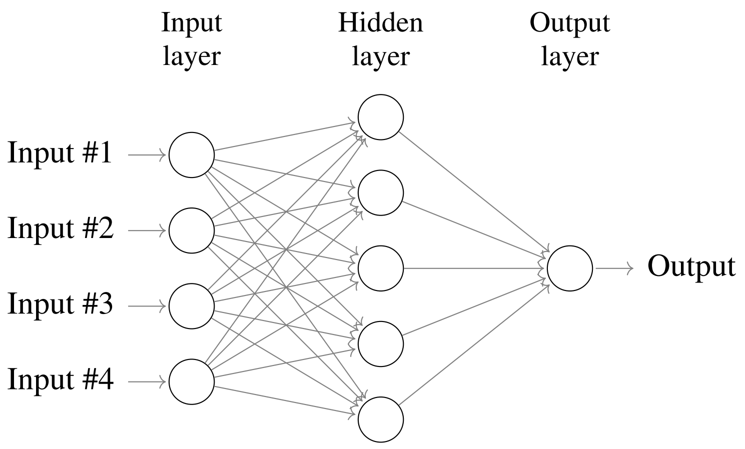


Figure 1: MLPNN with 3 layers

In this paper, we use i_j as the j-th input for the neural network. Besides, w^l_{jk} is used to represent the weight for the connection from the k-th neuron in the $(l-1)$-th layer to the j-th neuron in the l-th layer. Furthermore, we use b^l_j to represent the bias term for the j-th neuron in the l-th layer and z^l_j to represent the input for the j-th neuron in the l-th layer.

$$z^l_j = \begin{cases} \sum\limits_k w^l_{jk} a^{l-1}_k + b^l_j, & \text{If } l > 1 \\ i_j, & \text{Otherwise} \end{cases} \tag{1}$$

Moreover, we use a^l_j as the activation of the j-th neuron in the l-th layer. Using the notations above the activation a^l_j of the j-th neuron in the l-th layer is associated with the activations in the $(l-1)$-th layer by the following equation

$$a^l_j = \begin{cases} \sigma_c(z^l_j) = \sigma_c(\sum_k w^l_{jk} a^{l-1}_k + b^l_j), & \text{If } l > 1 \& l < L \\ z^l_j = \sum_k w^l_{jk} a^{l-1}_k + b^l_j, & \text{If } l == L \end{cases} \tag{2}$$

σ_c is the hyperbolic tangent function $\sigma_c(x) = \tanh(x)$ and it is the activation function for the hidden neurons.

We can rewrite the expression in a matrix form by defining an input vector $I \in \mathbb{R}^d$. The entries of the vector are just the values of i_j. Additionally, we define a weight matrix $w^l \in \mathbb{R}^{n^{(l)} \times n^{(l-1)}}$. The entries of the weight matrix w^l are the weights connecting to the l-th layer of the network; hence, the entries in the j-th row and k-th column is w^l_{kj}. Moreover, for each layer l we define a bias vector $b^l \in \mathbb{R}^{n^{(l)}}$. The entries of the vector are just the value of b^l_j. Furthermore, we define a $z^l \in \mathbb{R}^{n^{(l)}}$ as the input vector for neurons in the l-th layer.

$$z^l = \begin{cases} w^l a^{l-1} + b^l, & \text{If } l > 1 \\ I, & \text{Otherwise} \end{cases} \tag{3}$$

Lastly, we define the activation vector $a^l \in \mathbb{R}^{n^{(l)}}$ whose components are the activations a^l_j. Applying the new notation we can get the vectorized form.

$$a^l = \begin{cases} \sigma_c(z^l) = \sigma_c(w^l a^{l-1} + b^l), & \text{If } l > 1 \& l < L \\ z^l = w^l a^{l-1} + b^l, & \text{If } l == L \end{cases} \tag{4}$$

The superscripts n^l indicates the number of neurons in the in the layer l and d indicates the number of inputs to the network.

3.1.2 Learning Process

Backpropagation is a method used in ANNs in which the error term is propagated from the output layer to the input layer for the purpose of calculating the gradient, so weights can be updated to reduce the value of the cost function (Bengio et al. 2015). The cost function used for MLP is the mean squared error, and it is defined by the following equation.

$$Cost = \frac{1}{2n} \sum_x ||y(x) - a^L(x)||^2 \tag{5}$$

$y(x)$ is the actual real value, and $a^L(x)$ is the value predicted by the MLP. Adam Optimization algorithm (Kingma & Ba 2014) is used instead of the traditional stochastic gradient descent procedure to reduce the cost function and update the network's weight.

3.2 Long Short-Term Memory Neural Network (LSTMNN)

Recurrent neural networks have connections between nodes forming a directed graph along a sequence allowing them to have a dynamic behavior for a time sequence (Li & Wu 2015). Unlike feedforward networks, they can keep an internal state to process a sequence of inputs. The LSTM is an RNN but with an extra unit called LSTM unit composed by a cell, an input gate, an output gate, and a forget gate incorporating time delay to the internal state. The LSTM cell can memorize values of different time intervals and control the flow of the information into and out of the cell using the gates. Figure 2 shows the structure of the LSTM unit.

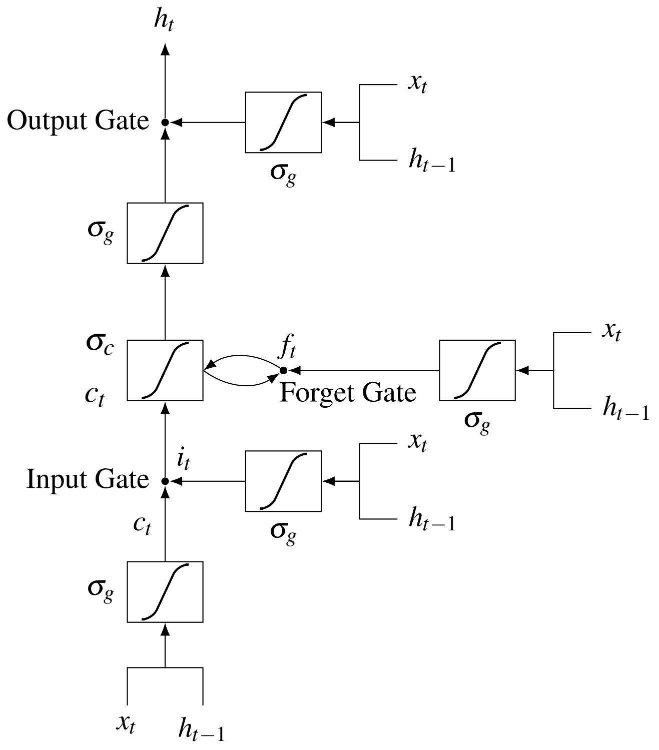


Figure 2: Structure of the LSTM unit

3.2.1 Information Processing

In this paper we use $x_t \in \mathbb{R}^d$ to indicate the input vector given to the network, $h_t \in \mathbb{R}^h$ to represent the output vector of the LSTM unit. Moreover, we use $b \in \mathbb{R}^h$ to represent the bias vector. Additionally, $f_t \in \mathbb{R}^h$ represents the forget gate's activation vector, $i_t \in \mathbb{R}^h$ represents the input gate's activation vector, $o_t \in \mathbb{R}^h$ represents the output gate's activation vector and $c_t \in \mathbb{R}^h$ represents the cell state vector. Furthermore, we use $W \in \mathbb{R}^{hxd}$ to represent the weight matrix between the input and hidden units. Moreover, $U \in \mathbb{R}^{hxh}$ to represent the weight between hidden units. The subscript t indexes time steps, the superscript h indicates the number of hidden units and d indicates the number of inputs. With the given notation we can define the variables with the equations below:

$$f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \tag{6}$$

$$i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \tag{7}$$

$$o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o) \tag{8}$$

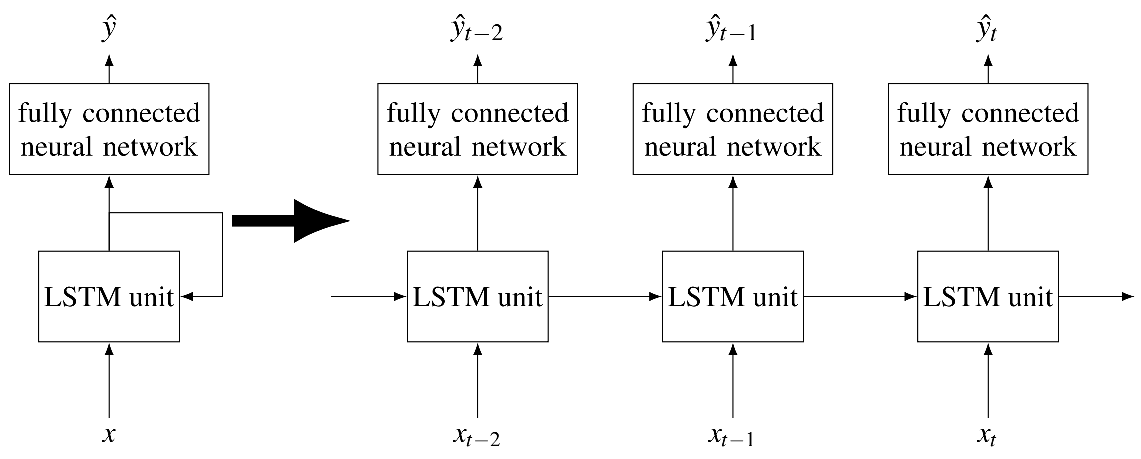
$$c_t = f_t \circ c_{t-1} + i_t \circ \sigma_c(W_c x_t + U_c h_{t-1} + b_c) \tag{9}$$

$$h_t = o_t \circ \sigma_c(c_t) \tag{10}$$

σ_g is the sigmoid function which is defined as $\sigma_g(z) = \frac{1}{1-exp(-z)}$ and σ_c is the hyperbolic tangent $\tanh(x)$.

The forget gate f returns a vector with values between 0 and 1, entries with a value of 0 means completely forget it and it will not be kept by the network, and entries with a value of 1 means keeping it entirely. Moreover, i and o also generate a vector of values between 0 and 1 deciding how complete the information gets in and out of the network. Moreover, the cell state c returns a vector of values between -1 and 1 since it uses σ_c.

The LSTM unit generates an output vector h_t which needs to be inserted into a fully connected neural network to get the prediction of the network $\hat{y}_t$. Therefore, combining the LSTM unit with the fully connected neural network, we get the complete structure of the LSTM network shown in figure 3.


$\hat{y}$
fully connected neural network
LSTM unit
x
$\hat{y}_{t-2}$
$\hat{y}_{t-1}$
$\hat{y}_t$
fully connected neural network
fully connected neural network
fully connected neural network
LSTM unit
LSTM unit
LSTM unit
x_{t-2}
x_{t-1}
x_t

(a) LSTM network

(b) unfolded LSTM network

Figure 3: LSTM network structure

3.2.2 Learning Process

The LSTMNN learns through backpropagation using the mean squared error as the cost function, and it is defined by the following equation:

$$Cost = \frac{1}{2n}\sum_x ||y(x) - \hat{y}(x)||^2$$

Where $y(x)$ is the actual real value, and $\hat{y}(x)$ is the value predicted by the LSTMNN. Moreover, Adam Optimization algorithm (Kingma & Ba 2014) is used to reduce the cost function and update the network's weight in the backpropagation process.

3.3 Convolutional Neural Network (CNN)

It is a deep, feedforward network inspired by the animal visual cortex which is composed by cortical neurons responding to signals only in a restricted region (Matsugu et al. 2003). Therefore, unlike MLP where every neuron is fully connected to neurons of the previous layer. In CNN each neuron is connected to a number of nearby neurons of the previous layer reducing the number of learning parameters dramatically. Moreover, all units of the same layer are connected to the previous layer in the same manner, with exactly the same weight and structure. The network is mainly composed by three layers, the convolutional layer, pooling layer, and fully connected layer.

3.3.1 Information Processing

3.3.1.1 Convolutional layer

It is the core element of a CNN applying convolutional operations to the input. It is a set of learnable filters, which is composed of weights that are updated in the learning process, with a small receptive field that extends through the full depth of the input image. In the forward process, each filter is convolved across the width and height of the input, generating the dot product between the entries of the filter and the input, then an activation function is applied to the result of the dot product to generate a 2D activation map for each filter. The activation maps are then stack along the depth to form the full output volume of the convolutional layer.

The layer has three hyperparameters:

- F, it represents the spatial extent.
- P, it represents the number of zero padding, and it is used to control the output volume spatial size.
- S, it represents the stride and controls how the filter convolves around the input volume. When $S = 1$ we move the filter one pixel at a time.

Therefore, an input volume of size $W_1 \times H_1 \times D_1$ convolved with a convolutional layer with K number of filters, each filter with $F \times F$ spatial extent that slides S pixel at a time using P number of zero padding. It generates a volume size of $W_2 \times H_2 \times D_2$ where:

$$W_2 = (W_1 - F + 2P)/S + 1 \tag{11}$$

$$H_2 = (H_1 - F + 2P)/S + 1 \tag{12}$$

$$D_2 = K \tag{13}$$

since parameters are shared, $F \cdot F \cdot D_1$ number of weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases.

3.3.1.2 Pooling Layer

It is another crucial building block in CNN, which is a form of non-linear down-sampling. Even though, there are many non-linear functions to implement the most popular among them is max pooling. It divides the picture into a number of nonoverlapping rectangles and outputs the maximum of each subregion. The pooling layer is used to progressively reduce the spatial size of the input, to reduce the number of learning parameters and to reduce computation. Furthermore, it operates on every depth of the input independently and resizes it spatially.

Hence, an input volume of size $W_1 \times H_1 \times D_1$ processed by a pooling layer with $F \times F$ spatial extent that slides S pixels at a time produces a volume size of $W_2 \times H_2 \times D_2$ where:

$$W_2 = (W_1 - F)/S + 1 \tag{14}$$

$$H_2 = (H_1 - F)/S + 1 \tag{15}$$

$$D_2 = D_1 \tag{16}$$

Usually, $S = F$ to prevent overlapping between the subregions and zero-padding is commonly set to zero ($P = 0$) for the pooling layers since the purpose is to reduce dimensionality.

3.3.1.3 Fully connected neural network

Finally, after the convolutional and pooling layers, input volume is unrolled into a 1-D vector and fed into a fully connected neural network like the MLP to generate the prediction of the CNN.

Figure 4 shows the structure of a CNN composed by a convolutional layer, using leaky-ReLu (rectified liner unit) as activation function with zero padding to keep the output's dimension equal to the input volume, a pooling layer and a fully connected layer with one hidden layer. Moreoever, leaky-ReLu is defined by the following equation:

$$f(x) = \begin{cases} x, & \text{If } x > 0 \\ 0.01x, & \text{Otherwise} \end{cases} \tag{17}$$

3.3.2 Learning Process

The cost function implemented to penalize the deviation between the real and predicted value is the mean squared error, and it is defined by the following equation.

$$Cost = \frac{1}{2n}\sum_{x} ||y(x) - \hat{y}(x)||^2$$

$y(x)$ is the actual real value, and $\hat{y}(x)$ is the value predicted by the CNN. Adam Optimization algorithm (Kingma & Ba 2014) is used to reduce the cost function and update the network's weight.

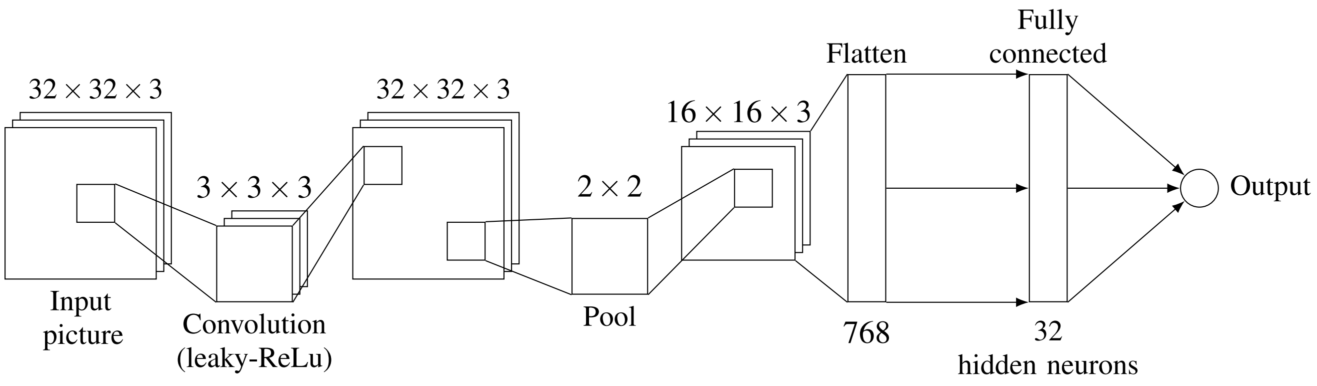


Figure 4: Example of a Convolutional Neural Network's structure

4 Data, Analysis, and Results

4.1 Data description

The data for the study has been collected from DataStream for the CSI 300 from 08/4/2005 to 23/07/2018 (DataStream 2018). The total number of samples for the index is 3467 trading days; each sample is composed by the daily high, low, closing price and trading volume. Furthermore, we split the whole data into a training set (80%) and a testing set (20%). Moreover, the following technical indicators, the 10 days exponential moving average (EMA10), 30 days exponential moving average (EMA30), 10 days rate of change (ROC10), 10 days stochastic oscillator (STO10), 3 days stochastic oscillator moving average (STOMA3), relative strength index (RSI) and accumulation distribution line (ADL), are selected as input features for our models.

EMA10 and EMA30 are picked to identify trends in CSI 300 to give more weight to recent price and to identify cross signals, ROC10 to capture momentum information, RSI to determine when the market is oversold or overbought, ADL to measure the underlying supply and demand of the market. Lastly, STO 10 and STOMA3 are implemented to recognize when the market is oversold or overbought and to identify the trading signal.

Technical indicators	Formula
EMA10, EMA30	A=2/(N+1); (P $\times$ A) + (Previous EMA $\times$ (1 - A))
ROC10	$\frac{\text{Today's Close - Close N period ago}}{\text{Close price N period ago}} \times 100$
STO10	$\frac{\text{Today's close - Lowest low in N periods}}{\text{Highest high in N period - Lowest low in N period}} \times 100$
STOMA3	$\frac{1}{N}\sum_{i=0}^{N-1} STO_{m-i}$
ADL	$\frac{\text{(C.P - L.P) - (H.P - C.P)}}{\text{(H.P - L.P)}} \times$ Volume + previous ADL
RSI	$100 - \frac{100}{1+U/D}$

Table 1: Formula for different technical indicators

Table 1 shows the formula for calculating the technical indicators, where A is the smoothing factor, N is the time period, P is the current price, C.P is the closing price, H.P is the highest price and L.P is the lowest price

4.1.1 Data Normalization

LeCun et al. (2012) found that ANNs learn much faster and more efficiently using input features with values that have a similar range in the training set. Therefore, we normalize the data using the following equation for each feature.

$$\hat{x} = \frac{x - E(x)}{\sqrt{VAR(x)}} \tag{18}$$

where $\hat{x}$ is the normalized feature. Table 2 shows the change on the maximum and minimum of different features before and after performing normalization. We use s_t to denote the vector of the 7 normalized technical indicators at time t and p_t to represent the real price of CSI 300 at time t. More precisely, we set $x_t = s_t$ and $y_t = p_{t+1}$, and consider (x_t, y_t) as a sample for MLPNN and LSTMNN, where x_t is the input and y_t is the desired output.

Technical indicators	Minimum	Maximum
ROC10	-33.64(-5.3)	18.56(2.8)
EMA10	843.97(-1.83)	5717.55(2.88)
EMA30	860.61(-1.82)	5505.62(2.7)
STO10	-291.64(-6.25)	394.1(4.43)
STOMA3	-84.42(-2.57)	233.86(2.17)
RSI	11.88(-2.47)	96,32(2.6)
ADL	-23398047.6(-1.67)	43494459217.1(2.72)

The value after normalization is in parenthesis

Table 2: Value Range

4.1.2 Converting time series to an image

CNN, as mentioned earlier, is an ANN inspired on the visual cortex mainly applied to visual image analysis. Therefore, owing to its nature, we need to encode our features into an image with a structure (W×H×D). We implement the Gramian Angular Field (GAF) to transform each feature into a 2D GAF image (Wang & Oates 2015). First of all, Given the feature $X = \{x_1, x_2 ..., x_n\}$ of n observation we rescale them, so all the values fall in the range of [-1 , 1]:

$$\hat{x_i} = \frac{(x_i - max(X)) + (x_i - min(X))}{max(X) - min(X)} \tag{19}$$

Secondly, we transform the rescaled feature $\hat{X}$ into the polar system by encoding the value as the angular cosine and time stamp as radius:

$$\begin{cases} \phi = \arccos(\hat{x}_i), -1 \leq \hat{x}_i \leq 1, \hat{x}_i \in \hat{X} \\ r = \frac{t_i}{N}, t_i \in \mathbb{N} \end{cases} \tag{20}$$

In the equation above t_i represents time and N is a constant factor to regularize the span of the polar coordinate system. After transforming the rescaled feature into the polar system, we can use the angular perspective by considering the trigonometric sum between each point to identify the temporal correlation within different time intervals. The GAF is defined as follows:

$$G = \begin{pmatrix} \cos(\phi_1 + \phi_1) & \cdots & \cos(\phi_1 + \phi_n \\ \cos(\phi_2 + \phi_1) & \cdots & \cos(\phi_2 + \phi_n) \\ \vdots & \ddots & \vdots \\ \cos(\phi_n + \phi_1) & \cdots & \cos(\phi_n + \phi_n) \end{pmatrix} \tag{21}$$

The equation returns a GAF image of dimension ($n \times n$), and to get an input volume with dimensions ($W \times H \times D$), we stack the features' GAF image along depth axis; hence, $W = H = n$ and D = number of features.

In this study, we use three features EMA10, EMA30, and ADL using 192 observation to generate an input volume with high resolution of dimension $(128 \times 128 \times 3)$. There are two reasons for selecting this dimension for the input volumes:

- In the test, we used different pictures size based on multiples of 32, and it was observed that the higher the resolution, the more the ANN learns generating more accurate results.
- Due to the limited computational power $(192 \times 192 \times 3)$ is the highest resolution the laptop could achieve. Moreover, we selected EMA10, EMA30, and ADL these three technical indicators to create the GAF image for detecting trading signals and measure the underlying supply and demand of the CSI 300.

We use G_t to denote the GAF image at time t and p_t to represent the price of CSI 300 at time t. Then, we set $x_t = G_t$ and $y_t = p_{t+1}$, and consider (x_t, y_t) as a sample for CNN, where x_t is the input and y_t is the desired output.

4.2 Analysis

4.2.1 Weight initialization

Neural network weights are the core of ANNs since they formulate the behavior of the networks. The weights get updated during the learning process, but they need to be initialized first, and the method used to initialize them will have a significant influence on the learning process. There are mainly three ways to initialize them:

1. **Zero initialization**, this method sets all the weights to 0. Hence, the neural network cannot perform symmetry-breaking, all the neurons perform the same calculation, and

the complexity of the whole network would be the same as the complexity of a single neuron.

2. **Random initialization**, this method initializes the weights randomly allowing the ANN to perform symmetry breaking, so the neurons perform different computations.

3. **He's or Xavier's initialization**, the first method considers the number of neurons in the previous layer when initialing the weights, while the second method considers the number of neurons in the previous and current layer to initialize them ((He et al. 2015); (Glorot & Bengio 2010)). Both of them have shown better efficiency and performance compared to the Zero and Random initialization. However, He's is recommended to be used with *ReLu* activation function while Xavier's for *sigmoid* and *tanh* activation functions.

First, Xavier's initialization is used to initialize the weights of MLPNN and LSTMNN since they use sigmoid and tanh activation function (Glorot & Bengio 2010). Equation 22 shows how Xavier's initialization is realized.

$$W_l \sim \mathcal{N}(0, \sqrt{2/\text{number of input} + \text{number of output}}) \tag{22}$$

Second, He's initialization is used for CNN since it uses a rectified linear unit (ReLU) as the activation function. Equation 23 shows how He's initialization is obtained (He et al. 2015).

$$W_l \sim \mathcal{N}(0, \sqrt{2/\text{number of input}}) \tag{23}$$

W_l represents the weights for layer l.

4.2.2 Hyperparameter setting for Adam optimization

Adam optimization is a popular learning algorithm that is able to adapt its learning rate α during the backpropagation process based on the average first moment β_1 (the mean) and the average second moment β_2 (the variance) (Kingma & Ba 2014). Therefore, unlike the classical stochastic gradient descent where the learning rate α is fixed for the whole training process. Adam optimization can update α during the backpropagation, so the weights are updated more efficiently. Furthermore, it has been empirically demonstrated that Adam optimization updates the weights more efficiently and accurately than stochastic gradient descent.

In this paper, the hyperparameters for Adam optimization are defined as $\alpha = 0.01$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\varepsilon = 1e^{-08}$, epsilon is used to prevent numbers to be divided by 0. The hyperparameters used are the same for the three models implemented in the study. The learning rate is chosen after amount of testing, and the other three hyperparameters are selected following the authors advise (Kingma & Ba 2014). The small learning rate required more training time but allow the model to have good convergence leading to better forecasting performance.

4.2.3 MLP training

In this paper we use a MLPNN with the following layers:

1. **Input layer** with 7 input neurons.
2. **Hidden layer** with 10 hidden neurons using hyperbolic tangent as the activation function.
3. **Output layer** with 1 output neuron.

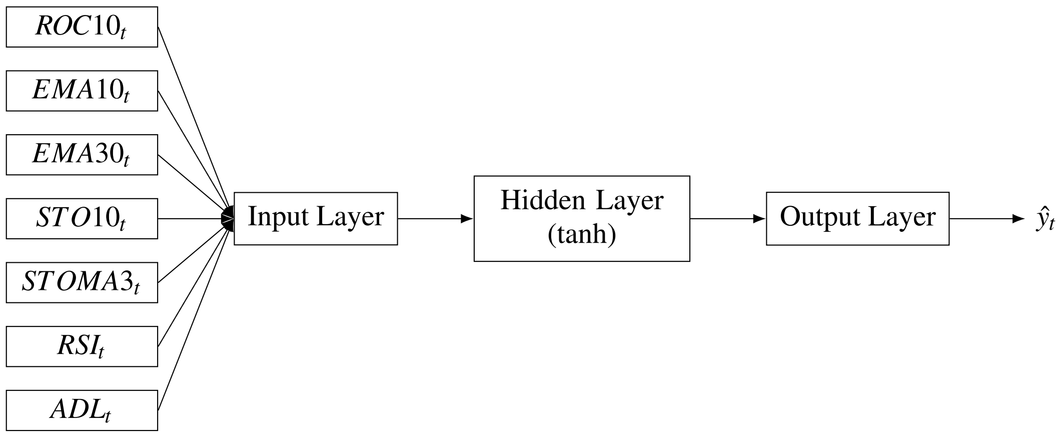


Figure 5: A visual presentation of the evaluated MLPNN model

The structure of this network was selected after many experiments. We tried several neural networks with different number of hidden neurons in a single layer and observed that 10 artificial neurons in the hidden layer generates the best results. Additionally, The MLPNN was trained using mini-batches of size 32, which is chosen after several testings, to improve the computational time and optimize the learning procedure (Masters & Luschi 2018). Moreover, Figure 5 shows the structure of the MLPNN used in the study, and the python code for the model is in Appendix A.

4.2.4 LSTM training

The LSTMNN structure proposed in this paper is composed by the following layers:

1. Input layer with 7 neurons
2. LSTM unit with 8 hidden units
3. Fully connected neural network composed by 2 layers:
   (a) Input layer with 8 input neurons
   (b) Output layer with 1 output neuron

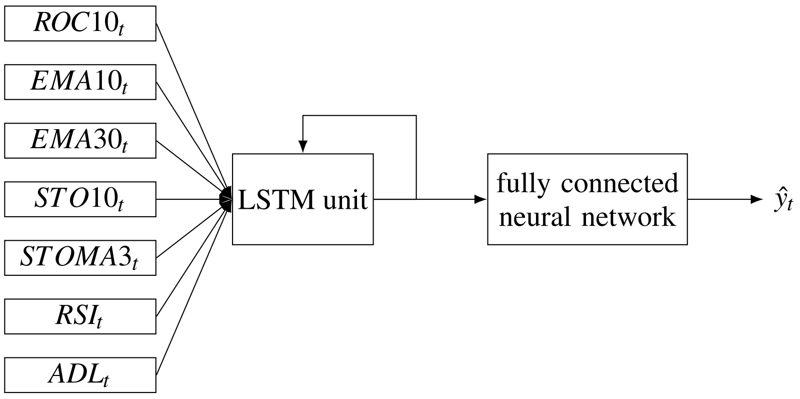


Figure 6: A visual presentation of the evaluated LSTMNN model

The architecture of the LSTMNN is selected after many observations. Neural networks with different number of hidden units were implemented in the experiments, and the results show that a LSTMNN with 8 hidden units generates the most accurate result. Furthermore, The LSTMNN was trained using mini-batches of size 32, which is selected after several testing (Masters & Luschi 2018). Figure 6 shows the architecture of the LSTMNN used in this paper, and the python code for the model is in Appendix B.

4.2.5 CNN training

The architecture of the CNN model used in this paper consists of the following layers:

1. Input layer to accept inputs of the volume of $(192 \times 192 \times 3)$
2. 2D convolutional layer with 3 filters of size (3×3)
3. 2D convolutional layer with 3 filters of size (3×3)
4. Max pooling layer with size (3×3)

5. Fully connected neural network with 4 layers:
    (a) Input layer with 12288 input neurons
    (b) Hidden layer with 4096 hidden neurons using leaky ReLu as the activation function
    (c) Hidden layer with 10 hidden neurons using leaky ReLu as the activation function
    (d) Output layer with 1 output neuron

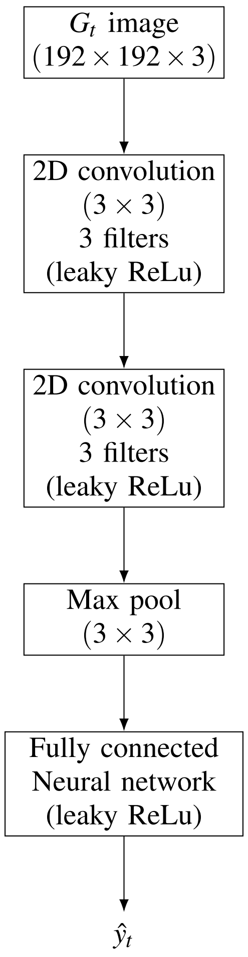


Figure 7: A visual presentation of the evaluated CNN model

We tried many different architectures for our CNN, using different numbers of convolutional and pooling layers with different size and combinations, the fully connected neural network was experimented several times too. Furthermore, the dimension of the input volume is the highest our hardware could manage, as mentioned earlier the higher the resolution of the input image, the more the network learns. Moreover, the CNN model was trained using mini-batches of size 32 which is selected after amounts of testing (Masters & Luschi 2018). Figure 7 shows the structure of the CNN used in the study, and the python code for the model can be found in Appendix C.

4.3 Model performance measurents

In literature, the mean square error (MSE), the mean absolute deviate (MAD), the mean absolute percentage error (MAPE) and R^2 are generally used for evaluating the forecasting performance of ANNs on the price change. These are obtained using the following formulas in Table 3.

Performance indicators	Formula
MSE	$\frac{1}{N}\sum_k^N (y_k - \hat{y}_k)^2$
MAD	$\frac{1}{N}\sum_k^N \lvert y_k - \hat{y}_k\rvert$
MAPE	$\frac{1}{N}\sum_k^N \lvert\frac{y_k - \hat{y}_k}{y_k}\rvert$
SS_{tot}	$\sum_k^N (y_k - \bar{y})^2$
SS_{res}	$\sum_k^N (\hat{y}_k - y_k)^2$
R^2	$1 - \frac{SS_{res}}{SS_{tot}}$

Table 3: Equations for the perfomance indicators

Where y_k indicates the actual price of the CSI 300 in the k-th observation, $\bar{y}$ represents the mean of the actual prices, $\hat{y}_k$ represents the predicted price of the ANN for the k-th observation and N is the total number of observations.

4.4 Results

Figure 8 plots the actual price and the price predicted by the three models. From the graph, we can see that the MLPNN and LSTMNN forecasted trend follows quite closely to the trend of the actual value of the CSI 300, but the MLPNN does better overall with prediction deviating less from the actual price compared to the LSTMNN. Besides, the CNN is the least accurate, its deviation from the actual trend is much greater than the two other models.

Table 4 shows the forecasting performance of the three ANNs. It can be observed that all of them have some forecasting power achieving a MAPE below 10% and a R^2 above 0. Among them, the MLPNN performes the best in all the performance measurements. It achieves a MAPE below 5% and a R^2 close to 1 matching the result of previous studies (Cao et al. (2005); Naeini et al. (2010); Vaisla & Bhatt (2010); Masoud (2014); Song et al. (2018)).

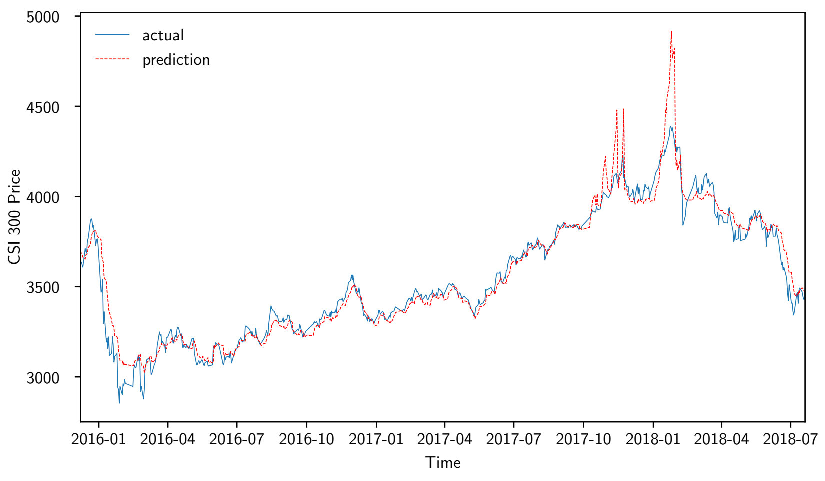
The performance of the LSTMNN ranks second with a MAD and MAPE that is almost 30% greater than the MLPNN, but still below 2% which is much better than the 24.2% achieved by Xiong et al. (2015) using mood and macroeconomic factors as inputs. Therefore, we observe a superiority on using technical indicators over the mood and macroeconomic factors as inputs for the LSTMNN on financial prediction. Moreover, the numerical result indicates that even though the LSTMNN is not as good as the MLPNN, it is still a very promising forecasting tool to predict the China stock market contradicting the results obtained by Chen et al. (2015), in their paper, they concluded that the LSTMNN is not a suitable forecasting model for the China stock market.

Moreover, the CNN performed the worst among the implemented models with a MAD and MAPE that is around 4 times more than the MLPNN's, a MSE that is around 12 times larger and a R^2 that is at least 3 times smaller. The implemented CNN model achieved a greater R^2 than the model implemented by Siripurapu (2014). However, the CNN could not perform better than the MLPNN, opposing the results obtained in the study realized by Tsantekidis et al. (2017). The superiority of the CNN model of Tsantekidis et al. (2017) could be due to two reasons:

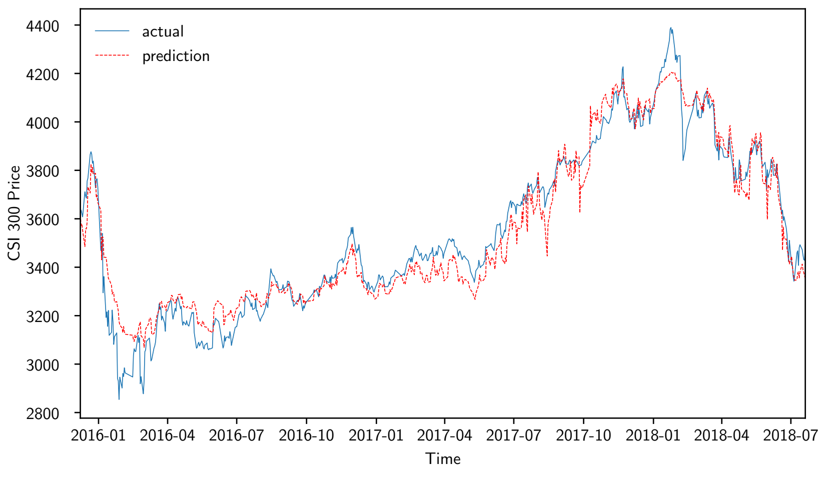
1. A superiority on the amount of the data used to train the network, they used a dataset with more than 4 million samples, while this study is conducted using 3467 samples.
2. A superiority of the architecture implemented.

Method	MLP	LSTM	CNN
MSE	6707.66	6777.64	83982.18
MAD	50.65	65.06	221.04
MAPE	0.014	0.018	0.063
R^2	0.942	0.941	0.275

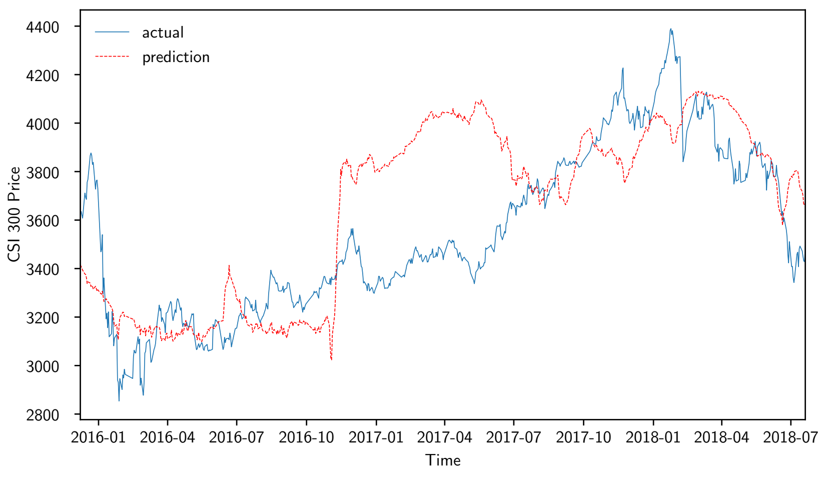
Table 4: Result of the three ANNs



(a) MLPNN's forecast



(b) LSTMNN's forecast



(c) CNN's forecast

Figure 8: comparison of the three models implemented

5 Conclusion & Further Work

In this paper, we analyze the forecasting performance of three ANNs. In order to show the difference in accuracy, we implemented all the models on the same set of data from the CSI 300. The results indicate that the three ANN models are all able to learn some relevant information from past price with evidence from the prediction performance on the CSI 300 using four different measurements. We reach to the conclusion that the MLPNN is the most suitable model among the three networks to be used as a forecasting tool, as it has the best forecasting power. Additionally, the performance of the LSTMNN is very close to the MLPNN being a very promising model for financial forecast. Furthermore, the CNN is the least suitable generating the least accurate prediction among the tested models. However, the low performance of the CNN could be caused by hardware contraints and time limit, as mentioned earlier in the paper, limiting the analysis of the CNN model implemented in the study.

Therefore, further investigation, specially on the CNN model, needs to be done in order to generalize the conclusion, since according to Tsantekidis et al. (2017), the CNN is capable of generating better forecasting performance than the MLPNN. Moreover, future work should also focus on the investigation of more complex neural networks to accomplish the current task, and on the research of other possible architectures to improve the models implemented in this paper especially the LSTMNN and CNN. These two models have shown excellent performance in other fields and generated optimistic results in this study, but there is no much research about them in the financial forecasting field compared to the MLPNN. The results of these further researches should lead us to new powerfull models for financial time serires forecasting.

References

Abu-Mostafa, Y. S. & Atiya, A. F. (1996), 'Introduction to financial forecasting', *Applied Intelligence* **6**(3), 205–213.

Adhikari, R. & Agrawal, R. (2014), 'A combination of artificial neural network and random walk models for financial time series forecasting', *Neural Computing and Applications* **24**(6), 1441–1449.

Bagheri, A., Peyhani, H. M. & Akbari, M. (2014), 'Financial forecasting using anfis networks with quantum-behaved particle swarm optimization', *Expert Systems with Applications* **41**(14), 6235–6250.

Bengio, Y., Goodfellow, I. J. & Courville, A. (2015), 'Deep learning', *Nature* **521**(7553), 436–444.

Cao, Q., Leggio, K. B. & Schniederjans, M. J. (2005), 'A comparison between fama and french's model and artificial neural networks in predicting the chinese stock market', *Computers & Operations Research* **32**(10), 2499–2512.

Chaâbane, N. (2014), 'A hybrid arfima and neural network model for electricity price prediction', *International Journal of Electrical Power & Energy Systems* **55**, 187–194.

Chandar, S. K., Sumathi, M. & Sivanandam, S. (2016), 'Prediction of stock market price using hybrid of wavelet transform and artificial neural network', *Indian Journal of Science and Technology* **9**(8).

Chang, P.-C. et al. (2012), 'A novel model by evolving partially connected neural network for stock price trend forecasting', *Expert Systems with Applications* **39**(1), 611–620.

Chen, K., Zhou, Y. & Dai, F. (2015), A lstm-based method for stock returns prediction: A case study of china stock market, *in* 'Big Data (Big Data), 2015 IEEE International Conference on', IEEE, pp. 2823–2824.

Chen, W.-H., Shih, J.-Y. & Wu, S. (2006), 'Comparison of support-vector machines and back propagation neural networks in forecasting the six major asian stock markets', *International Journal of Electronic Finance* **1**(1), 49–67.

Chiroma, H., Abdulkareem, S. & Herawan, T. (2015), 'Evolutionary neural network model for west texas intermediate crude oil price prediction', *Applied Energy* **142**, 266–273.

Constantinou, E., Georgiades, R., Kazandjian, A. & Kouretas, G. P. (2006), 'Regime switching and artificial neural network forecasting of the cyprus stock exchange daily returns', *International Journal of Finance & Economics* **11**(4), 371–383.

Dai, W., Wu, J.-Y. & Lu, C.-J. (2012), 'Combining nonlinear independent component analysis and neural network for the prediction of asian stock market indexes', *Expert systems with applications* **39**(4), 4444–4452.

DataStream (2018), 'High, low, close and volume of csi 300'. Available at: http://data.worldbank.org [Accessed 10 Jul. 2018].

De Gooijer, J. G. & Hyndman, R. J. (2006), '25 years of time series forecasting', *International journal of forecasting* **22**(3), 443–473.

Dutta, G., Jha, P., Laha, A. K. & Mohan, N. (2006), 'Artificial neural network models for forecasting stock price index in the bombay stock exchange', *Journal of Emerging Market Finance* **5**(3), 283–295.

Glorot, X. & Bengio, Y. (2010), Understanding the difficulty of training deep feedforward neural networks, *in* 'Proceedings of the thirteenth international conference on artificial intelligence and statistics', pp. 249–256.

Guresen, E., Kayakutlu, G. & Daim, T. U. (2011), 'Using artificial neural network models in stock market index prediction', *Expert Systems with Applications* **38**(8), 10389–10397.

Hamzaçebi, C., Akay, D. & Kutay, F. (2009), 'Comparison of direct and iterative artificial neural network forecast approaches in multi-periodic time series forecasting', *Expert Systems with Applications* **36**(2), 3839–3844.

Hassan, M. R., Nath, B. & Kirley, M. (2007), 'A fusion model of hmm, ann and ga for stock market forecasting', *Expert systems with Applications* **33**(1), 171–180.

He, K., Zhang, X., Ren, S. & Sun, J. (2015), Delving deep into rectifiers: Surpassing human-level performance on imagenet classification, *in* 'Proceedings of the IEEE international conference on computer vision', pp. 1026–1034.

Hsieh, T.-J., Hsiao, H.-F. & Yeh, W.-C. (2011), 'Forecasting stock markets using wavelet transforms and recurrent neural networks: An integrated system based on artificial bee colony algorithm', *Applied soft computing* **11**(2), 2510–2525.

Huang, S.-C. & Wu, T.-K. (2010), ‘Integrating recurrent som with wavelet-based kernel partial least square regressions for financial forecasting’, *Expert Systems with Applications* **37**(8), 5698–5705.

Huarng, K. & Yu, T. H.-K. (2006), ‘The application of neural networks to forecast fuzzy time series’, *Physica A: Statistical Mechanics and its Applications* **363**(2), 481–491.

Jia, H. (2016), ‘Investigation into the effectiveness of long short term memory networks for stock price prediction’, *arXiv preprint arXiv:1603.07893* .

Kara, Y., Boyacioglu, M. A. & Baykan, Ö. K. (2011), ‘Predicting direction of stock price index movement using artificial neural networks and support vector machines: The sample of the istanbul stock exchange’, *Expert systems with Applications* **38**(5), 5311–5319.

Kazem, A., Sharifi, E., Hussain, F. K., Saberi, M. & Hussain, O. K. (2013), ‘Support vector regression with chaos-based firefly algorithm for stock market price forecasting’, *Applied soft computing* **13**(2), 947–958.

Khan, A. U., Bandopadhyaya, T. & Sharma, S. (2008), ‘Genetic algorithm based backpropagation neural network performs better than backpropagation neural network in stock rates prediction’, *Journal of Computer Science and Network Security* **8**(7), 162–166.

Khandelwal, I., Adhikari, R. & Verma, G. (2015), ‘Time series forecasting using hybrid arima and ann models based on dwt decomposition’, *Procedia Computer Science* **48**, 173–179.

Khashei, M. & Bijari, M. (2010), ‘An artificial neural network (p, d, q) model for timeseries forecasting’, *Expert Systems with applications* **37**(1), 479–489.

Kingma, D. P. & Ba, J. (2014), ‘Adam: A method for stochastic optimization’, *arXiv preprint arXiv:1412.6980* .

Kristjanpoller, W. & Minutolo, M. C. (2015), ‘Gold price volatility: A forecasting approach using the artificial neural network–garch model’, *Expert Systems with Applications* **42**(20), 7245–7251.

Lawrence, S., Giles, C. L., Tsoi, A. C. & Back, A. D. (1997), ‘Face recognition: A convolutional neural-network approach’, *IEEE transactions on neural networks* **8**(1), 98–113.

LeCun, Y. A., Bottou, L., Orr, G. B. & Müller, K.-R. (2012), Efficient backprop, *in* ‘Neural networks: Tricks of the trade’, Springer, pp. 9–48.

Li, X. & Wu, X. (2015), Constructing long short-term memory based deep recurrent neural networks for large vocabulary speech recognition, *in* 'Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on', IEEE, pp. 4520–4524.

Maia, A. L. S. & de Carvalho, F. d. A. (2011), 'Holt's exponential smoothing and neural network models for forecasting interval-valued time series', *International Journal of Forecasting* **27**(3), 740–759.

Masoud, N. (2014), 'Predicting direction of stock prices index movement using artificial neural networks: The case of libyan financial market', *British Journal of Economics, Management & Trade* **4**(4), 597–619.

Masters, D. & Luschi, C. (2018), 'Revisiting small batch training for deep neural networks', *arXiv preprint arXiv:1804.07612* .

Matsugu, M., Mori, K., Mitari, Y. & Kaneda, Y. (2003), 'Subject independent facial expression recognition with robust face detection using a convolutional neural network', *Neural Networks* **16**(5-6), 555–559.

McNally, S., Roche, J. & Caton, S. (2018), Predicting the price of bitcoin using machine learning, *in* 'Parallel, Distributed and Network-based Processing (PDP), 2018 26th Euromicro International Conference on', IEEE, pp. 339–343.

Naeini, M. P., Taremian, H. & Hashemi, H. B. (2010), Stock market value prediction using neural networks, *in* 'Computer Information Systems and Industrial Management Applications (CISIM), 2010 International Conference on', IEEE, pp. 132–136.

Nair, B. B., Sai, S. G., Naveen, A., Lakshmi, A., Venkatesh, G. & Mohandas, V. (2011), A ga-artificial neural network hybrid system for financial time series forecasting, *in* 'Information Technology and Mobile Communication', Springer, pp. 499–506.

O'Connor, N. & Madden, M. G. (2006), A neural network approach to predicting stock exchange movements using external factors, *in* 'Applications and Innovations in Intelligent Systems XIII', Springer, pp. 64–77.

Oh, K. J. & Kim, K.-j. (2002), 'Analyzing stock market tick data using piecewise nonlinear model', *Expert Systems with Applications* **22**(3), 249–255.

Orhan, U., Hekim, M. & Ozer, M. (2011), 'Eeg signals classification using the k-means clustering and a multilayer perceptron neural network model', *Expert Systems with Applications* **38**(10), 13475–13481.

Panda, C. & Narasimhan, V. (2006), ‘Predicting stock returns: an experiment of the artificial neural network in indian stock market’, *South Asia economic journal* **7**(2), 205–218.

Patel, J., Shah, S., Thakkar, P. & Kotecha, K. (2015), ‘Predicting stock and stock price index movement using trend deterministic data preparation and machine learning techniques’, *Expert Systems with Applications* **42**(1), 259–268.

Pradeepkumar, D. & Ravi, V. (2017), ‘Forecasting financial time series volatility using particle swarm optimization trained quantile regression neural network’, *Applied Soft Computing* **58**, 35–52.

Pulido, M., Melin, P. & Castillo, O. (2014), ‘Particle swarm optimization of ensemble neural networks with fuzzy aggregation for time series prediction of the mexican stock exchange’, *Information Sciences* **280**, 188–204.

Qiu, M. & Song, Y. (2016), ‘Predicting the direction of stock market index movement using an optimized artificial neural network model’, *PloS one* **11**(5), e0155133.

Rather, A. M., Agarwal, A. & Sastry, V. (2015), ‘Recurrent neural network and a hybrid model for prediction of stock returns’, *Expert Systems with Applications* **42**(6), 3234–3241.

Roh, T. H. (2007), ‘Forecasting the volatility of stock price index’, *Expert Systems with Applications* **33**(4), 916–922.

Rosenblatt, F. (1961), Principles of neurodynamics. perceptrons and the theory of brain mechanisms, Technical report, CORNELL AERONAUTICAL LAB INC BUFFALO NY.

Schmidhuber, J. (2015), ‘Deep learning in neural networks: An overview’, *Neural networks* **61**, 85–117.

Shen, W., Guo, X., Wu, C. & Wu, D. (2011), ‘Forecasting stock indices using radial basis function neural networks optimized by artificial fish swarm algorithm’, *Knowledge-Based Systems* **24**(3), 378–385.

Siripurapu, A. (2014), ‘Convolutional networks for stock trading’.

Song, Y.-G., Zhou, Y.-L. & Han, R.-J. (2018), ‘Neural networks for stock price prediction’, *arXiv preprint arXiv:1805.11317* .

Sundermeyer, M., Schlüter, R. & Ney, H. (2012), Lstm neural networks for language modeling, *in* ‘Thirteenth annual conference of the international speech communication association’.

Tsantekidis, A., Passalis, N., Tefas, A., Kanniainen, J., Gabbouj, M. & Iosifidis, A. (2017), Forecasting stock prices from the limit order book using convolutional neural networks, *in* 'Business Informatics (CBI), 2017 IEEE 19th Conference on', Vol. 1, IEEE, pp. 7–12.

Tseng, C.-H., Cheng, S.-T., Wang, Y.-H. & Peng, J.-T. (2008), 'Artificial neural network model of the hybrid egarch volatility of the taiwan stock index option prices', *Physica A: Statistical Mechanics and its Applications* **387**(13), 3192–3200.

Vaisla, K. S. & Bhatt, A. K. (2010), 'An analysis of the performance of artificial neural network technique for stock market forecasting', *International Journal on Computer Science and Engineering* **2**(6), 2104–2109.

Van Gerven, M. & Bohte, S. (2018), *Artificial neural networks as models of neural information processing*, Frontiers Media SA.

Wang, J.-J., Wang, J.-Z., Zhang, Z.-G. & Guo, S.-P. (2012), 'Stock index forecasting based on a hybrid model', *Omega* **40**(6), 758–766.

Wang, J. & Wang, J. (2016), 'Forecasting energy market indices with recurrent neural networks: Case study of crude oil price fluctuations', *Energy* **102**, 365–374.

Wang, J. & Wang, J. (2017), 'Forecasting stochastic neural network based on financial empirical mode decomposition', *Neural Networks* **90**, 8–20.

Wang, L., Zeng, Y. & Chen, T. (2015), 'Back propagation neural network with adaptive differential evolution algorithm for time series forecasting', *Expert Systems with Applications* **42**(2), 855–863.

Wang, Y.-F. (2002), 'Predicting stock price using fuzzy grey prediction system', *Expert systems with applications* **22**(1), 33–38.

Wang, Y.-H. (2009), 'Nonlinear neural network forecasting model for stock index option price: Hybrid gjr–garch approach', *Expert Systems with Applications* **36**(1), 564–570.

Wang, Z. & Oates, T. (2015), 'Imaging time-series to improve classification and imputation', *arXiv preprint arXiv:1506.00327* .

Wei, L.-Y. (2016), 'A hybrid anfis model based on empirical mode decomposition for stock time series forecasting', *Applied Soft Computing* **42**, 368–376.

Xiong, R., Nichols, E. P. & Shen, Y. (2015), 'Deep learning stock volatility with google domestic trends', *arXiv preprint arXiv:1512.04916* .

Yu, T. H.-K. & Huarng, K.-H. (2008), ‘A bivariate fuzzy time series model to forecast the taiex’, *Expert Systems with Applications* **34**(4), 2945–2952.

Zhang, Y.-Q. & Wan, X. (2007), ‘Statistical fuzzy interval neural networks for currency exchange rate time series prediction’, *Applied Soft Computing* **7**(4), 1149–1156.

Zhang, Y. & Wu, L. (2009), ‘Stock market prediction of s&p 500 via combination of improved bco approach and bp neural network’, *Expert systems with applications* **36**(5), 8849–8854.

Zhuge, Q., Xu, L. & Zhang, G. (2017), ‘Lstm neural network with emotional analysis for prediction of stock price.’, *Engineering Letters* **25**(2).

A Python code for the MLPNN

the code can be downloaded from https://github.com/AndyChenXia/ANNs-to-predict-the-CSI-300.git

```
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
import pandas as pd
import math
import datetime
import os


os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
#exponential moving average for 10
def eMovingAverage(x,n):
        mv10 = []
        multiplier = 2/(n+1)

        mv10.append(np.sum(x[0:n]) / n)

        for i in range(n,len(x)):
                mv10.append(((x[i] - mv10[i-(n)]) * multiplier
                   ) + mv10[i-(n)])
        return mv10

## simple moving average
def movingAverage(x,n):
        mv = []
        for i in range(n-1,len(x)):
                mv.append(np.sum(x[i-n-1:i])/n)
        return mv

## ADL
def adl_cal(high,low,close,volume):
        acumulator = np.zeros(len(low))
        multiplier1 = ((close[0] - low[0]) - (high[0] - close
           [0])) / (high[0] - low[0])
        acumulator[0] = multiplier1 * volume[0]

        for i in range(1,len(low)):
                flow_multiplier = ((close[i] - low[i]) - (high
                   [i] - close[i])) / (high[i] - low[i])
                flow_volume = flow_multiplier * volume[i]
                acumulator[i] = acumulator[i-1] + flow_volume
        return acumulator

## momentum ROC
def momentum(x):
        acumulator = []
        counter = 0
        for i in range(10,len(x)):
                acumulator.append(((x[i] - x[counter])/x[i]) *
                    100)
                counter = counter + 1
        return acumulator

## len check
def lenCheck(x,y):
        return (len(x) == len(y))

## stochastic oscillator including the same day ---- 10 days
def stochasticOscillator(x):
        acumulator = []
        for i in range(9,len(x)):
                recent_close = x[i]
                high = np.max(x[i-9:i])
                low = np.min(x[i-9:i])
                acumulator.append(((recent_close - low) / (
                   high - low)) * 100)
        return acumulator

def profit(x):
        acumulator = []
        for i in range(len(x)-1):
                result = x[i] - x[i+1]
                if result > 0:
                        acumulator.append(1)
                else:
                        acumulator.append(0)
        return acumulator

## RSI
def rsiCalculation(x):
        loss = []
        gain = []
        rsi = []
        for i in range(1,11):
                result = x[i] - x[i-1]

                if result >= 0.0:
                        gain.append(result)
                else:
                        loss.append(abs(result))

        first_gains = np.sum(gain) / 10
        first_loss = np.sum(loss) / 10



        rs = first_gains/ first_loss
        rsi.append(100 - (100 / (1+rs)))

        average_gain = []
        average_loss = []
        average_gain.append(first_gains)
        average_loss.append(first_loss)

        for i in range(11,len(x)):

                result = x[i] - x[i-1]
                averageGain = 0
                averageLoss = 0

                if result >= 0.0:
                        averageGain = ((average_gain[i-11] *
                           9) + result) / 10
                        averageLoss = ((average_loss[i-11] *
                           9)) / 10

                        average_gain.append(averageGain)
                        average_loss.append(averageLoss)
                else:
                        averageGain = ((average_gain[i-11] *
                           9)) / 10
                        averageLoss = ((average_loss[i-11] *
                           9) + abs(result)) / 10

                        average_gain.append(averageGain)
                        average_loss.append(averageLoss)

                rs = averageGain/averageLoss
                rsi.append(100 - (100 / (1+rs)))

        return rsi

##data normalization
def normalization(x):
        mean = np.mean(x)
        deviation = np.std(x)
        norm_data = (x - mean) / deviation
        return mean, deviation , norm_data
##
def max_min(x):
        print(np.min(x),np.max(x))

#getting CSI data
data = pd.read_csv("CSIData2.csv")

date = np.array(data['Name'])
csi_high = np.array(data['PRICE_HIGH'],dtype = float)
csi_low = np.array(data['PRICE_LOW'],dtype = float)
csi_close = np.array(data['PRICE_INDEX'],dtype = float)
csi_volume = np.array(data['VOLUME'],dtype = float)

logic = np.isnan(csi_volume)



#for removing nans
date = date[~logic]
csi_high = csi_high[~logic]
csi_low = csi_low[~logic]
csi_close = csi_close[~logic]
csi_volume = csi_volume[~logic]

dates = [datetime.datetime.strptime(date1, '%m/%d/%Y') for
    date1 in date]

#exchange data
data2 = np.genfromtxt("exchange.csv",delimiter = ",",
    skip_header=2)
exchange = data2[:,1]
exchange = exchange[~logic]



# 10 days exponential moving average
```

```
emove10_close =  eMovingAverage(csi_close ,10)

## 10 days momentum
momentum10_close = momentum(csi_close)

# 10 days stochastic oscillator && 3 period moving average of
   oscillator
oscillator10_close = stochasticOscillator(csi_close)
oscilaltor10movingAverage_close = movingAverage(
   oscillator10_close ,3)
## 30 days exponential moving average
emove30_close = eMovingAverage(csi_close ,30)

## RSI calculation 10 days
rsi_close = rsiCalculation(csi_close)



## Adjusting data length for training
adjustment = len(dates) - len(emove30_close)
csi_high = csi_high[adjustment:]
csi_low = csi_low[adjustment:]
csi_close = csi_close[adjustment:]
csi_volume = csi_volume[adjustment:]

momentum10_close = momentum10_close[adjustment -10:-1]
emove10_close = emove10_close[adjustment -9:-1]
oscillator10_close = oscillator10_close[adjustment -9:-1]
oscilaltor10movingAverage_close =
   oscilaltor10movingAverage_close[adjustment -11:-1]
rsi_close = rsi_close[adjustment -10:-1]
emove30_close = emove30_close[0:-1]

## Accumulation distribution line ADL
adl = adl_cal(csi_high ,csi_low ,csi_close ,csi_volume)
adl = adl[0:-1]
```

```
## data normalization
sample = len(momentum10_close)
sample_8 = math.ceil(sample *.8)
sample_2 = math.floor(sample * .2)

norm_momentum10_close = normalization(momentum10_close[:
    sample_8])
norm_emove10_close = normalization(emove10_close[:sample_8])
norm_emove30_close = normalization(emove30_close[:sample_8])
norm_oscillator10_close = normalization(oscillator10_close[:
    sample_8])
norm_oscilaltor10movingAverage_close = normalization(
    oscilaltor10movingAverage_close[:sample_8])
norm_rsi_close = normalization(rsi_close[:sample_8])
norm_adl = normalization(adl[:sample_8])
norm_csi_close = normalization(csi_close[:sample_8])
norm_exchange = normalization(exchange[:sample_8])

data_set = np.vstack((norm_momentum10_close[2],
    norm_emove10_close[2],norm_emove30_close[2],
    norm_oscillator10_close[2],
    norm_oscilaltor10movingAverage_close[2],norm_rsi_close[2],
    norm_adl[2]))
data_set = data_set.T

norm_momentum10_close1 = (momentum10_close[sample_8:sample] -
    norm_momentum10_close[0]) / norm_momentum10_close[1]
norm_emove10_close1 =  (emove10_close[sample_8:sample] -
    norm_emove10_close[0])  / norm_emove10_close[1]
norm_emove30_close1 =  (emove30_close[sample_8:sample] -
    norm_emove30_close[0])  / norm_emove30_close[1]
norm_oscillator10_close1 =  (oscillator10_close[sample_8:
    sample] - norm_oscillator10_close[0])  /
    norm_oscillator10_close[1]
```

```
norm_oscilaltor10movingAverage_close1 = (
   oscilaltor10movingAverage_close[sample_8:sample] -
   norm_oscilaltor10movingAverage_close[0]) /
   norm_oscilaltor10movingAverage_close[1]
norm_rsi_close1 = (rsi_close[sample_8:sample] -
   norm_rsi_close[0]) / norm_rsi_close[1]
norm_adl1 = (adl[sample_8:sample] - norm_adl[0]) / norm_adl
   [1]
norm_csi_close1 = (csi_close[sample_8:sample] -
   norm_csi_close[0]) / norm_csi_close[1]
norm_exchange1 = (exchange[sample_8:sample] - norm_exchange
   [0]) / norm_exchange[1]

training_set = np.vstack((norm_momentum10_close1,
   norm_emove10_close1,norm_emove30_close1,
   norm_oscillator10_close1,
   norm_oscilaltor10movingAverage_close1,norm_rsi_close1,
   norm_adl1))
training_set = training_set.T

num_input_neurons = 7
num_hidden_neurons_1 = 10
num_output_neurons = 1
learning_rate = 0.01
lambda1 = 0.01
epochs = 2000

input_x = tf.placeholder(dtype = tf.float32,shape = (None,
   num_input_neurons))
y_true = tf.placeholder(dtype = tf.float32,shape = (None))


weight_1 = tf.get_variable(name = "w1",shape=(
   num_input_neurons, num_hidden_neurons_1), initializer=tf.
   keras.initializers.he_normal())
```

```
b_1 = tf.get_variable(name = "b1",shape=(1,
    num_hidden_neurons_1), initializer=tf.contrib.layers.
    xavier_initializer())
weight_2 = tf.get_variable(name = "w2",shape=(
    num_hidden_neurons_1,num_output_neurons), initializer=tf.
    keras.initializers.he_normal())
b_2 = tf.get_variable(name = "b2",shape=(1, num_output_neurons
    ), initializer=tf.contrib.layers.xavier_initializer())

## feed forward
z_1 = tf.add(tf.matmul(input_x ,weight_1),b_1)
a_1 = tf.tanh(z_1)
z_2 = tf.add(tf.matmul(a_1 ,weight_2),b_2)


weight_1_square = tf.square(weight_1)
weight_2_square = tf.square(weight_2)

weight_1_sum = tf.reduce_sum(weight_1_square)
weight_2_sum = tf.reduce_sum(weight_2_square)
weight_sum = weight_1_sum + weight_2_sum

cost_loss = tf.reduce_mean(tf.square(y_true - z_2))
accuracy = tf.reduce_mean(tf.square(y_true - z_2))
optimizer = tf.train.AdamOptimizer( learning_rate =
    learning_rate , beta1 = 0.9,beta2=0.999, epsilon=1e-08,)
train = optimizer.minimize(cost_loss)




batch_size = 32

y_predicted = []
y_predicted1 = []
```

```
init = tf.global_variables_initializer()
with tf.Session() as sess:
        sess.run(init)
        # training process
        for steps in range(0,epochs):
                for i in range(0,sample_8 , batch_size):

                        sess.run(train , feed_dict = {input_x:
                           data_set[i: i + batch_size],y_true:
                           csi_close[i+1: i + batch_size + 1]})

        for i in range(sample_8):
                ans = sess.run(z_2 ,{ input_x:data_set[i].
                   reshape(1,7) , y_true:csi_close[i+1]})
                y_predicted.append(ans[0][0])

        for j in range(sample_2):
                ans1 = sess.run(z_2 ,{ input_x:training_set[j].
                   reshape(1,7) , y_true:csi_close[sample_8 + j
                   +1]})
                y_predicted1.append(ans1[0][0])


mse = np.sum(np.square(csi_close[1:sample_8+1] - y_predicted))
   /len(y_predicted) ## 49926

np.savetxt('MLPpredict'+trainning+'.csv',y_predicted1 ,
   delimiter = ',')

print('MSE training',mse)
print('MAPE training', np.mean(abs((csi_close[1:sample_8+1] -
   y_predicted))/ csi_close[1:sample_8+1]))

print('MSE testing',np.mean(np.square(csi_close[sample_8+1: 1
   + sample_8+sample_2] - y_predicted1)))
```

```
print('MAPE testing', np.mean(abs((csi_close[sample_8+1: 1 +
    sample_8+sample_2] - y_predicted1))/ csi_close[sample_8+1: 1
     + sample_8+sample_2]))
```

B Python code for the LSTMNN

the code can be downloaded from https://github.com/AndyChenXia/ANNs-to-predict-the-CSI-300.git

```
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
import pandas as pd
import math
import datetime
import os



os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
#exponential moving average for 10
def eMovingAverage(x,n):
        mv10 = []
        multiplier = 2/(n+1)

        mv10.append(np.sum(x[0:n]) / n)

        for i in range(n,len(x)):
                mv10.append(((x[i] - mv10[i-(n)]) * multiplier
                    ) + mv10[i-(n)])
        return mv10

## simple moving average
def movingAverage(x,n):
        mv = []
        for i in range(n-1,len(x)):
                mv.append(np.sum(x[i-n-1:i])/n)
        return mv

## ADL
def adl_cal(high,low,close,volume):
        acumulator = np.zeros(len(low))
        multiplier1 = ((close[0] - low[0]) - (high[0] - close
           [0])) / (high[0] - low[0])
        acumulator[0] = multiplier1 * volume[0]

        for i in range(1,len(low)):
                flow_multiplier = ((close[i] - low[i]) - (high
                   [i] - close[i])) / (high[i] - low[i])
                flow_volume = flow_multiplier * volume[i]
                acumulator[i] = acumulator[i-1] + flow_volume
        return acumulator

## momentum ROC
def momentum(x):
        acumulator = []
        counter = 0
        for i in range(10,len(x)):
                acumulator.append(((x[i] - x[counter])/x[i]) *
                    100)
                counter = counter + 1
        return acumulator

## len check
def lenCheck(x,y):
        return (len(x) == len(y))

## stochastic oscillator including the same day --- 10 days
def stochasticOscillator(x):
        acumulator = []
        for i in range(9,len(x)):
                recent_close = x[i]
                high = np.max(x[i-9:i])
                low = np.min(x[i-9:i])
                acumulator.append(((recent_close - low) / (
                   high - low)) * 100)
        return acumulator

def profit(x):
        acumulator = []
        for i in range(len(x)-1):
                result = x[i] - x[i+1]
                if result > 0:
                        acumulator.append(1)
                else:
                        acumulator.append(0)
        return acumulator

## RSI
def rsiCalculation(x):
        loss = []
        gain = []
        rsi = []
        for i in range(1,11):
                result = x[i] - x[i-1]

                if result >= 0.0:
                        gain.append(result)
                else:
                        loss.append(abs(result))

        first_gains = np.sum(gain) / 10
        first_loss = np.sum(loss) / 10




        rs = first_gains/ first_loss
        rsi.append(100 - (100 / (1+rs)))

        average_gain = []
        average_loss = []
        average_gain.append(first_gains)
        average_loss.append(first_loss)

        for i in range(11,len(x)):

                result = x[i] - x[i-1]
                averageGain = 0
                averageLoss = 0

                if result >= 0.0:
                        averageGain = ((average_gain[i-11] *
                           9) + result) / 10
                        averageLoss = ((average_loss[i-11] *
                           9)) / 10

                        average_gain.append(averageGain)
                        average_loss.append(averageLoss)
                else:
                        averageGain = ((average_gain[i-11] *
                           9)) / 10
                        averageLoss = ((average_loss[i-11] *
                           9) + abs(result)) / 10

                        average_gain.append(averageGain)
                        average_loss.append(averageLoss)

                rs = averageGain/averageLoss
                rsi.append(100 - (100 / (1+rs)))

        return rsi

##data normalization
def normalization(x):
        mean = np.mean(x)
        deviation = np.std(x)
        norm_data = (x - mean) / deviation
        return mean, deviation , norm_data

def gaf_Image(x):
        data = []
        for i in range(len(x)):
                arccos = np.arccos(x[i])
                matrix = arccos.T + arccos
                data.append(np.cos(matrix))

        return np.array(data)

def data_to_image(x,days):
        conv_data = []
        for z in range(0,len(x)-days):
                maximum = np.amax(x[z:z+days])
                minimum = np.amin(x[z:z+days])
                numerator = ((x[z:z+days] - maximum) + (x[z:z+
                   days] - minimum))
                norm = numerator / (maximum - minimum)
                arccos = np.arccos(norm).reshape(1,days_taken)
                matrix = arccos.T + arccos
                conv_data.append(matrix)
        return np.reshape(np.array(conv_data),[-1,days,days])

def image_ordering(image):
        number = len(image[0])
        x = len(image[0][0])
        y = len(image[0][0][0])
        z = len(image)

        acc = np.zeros([number,x,y,z])

        for i in range(z):
                for j in range(number):
                        for q in range(x):
                                for g in range(y):
```

```
                                                    acc[j][q][g][i] =
                                                        image[i][j][q][g]
        return acc

##
def max_min(x):
        print(np.min(x),np.max(x))

def prediction_normalization(x):
        maximum = np.amax(x)
        minimum = np.amin(x)
        numerator = ((x - maximum) + (x - minimum))
        norm = numerator / (maximum - minimum)
        return norm, numerator, (maximum - minimum)

#getting CSI data
data = pd.read_csv("CSIData2.csv")

date = np.array(data['Name'])
csi_high = np.array(data['PRICE_HIGH'],dtype = float)
csi_low = np.array(data['PRICE_LOW'],dtype = float)
csi_close = np.array(data['PRICE_INDEX'],dtype = float)
csi_volume = np.array(data['VOLUME'],dtype = float)

logic = np.isnan(csi_volume)



#for removing nans
date = date[~logic]
csi_high = csi_high[~logic]
csi_low = csi_low[~logic]
csi_close = csi_close[~logic]
csi_volume = csi_volume[~logic]

dates = [datetime.datetime.strptime(date1, '%m/%d/%Y') for
    date1 in date]
```

```
#exchange data
data2 = np.genfromtxt("exchange.csv",delimiter = ",",
    skip_header=2)
exchange = data2[:,1]
exchange = exchange[~logic]

##features to use exponential moving average 10 days and 30
    days, momentum for 10 days, forex exchange rate too, ADL, 3
    days close price, Stochastic oscillator

# 10 days exponential moving average
emove10_close =  eMovingAverage(csi_close,10)

## 10 days momentum
momentum10_close = momentum(csi_close)

# 10 days stochastic oscillator && 3 period moving average of
    oscillator
oscillator10_close = stochasticOscillator(csi_close)
oscilaltor10movingAverage_close = movingAverage(
    oscillator10_close,3)
## 30 days exponential moving average
emove30_close = eMovingAverage(csi_close,30)

## RSI calculation 10 days
rsi_close = rsiCalculation(csi_close)



## Adjusting data length for training
adjustment = len(dates) - len(emove30_close)
csi_high = csi_high[adjustment:]
csi_low = csi_low[adjustment:]
csi_close = csi_close[adjustment:]
csi_volume = csi_volume[adjustment:]
dates = dates[adjustment:]
```

```
momentum10_close = momentum10_close[adjustment -10:-1]
emove10_close = emove10_close[adjustment -9:-1]
oscillator10_close = oscillator10_close[adjustment -9:-1]
oscilaltor10movingAverage_close =
    oscilaltor10movingAverage_close[adjustment -11:-1]
rsi_close = rsi_close[adjustment -10:-1]
emove30_close = emove30_close[0:-1]

## Accumulation distribution line ADL
adl = adl_cal(csi_high ,csi_low ,csi_close ,csi_volume)
adl = adl[0:-1]

sample = len(momentum10_close)
sample_8 = math.ceil(sample *.8)
sample_2 = math.floor(sample * .2)


norm_momentum10_close = normalization(momentum10_close[:
    sample_8])
norm_emove10_close = normalization(emove10_close[:sample_8])
norm_emove30_close = normalization(emove30_close[:sample_8])
norm_oscillator10_close = normalization(oscillator10_close[:
    sample_8])
norm_oscilaltor10movingAverage_close = normalization(
    oscilaltor10movingAverage_close[:sample_8])
norm_rsi_close = normalization(rsi_close[:sample_8])
norm_adl = normalization(adl[:sample_8])
norm_csi_close = normalization(csi_close[:sample_8])


data_set = np.vstack((norm_momentum10_close[2],
    norm_emove10_close[2],norm_emove30_close[2],
    norm_oscillator10_close[2],
    norm_oscilaltor10movingAverage_close[2],norm_rsi_close[2],
    norm_adl[2]))
```

```
data_set = data_set.T


norm_momentum10_close2 = (momentum10_close[sample_8:sample_8+
   sample_2] - norm_momentum10_close[0]) /
   norm_momentum10_close[1]
norm_emove10_close2 =  (emove10_close[sample_8:  sample_8+
   sample_2] - norm_emove10_close[0])  / norm_emove10_close[1]
norm_emove30_close2 =  (emove30_close[sample_8:  sample_8+
   sample_2] - norm_emove30_close[0])  / norm_emove30_close[1]
norm_oscillator10_close2 =  (oscillator10_close[sample_8:
   sample_8+sample_2] - norm_oscillator10_close[0])  /
   norm_oscillator10_close[1]
norm_oscilaltor10movingAverage_close2 =  (
   oscilaltor10movingAverage_close[sample_8:sample_8+sample_2]
   - norm_oscilaltor10movingAverage_close[0])  /
   norm_oscilaltor10movingAverage_close[1]
norm_rsi_close2 =  (rsi_close[sample_8:sample_8+sample_2] -
   norm_rsi_close[0])  / norm_rsi_close[1]
norm_adl2 = (adl[sample_8:sample_8+sample_2] - norm_adl[0])  /
    norm_adl[1]
norm_csi_close2 =  (csi_close[sample_8:sample_8+sample_2] -
   norm_csi_close[0]) / norm_csi_close[1]

training_set = np.vstack((norm_momentum10_close2,
   norm_emove10_close2,norm_emove30_close2,
   norm_oscillator10_close2,
   norm_oscilaltor10movingAverage_close2,norm_rsi_close2,
   norm_adl2))
training_set = training_set.T

element_size = 7
time_steps = 1
hidden_neurons = 8
batch_size = 32
learning_rate = 0.01
```

```
epochs = 1500
sequence_array = np.ones(batch_size) * time_steps

initial = tf.keras.initializers.he_normal()
input_x = tf.placeholder(shape = [None,time_steps ,element_size
   ],dtype = tf.float32)
y_true = tf.placeholder(shape = (None),dtype = tf.float32)
w1 = tf.Variable(initial([hidden_neurons ,1]))
b1 = tf.Variable(initial([1]))

lstm_cell = tf.contrib.rnn.LSTMCell(hidden_neurons ,initializer
    = initial ,forget_bias = 1.0)
outputs , states = tf.nn.dynamic_rnn(lstm_cell , input_x , dtype=
   tf.float32)

final_output = tf.matmul(states[1],w1) + b1
cost_function =  tf.reduce_mean(tf.square(y_true -
   final_output))
accuracy = tf.reduce_mean(tf.square(y_true - final_output))

optimization = tf.train.AdamOptimizer(learning_rate =
   learning_rate)
train = optimization.minimize(cost_function)



y_test = []
y_train = []

with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        for steps in range(epochs):
                for i in range(0,sample_8 , batch_size):

                        sess.run(train ,feed_dict = {input_x:
                           data_set[i: i + batch_size].reshape
                           (-1,time_steps ,7),y_true:csi_close[i
```

```
                    +1: i + batch_size + 1]})
        for i in range(sample_8):
                ans = sess.run(final_output ,feed_dict = {
                   input_x:data_set[i].reshape(-1,time_steps ,7)
                   ,y_true:csi_close[i + 1]})
                y_train.append(ans[0][0])
        for i in range(sample_2):
                ans1 = sess.run(final_output ,feed_dict = {
                   input_x:training_set[i].reshape(-1,
                   time_steps ,7),y_true:csi_close[sample_8 + i
                   +1]})
                y_test.append(ans1[0][0])

y_test= np.array(y_test)
np.savetxt('LSTMpredict.csv',y_test ,delimiter = ',')

mse = np.sum(np.square(csi_close[1:sample_8+1] - y_train))/len
   (y_train)
print('MSE training',mse)
print('MAPE training', np.mean(abs((csi_close[1:sample_8+1] -
   y_train))/ csi_close[1:sample_8+1]))

print('MSE testing',np.mean(np.square(csi_close[sample_8 + 1:
   sample_2 + 1 + sample_8 ] - y_test)))
print('MAPE testing', np.mean(abs((csi_close[sample_8 + 1:
   sample_2 + 1 + sample_8 ] - y_test))/ csi_close[sample_8 +
   1: 1 + sample_8 + sample_2]))
```

C Python code for the CNN

the code can be downloaded from https://github.com/AndyChenXia/ANNs-to-predict-the-CSI-300.git

```
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
import pandas as pd
import math
import datetime
import os



os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
#exponential moving average for 10
def eMovingAverage(x,n):
        mv10 = []
        multiplier = 2/(n+1)

        mv10.append(np.sum(x[0:n]) / n)

        for i in range(n,len(x)):
                mv10.append(((x[i] - mv10[i-(n)]) * multiplier
                    ) + mv10[i-(n)])
        return mv10

## simple moving average
def movingAverage(x,n):
        mv = []
        for i in range(n-1,len(x)):
                mv.append(np.sum(x[i-n-1:i])/n)
        return mv

## ADL
def adl_cal(high,low,close,volume):
        acumulator = np.zeros(len(low))
        multiplier1 = ((close[0] - low[0]) - (high[0] - close
           [0])) / (high[0] - low[0])
        acumulator[0] = multiplier1 * volume[0]

        for i in range(1,len(low)):
                flow_multiplier = ((close[i] - low[i]) - (high
                   [i] - close[i])) / (high[i] - low[i])
                flow_volume = flow_multiplier * volume[i]
                acumulator[i] = acumulator[i-1] + flow_volume
        return  acumulator

## momentum ROC
def momentum(x):
        acumulator = []
        counter = 0
        for i in range(10,len(x)):
                acumulator.append(((x[i] - x[counter])/x[i]) *
                    100)
                counter = counter + 1
        return  acumulator

## len check
def lenCheck(x,y):
        return (len(x) == len(y))

## stochastic oscillator including the same day --- 10 days
def stochasticOscillator(x):
        acumulator = []
        for i in range(9,len(x)):
                recent_close = x[i]
                high = np.max(x[i-9:i])
                low = np.min(x[i-9:i])
                acumulator.append(((recent_close - low) / (
                   high - low)) * 100)
        return acumulator

def profit(x):
        acumulator = []
        for i in range(len(x)-1):
                result = x[i] - x[i+1]
                if result > 0:
                        acumulator.append(1)
                else:
                        acumulator.append(0)
        return acumulator

## RSI
def rsiCalculation(x):
        loss = []
        gain = []
        rsi = []
        for i in range(1,11):
                result = x[i] - x[i-1]

                if result >= 0.0:
                        gain.append(result)
                else:
                        loss.append(abs(result))

        first_gains = np.sum(gain) / 10
        first_loss = np.sum(loss) / 10



        rs = first_gains/ first_loss
        rsi.append(100 - (100 / (1+rs)))

        average_gain = []
        average_loss = []
        average_gain.append(first_gains)
        average_loss.append(first_loss)

        for i in range(11,len(x)):

                result = x[i] - x[i-1]
                averageGain = 0
                averageLoss = 0

                if result >= 0.0:
                        averageGain = ((average_gain[i-11] *
                           9) + result) / 10
                        averageLoss = ((average_loss[i-11] *
                           9)) / 10

                        average_gain.append(averageGain)
                        average_loss.append(averageLoss)
                else:
                        averageGain = ((average_gain[i-11] *
                           9)) / 10
                        averageLoss = ((average_loss[i-11] *
                           9) + abs(result)) / 10

                        average_gain.append(averageGain)
                        average_loss.append(averageLoss)

                rs = averageGain/averageLoss
                rsi.append(100 - (100 / (1+rs)))

        return rsi

##data normalization
def normalization(x):
        mean = np.mean(x)
        deviation = np.std(x)
        norm_data = (x - mean) / deviation
        return mean,deviation ,norm_data

def gaf_Image(x):
        data = []
        for i in range(len(x)):
                arccos = np.arccos(x[i])
                matrix = arccos.T + arccos
                data.append(np.cos(matrix))

        return np.array(data)

def data_to_image(x,days):
        conv_data = []
        for z in range(0,len(x)-days):
                maximum = np.amax(x[z:z+days])
                minimum = np.amin(x[z:z+days])
                numerator = ((x[z:z+days] - maximum) + (x[z:z+
                   days] - minimum))
                norm = numerator / (maximum - minimum)
                arccos = np.arccos(norm).reshape(1,days_taken)
                matrix = arccos.T + arccos
                conv_data.append(matrix)
        return np.reshape(np.array(conv_data),[-1,days,days])
##
def max_min(x):
        print(np.min(x),np.max(x))

def prediction_normalization(x):
        maximum = np.amax(x)
        minimum = np.amin(x)
        numerator = ((x - maximum) + (x - minimum))
        norm = numerator / (maximum - minimum)
        return norm, numerator, (maximum - minimum)

#getting CSI data
data = pd.read_csv("CSIData2.csv")
```

```
date = np.array(data['Name'])
csi_high = np.array(data['PRICE_HIGH'],dtype = float)
csi_low = np.array(data['PRICE_LOW'],dtype = float)
csi_close = np.array(data['PRICE_INDEX'],dtype = float)
csi_volume = np.array(data['VOLUME'],dtype = float)

logic = np.isnan(csi_volume)



#for removing nans
date = date[~logic]
csi_high = csi_high[~logic]
csi_low = csi_low[~logic]
csi_close = csi_close[~logic]
csi_volume = csi_volume[~logic]

dates = [datetime.datetime.strptime(date1, '%m/%d/%Y') for
    date1 in date]

#exchange data
data2 = np.genfromtxt("exchange.csv",delimiter = ",",
    skip_header=2)
exchange = data2[:,1]
exchange = exchange[~logic]

##features to use exponential moving average 10 days and 30
    days, momentum for 10 days, forex exchange rate too, ADL, 3
    days close price, Stochastic oscillator

# 10 days exponential moving average
emove10_close =  eMovingAverage(csi_close,10)

## 10 days momentum
momentum10_close = momentum(csi_close)
```

```
# 10 days stochastic oscillator && 3 period moving average of
    oscillator
oscillator10_close = stochasticOscillator(csi_close)
oscilaltor10movingAverage_close = movingAverage(
    oscillator10_close ,3)
## 30 days exponential moving average
emove30_close = eMovingAverage(csi_close ,30)

## RSI calculation 10 days
rsi_close = rsiCalculation(csi_close)


## Adjusting data length for training
adjustment = len(dates) - len(emove30_close)
csi_high = csi_high[adjustment:]
csi_low = csi_low[adjustment:]
csi_close = csi_close[adjustment:]
csi_volume = csi_volume[adjustment:]
dates = dates[adjustment:]

momentum10_close = momentum10_close[adjustment -10:-1]
emove10_close = emove10_close[adjustment -9:-1]
oscillator10_close = oscillator10_close[adjustment -9:-1]
oscilaltor10movingAverage_close =
    oscilaltor10movingAverage_close[adjustment -11:-1]
rsi_close = rsi_close[adjustment -10:-1]
emove30_close = emove30_close[0: -1]

## Accumulation distribution line ADL
adl = adl_cal(csi_high ,csi_low ,csi_close ,csi_volume)
adl = adl[0: -1]


## data normalization
sample = len(momentum10_close)
sample_7 = math.ceil(sample *.8)
```

```
sample_15 = math.floor(sample * .2)



days_taken = 64*3



cnn_emove10_close = data_to_image(emove10_close[0:sample_7],
    days_taken)
cnn_emove30_close = data_to_image(emove30_close[0:sample_7],
    days_taken)
cnn_adl =data_to_image(adl[0:sample_7],days_taken)

sample2 = len(cnn_emove10_close)
input_image = np.stack([cnn_emove10_close,cnn_emove30_close,
    cnn_adl])
input_image = np.transpose(input_image,[1,2,3,0])

print(input_image.shape)



cnn_emove10_close2 = data_to_image(emove10_close[sample_7 -
    days_taken:sample_15 + sample_7],days_taken)
cnn_emove30_close2 = data_to_image(emove30_close[sample_7 -
    days_taken:sample_15 + sample_7],days_taken)
cnn_adl2 = data_to_image(adl[sample_7 - days_taken:sample_15 +
     sample_7],days_taken)



sample2_testing = len(cnn_emove10_close2)

testing_image = np.stack([cnn_emove10_close2,
    cnn_emove30_close2,cnn_adl2])
testing_image = np.transpose(testing_image,[1,2,3,0])

def weight_variable(shape):
        initial = tf.keras.initializers.he_normal()
        return tf.Variable(initial(shape))

def bias_variable(shape):
        initial = tf.keras.initializers.he_normal()
        return tf.Variable(initial(shape))

def conv2d(x, W,mode):
        return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1],
           padding=mode)

def conv1d(x, W,mode):
        return tf.nn.conv1d(x, W, stride=1, padding=mode)

def pool(x):
        return tf.nn.max_pool(x, ksize=[1, 3, 3, 1], strides
           =[1, 3, 3, 1], padding='VALID')

def conv_layer(input,W,b,mode = 'SAME'):
        return tf.nn.leaky_relu(conv2d(input, W,mode) + b)

def conv_layer1d(input,W,b,mode = 'SAME'):
        return tf.nn.leaky_relu(conv1d(input, W,mode) + b)

def full_layer(input1,inputneurons, size):
       W = weight_variable([inputneurons, size])
        b = bias_variable([size])
        return tf.matmul(input1, W) + b

def weights_calc(shape):
        ans =  tf.keras.initializers.he_normal()
        ans = tf.Variable(ans(shape))
        d = tf.constant([],dtype = tf.float32)
        for i in range(shape[0]):
                for x in range(shape[1]):
                        for y in range(shape[2]):
                              for z in range(shape[3]):
                                    if z == y:
                                          m = tf.
                                             Variable([
                                             ans[i][x][y
                                             ][z]],dtype
                                             = tf.float32
                                             )
                                          d = tf.concat
                                             ([d,m],axis
                                             = 0)



                                    else:
                                          n = tf.
                                             constant
                                             ([0],dtype =
                                              tf.float32)

                                          d = tf.concat
                                             ([d,n],axis
                                             = 0)
            return tf.reshape(d,(shape))


learning_rate = 0.01
epochs = 50

x_input = tf.placeholder(tf.float32 ,shape = (None,days_taken ,
   days_taken ,3))
y_true = tf.placeholder(tf.float32 ,shape = (None))

with tf.device('/cpu:0'):
      initial = tf.keras.initializers.he_normal()
```

```
        shape_w1 = [3,3,3,3]
        shape_w2 = [3,3,3,3]
        shape_w3 = [3,3,4,8]
        shape_w4 = [3,3,14,7]

        w1 = tf.Variable(initial(shape_w1))
        b1 = tf.Variable(initial([shape_w1[3]]))
        conv1 = conv_layer(x_input,w1,b1)

        w2 = tf.Variable(initial(shape_w2))
        b2 = tf.Variable(initial([shape_w2[3]]))
        conv2 = conv_layer(conv1,w2,b2)
        conv2_pool = pool(conv2)
        print(conv2_pool)

        conv3_flat = tf.reshape(conv2_pool, [-1, 64*64*3])

        a_2 = tf.nn.leaky_relu(full_layer(conv3_flat
           ,64*64*3,64*64))
        a_3 = tf.nn.leaky_relu(full_layer(a_2,64*64,10))
        z_2 = full_layer(a_3,10,1)
        cost_loss = tf.reduce_mean(tf.square(y_true - z_2))

        optimizer = tf.train.AdamOptimizer(learning_rate =
           learning_rate)
        train = optimizer.minimize(cost_loss)

        accuracy = tf.reduce_mean(tf.square(y_true - z_2))

batch_size = 32

y_predicted = []
y_predicted_testing = []

init = tf.global_variables_initializer()
```

```
with tf.Session() as sess:
        sess.run(init)
        for steps in range(epochs):
                for i in range(0,sample2,batch_size):
                        sess.run(train,feed_dict = {x_input:
                           input_image[i:i+batch_size],y_true:
                           csi_close[i+days_taken+1:i+
                           batch_size+days_taken+1]})
        for i in range(sample2):
                ans = sess.run(z_2,feed_dict = {x_input:
                   input_image[i].reshape([-1,days_taken,
                   days_taken,3])})
                y_predicted.append(ans[0][0])
        for i in range(sample2_testing):
                ans2 = sess.run(z_2,feed_dict = {x_input:
                   testing_image[i].reshape([-1,days_taken,
                   days_taken,3])})
                y_predicted_testing.append(ans2[0][0])

y_predicted_testing = np.array(y_predicted_testing)
np.savetxt('CNNpredict.csv',y_predicted_testing,delimiter = ',
   ')

y_predicted = np.array(y_predicted)
mse = np.sum(np.square(csi_close[days_taken+1:sample2+
   days_taken +1] - y_predicted))/len(y_predicted) ## 49926#
print('MSE training',mse)
print('MAPE training', np.mean(abs((csi_close[days_taken +1:
   sample2+days_taken+1] - y_predicted))/ csi_close[days_taken
   +1:sample2+days_taken+1]))

mse1 = np.sum(np.square(csi_close[sample2+days_taken + 1:
   sample2+days_taken+ sample2_testing +1] -
   y_predicted_testing))/len(y_predicted_testing)
print('MSE testing',mse1)
print('MAPE testing', np.mean(abs((csi_close[sample2 +
   days_taken +1:sample2+days_taken+ sample2_testing +1] -
   y_predicted_testing))/ csi_close[sample2+days_taken +1:
   sample2+days_taken+ sample2_testing +1]))
```