INSTITUTO
UNIVERSITÁRIO
DE LISBOA

Financial Time Series Forecasting using Artificial Neural Networks

Luís Gil Miguens Cardoso

Master in Economics

Supervisor:
PhD in Management José Joaquim Dias Curto,
Associate Professor,
ISCTE — Instituto Universitário de Lisboa

November, 2020

BUSINESS
SCHOOL

Department of Economics

Financial Time Series Forecasting using Artificial Neural Networks

Luís Gil Miguens Cardoso

Master in Economics

Supervisor:
PhD in Management José Joaquim Dias Curto,
Associate Professor,
ISCTE — Instituto Universitário de Lisboa

November, 2020

I dedicate this work to my family, my parents in particular. It is through their constant support, financial and otherwise, that I have been able to get this far. I'm sorry I'm not always fair to you. I hope you're proud.

Acknowledgment

I would like to thank my professors at ISCTE, who have patiently and passionately guided me through my academic pursuits: Prof. Henrique Monteiro, who has generously provided his intellectual and personal support to me and to his other students, often going beyond what one would expect from a university professor. His kind words have motivated me to pursue further academic learning and to enjoy pedagogy. Prof. Filipe Ramos, who gave me confidence and pushed me to command more advanced mathematics. His kindness and humility make him an exemplary professor. Were it not for our discussions of neural networks, I may not have pursued this study. Prof. Maria João Cortinhal, who would always indulge my desire to discuss probability problems with her for hours after class. I still have an ongoing project she has helped kick-start, for which I'm very grateful. Prof. José Dias Curto, who has an infectious positive energy that makes me feel more motivated and confident to work on my ideas. I've been very fortunate to have him as my supervisor. I hold these people, as well as all of my professors who I have not mentioned directly, very close to my heart, truthfully. All my professors at ISCTE, and even prior to university, have generously spent many hours outside of class indulging my curiosity and thirst for knowledge. The rich discussions I've had with them have shaped me to become a better student and a better person. I can only hope to make them proud.

Resumo

Este estudo constrói modelos de redes neuronais artificiais com o uso de *stacked autoencoders* (SAE) para extrair variáveis latentes sem ruído e *long short-term memory* (LSTM) para gerar previsões para o *next-day adjusted closing price* do S&P500. Dados para sete índices de ações diferentes, indicadores técnicos e variáveis macroeconómicas são usados para treinar três modelos diferentes: um 'modelo de preço' que prevê o preço do dia seguinte, um 'modelo de mudança' que prevê a mudança relativa no preço e um ' modelo binário' que prevê a probabilidade de um aumento de preço. Os modelos foram avaliados com base na sua precisão preditiva e lucratividade. Os resultados mostram que os modelos falham em generalizar bem ou caem num mínimo vicioso que se aproxima de um *naive predictor*. Além disso, os modelos parecem particularmente fracos a prever quebras na série, provavelmente devido à sua infrequência. Isto pode fornecer evidências que apoiam a hipótese do mercado eficiente.

Abstract

This study builds an artificial neural network framework with the use of stacked autoencoders (SAE) to extract deep denoised features, and long short-term memory (LSTM) to generate forecasts for the next-day adjusted closing price of S&P500. Data for seven different stock indices, technical indicators, and macroeconomic variables is used to train three different models: a 'price model' which predicts the next-day price, a 'change model' which predicts the relative change in price, and a 'binary model' which predicts the probability of a price increase. The models were judged based on predictive accuracy and profitability. Results show the models either fail to generalize well or fall prey to a vicious minimum approximating a naive predictor. Furthermore, the models appear particularly poor at predicting breaks in the series, likely due to their infrequency. This might provide evidence supporting the efficient market hypothesis.

Contents

CHAPTER 1

Introduction

The use of artificial neural networks has become popular for recognizing, identifying, and predicting complex patterns in data, be it in image recognition, speech recognition, text prediction, or others. Their power comes from their flexibility and ability to describe highly complex non-linear relationships, as per the universal approximation theorem, which, loosely, states that for any function of arbitrary complexity there is a neural network that can approximate it with arbitrary precision. Because of this, there has been interest in using them for forecasting financial time series, known for their high volatility. The idea is that while linear regression models fail to reproduce non-linear behavior in the data, and the efficient market hypothesis notwithstanding, a neural network model may be able to capture those patterns successfully.

With this in mind, the goal of this study is to generate the next-day forecasts for the adjusted closing price of S&P500 by crafting an artifical neural network framework consisting of a long short-term memory stacked autoencoder for denoising and dimensionality reduction, and a long short-term memory network for prediction. Three different models are trained. A 'price model' which is trained to generate the next-day price, a 'change model' which is trained to generate the relative change in price, and a 'binary model' which is trained to generate the probability that the price will increase. The models are evaluated on their predictive accuracy, and profitability when trading following simple strategies. The models were tuned with primacy for profit.

This work strives to endow the reader with an intuitive understanding of the methods used. Furthermore, all the Python code and data used are provided for full transparency, reproducibility, and improvement for future research: `https://github.com/LumpyJumbo/NeuralNet`.

Chapter 2 goes over a brief review of past literature on machine learning models for predicting financial time series, as well as other methods involved in the process. Chapter 3 provides the theoretical background used as the foundation of the methods used in this study. Chapter 4 details the framework used to generate the forecasts of the next-day adjusted closing price for S&P500. Chapter 5 provides the results, both in terms of predictive accuracy and profitability, of the models presented in the prior chapter. Finally, closing remarks are presented in Chapter 6.

CHAPTER 2

Literature Review

This chapter goes over some of the previous work done in financial time series forecasting using artificial neural networks, focusing on the methods used. This includes not only the machine learning algorithms for forecasting, but also the algorithms used in preprocessing data. Some key literature which provides insight into the theory and application of these methods is also discussed in brief detail. Further detail on the theory and methodology relevant to this work is provided in the subsequent theoretical framework chapter.

The use of artificial neural networks for stock market prediction can be traced at least as far back as 1988. Halbert White, one of the most influential econometricians of all time, was intrigued by the success of neural networks across many fields, and was keen on testing their efficacy in predicting stock markets. White (1988) attempts to find evidence against the efficient market hypothesis by using a basic multilayer perceptron with no recursive features to obtain predictions for IBM daily stock returns. In his introduction, he makes a case for the efficient market hypothesis while raising a very interesting point: It makes sense for there to be no regularities in the historical data of stocks left to be exploited, as opportunities for arbitrage are quickly seized; On the other hand, arguments in favor of human bounded rationality might lead one to suspect that there are regularities that have just not been able to be captured and exploited yet. If one succeeds in developing a technology which could tap into these opportunities, the owner of such technology would reap profits so long as it is not public domain, at which point they would be back where they started. This argument the author makes suggests a logical inevitability of the efficient market hypothesis, but that nonetheless may be bounded by the publicly available technology. Further research into developing sophisticated methods for forecasting may progressively reveal exploitable regularities in the data.

Autoregressive models have largely shown no contradiction with the efficient market hypothesis, but as White states, this is not proof of its universal validity. If anything, it tells us there is no simple linear relationship between today and the past. Interest in the application of artificial neural networks is thus made clear, given the adequacy of these models in capturing complex non-linear behavior that eludes a predetermined functional form.

Most interesting are his concluding remarks. Although the incredibly simple neural network failed to adequately perform, White points out the potential for future research

in this field, including the use of more variables, such as leading indicators and macroeconomic data, and more sophisticated architectures with recursive features, which is very on point with modern research on this field.

The recent work of Bao, Yue, & Rao (2017) introduces a novel model for preprocessing the data and obtaining forecasts for the next-day value of stock indices. They first use a wavelet transform (WT) to denoise the time series data, followed by stacked autoencoders (SAE) which capture the deep features of the data, and afterwards the processed data is used for training a long-short term memory (LSTM) architecture for forecasting. They have dubbed it WSAEs-LSTM. This model does not differ from its contemporary models in terms of the fundamental architecture used in forecasting. Where it innovates is in how it uses stacked autoencoders to extract deep features in the data before forecasting.

The data used are stock indices from different key markets around the world, the purpose for which is to serve as evidence of the robustness of the model to different levels of development in markets. They also use macro data such as the interest rate, along with a wide selection of indicators, all of which aid in introducing more relevant information for the model to work with. Besides their own designed model, they also train a group of competitor architectures in order to discern if their introduction of stacked autoencoders to the preprocessing of the data provides any significant benefits for forecasting. Their results provide strong evidence that preprocessing the data with SAE before forecasting can have large effects in results, as the models which lacked SAE performed the poorest. LSTM are also shown to perform better than standard RNN.

Other recent works include Lachiheb, & Gouider (2018). They use input dimensionality reduction for better prediction in a non-recursive neural network. Another point of interest is the high frequency of the data, which is every 5 minutes. Hsieh, Hsiao, & Yeh (2011) provide an interesting approach, combining wavelets for denoising and the standard RNN model with the artificial bee colony algorithm (ABC). Their proposed model outperforms its competitors, most of which are not neural network based. It would be interesting to have a comparison between this model and those of the likes shown in Bao, Yue, & Rao (2017).

It has been brought up how preprocessing the data can have a significant effect on the effectiveness of the neural networks. As such some work pertaining to preprocessing and denoising is discussed in the following paragraphs.

The Fourier transform provides a way to decompose a signal into its pure sine wave frequency components. One might arbitrarily determine that some frequencies pertain

to noise, and simply remove them before applying the inverse of the transform to reconstruct the now noiseless signal. This method for filtering is described in Estrella (2007) in application to business cycle extraction, and is less suited for high-frequency data with less regular behavior. Its biggest limitation is that the Fourier transform assumes that the pure frequencies that make up the signal are everlasting in time.

The work of Ramsey (1999) provides a better method. He summarizes 'the econometrician's plight' very eloquently in his introduction, highlighting the reasons as to why it is so challenging for economists to build empirical models, especially for financial data. The article reviews the possible applications of wavelets to economic and financial data, including forecasting, and potential research opportunities. Wavelets, unlike sine waves, are finite in time. This analysis allows one to admit spontaneous localized shocks. Through time-scale decomposition, one is able to decompose one's original signal into arbitrarily many wavelets of multiple arbitrary scales. A signal is, then, a weighted sum over all of its composing wavelets. This allows one to filter out noise by setting to 0 the coefficients of the wavelets that do not pass a defined threshold, and then reconstructing the signal with what is left. This is called shrinkage, and a very detailed explanation for it can be found in Donoho, & Johnstone (1995).

Finally, mention and praise to Goodfellow, Bengio, & Courville (2016) for the free availability of their Deep Learning book, Olah (2017) for their comprehensive blog on artificial neural networks, and Chollet et al (2015) for the Keras library for machine learning in Python, which made this research possible.

CHAPTER 3

Theoretical Framework

This chapter provides an overview of the artifical neural networks theory and methodology which provide the backbone for this work. Details on the actual models used and their construction can be found in the methodology chapter.

3.1. Artificial Neural Networks

Artifical neural networks are algorithms based on an intuitive understanding of how brains learn to classify and relate information. They are comprised of nodes, called neurons, connected to each other in a hierarchical structure. The connections between neurons can be likened to synapses in biological brains. Successful application has been found in many problems, such as image classification, speech recognition, medical diagnosis, and many others. They excel at tasks that a human brain would generally be able to perform after learning, but that are too complex to model accurately with more simple classes of model, such as linear regression.

Figure 1 is an illustration of a multilayer perceptron, one of the most basic architectures. Each neuron in a layer is connected to every neuron of the previous layer. This is to say that the value of a neuron, for a given input, is a function of the value of all previous neurons. Hence why these are called feedforward networks, as the values of the input are fed forward through the different strata of the model, generating latent values, until the output is generated in the final layer.

Generally, the value of a neuron is given by:

$$h_j = f(b_j + \sum_{i=1}^{n}(x_i * w_i)) \tag{3.1}$$

Where x_i are the previous layer's neurons, w_i are the respective weights associated to them (i.e. the strength of the connection between h_j and x_i), b_j is a bias associated to h_j, and f is called the activation function.

There are many possible activation functions, depending on the application. Some common examples are the sigmoid (σ), hyperbolic tangent ($tanh$), and rectified linear unit (ReLU) functions. It is the use of an activation function that provides the model with the ability to establish non-linear relationships, as otherwise the output would just

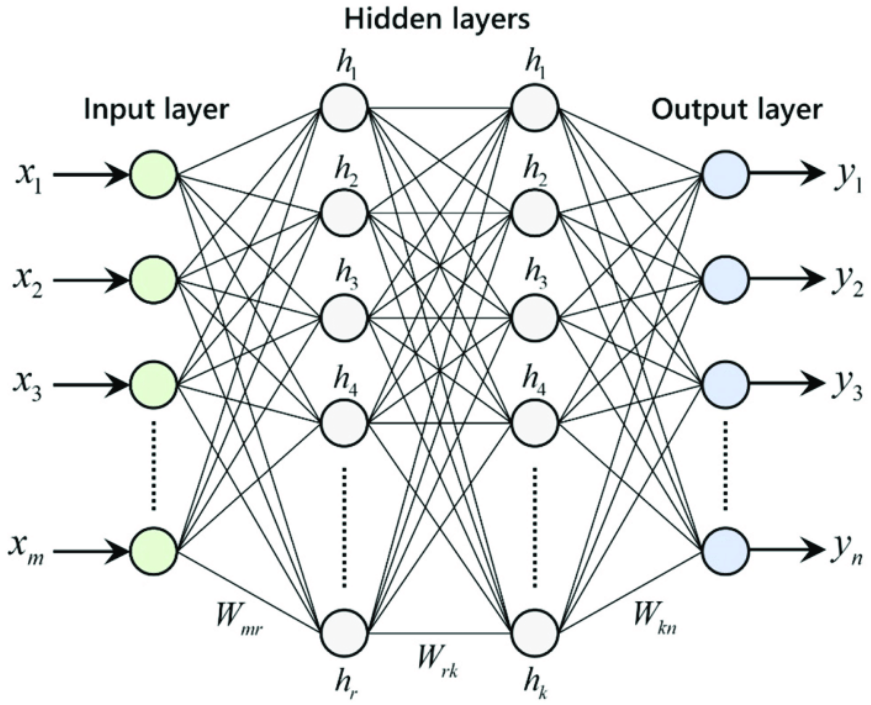


FIGURE 1. General structure of a multilayer perceptron with two hidden layers.

In Fernández-Cabán, Masters, and Phillips (2018)

be a composite of linear functions. For the case of the sigmoid activation there is, again, an analogy to the biological brain, as one can understand the neuron to be more active and have fired (sent a signal to the next layer) the closer it is to 1, and vice versa for 0.

The method for setting the correct (or rather, preferred) values for the weights and biases in a neural network is similar to that of regression models. A sample of inputs and their respective pre-attributed outputs (called labels) is required. The model's output for those inputs is then compared to their labels to create some loss function, such as square error, absolute error, or any other function. Using the ordinary least squares method is infeasible or at the very least impractical for neural networks, as the parameters are usually in the order of 10,000s or 100,000s, so instead the usually applied method is gradient descent. The parameters of the model are firstly randomly initialized and then the gradient over all its parameters is generated. From that point, steps of a given size (given by the learning rate) are taken in the opposite direction of the gradient, so as to minimize the loss function. Note that this gradient is computed for the average of the loss function of a given subset of inputs from the data. This is usually a random subset of a predetermined size and it is referred to as a batch. In other words, the data set is cut up randomly into subsets (batches) of equal and predetermined size, and for each batch, a step in the opposite direction of the gradient for the average loss of that batch

is taken. A full loop through the data using this process is called an epoch. The number of epochs, the batch size, and the learning rate are all hyperparameters, among others, that will condition the training and, therefore, the fit of the model. The overall process is referred to as training the model. There is no guarantee that a global minimum is found, so other optimizers have been created based on gradient descent, such as stochastic gradient descent (SGD) and Adam (adaptive moment estimation over SGD) which attempt to prevent the training process from locking itself into an undesirable local minimum.

One thing to note is that, unlike regression models, where the interest usually lies in gaining insight over the specifics of the relationships between variables, artificial neural networks are essentially a black box model. That is to say that it is difficult, if not impossible, to make sense of what is happening inside the model between the input and output ends. The concern is, then, whether they perform the task they were trained for adequately or not.

When training and using a neural network, it is important that the data is scaled appropriately. If the scale of the data varies along the sample, the optimization might favor parts of the sample of larger scale, as they might contribute more to the loss function. The greater sensitivity to scale might also cause the model to generalize poorly. A large scale for data can also result in greater weights which make the learning process unstable and also lead to poor generalization, as the weights respond more intensely to changes in the scale of input. To avoid these issues, one can apply some linear transformation to the data, which is easily inversible at the output end for application, e.g. normalization or standardization. This way the scale can remain consistent and small along the sample and across different variables, without affecting the desired ouput of the model.

A neural network can have any amount of hidden layers of any size. The amount of hidden layers is referred to as the depth of the model, and it is generally understood that a greater depth results in a greater ability to extract more complex features in the data. The greater the depth, and the greater the sizes of the hidden layers, the more capacity the model has to fit the training data (more parameters). This is why one of the challenges when training neural networks is to choose their shape such that they do not overfit, while also being able to capture the necessary features to perform their task. The amount of hidden units, and the depth, along with the epochs, batch size, learning rate, and several possible others, all comprise the hyperparameters of a neural network model.

To be able to determine which set of hyperparameters is most suited, a neural network model should be validated on a separate sample, i.e., its fit is quantified for values it was not trained for in order to determine how well it generalizes. Hyperparameters set the overall structure of neural networks, as well as how they are trained, so each configuration

of hyperparameters will yield new sets of parameters when trained. Much like parameters are optimized through gradient descent, hyperparameters ought also to be optimized, to minimize loss in the validation sample. The two optimization problems differ in complexity, as hyperparameter optimization (also referred to as tuning) implies parameter optimization.

In order to finally evaluate a neural network model's ability after it has been through training and validation, its fit should be tested on another separate sample.

3.2. Recurrent Neural Networks

In the previous illustration, the input space takes only one dimension of input (henceforth referred to as feature dimension), corresponding to the different variables, or features, of the input object, i.e. a cross-section of data. To make use of a second, ordered dimension (time) there is another class of model called recurrent neural networks (RNN). The input for these models is two dimensional, where unit width slices along the feature dimension (orthogonal to time) are fed one after another, each altering the latent state of the model in that order.

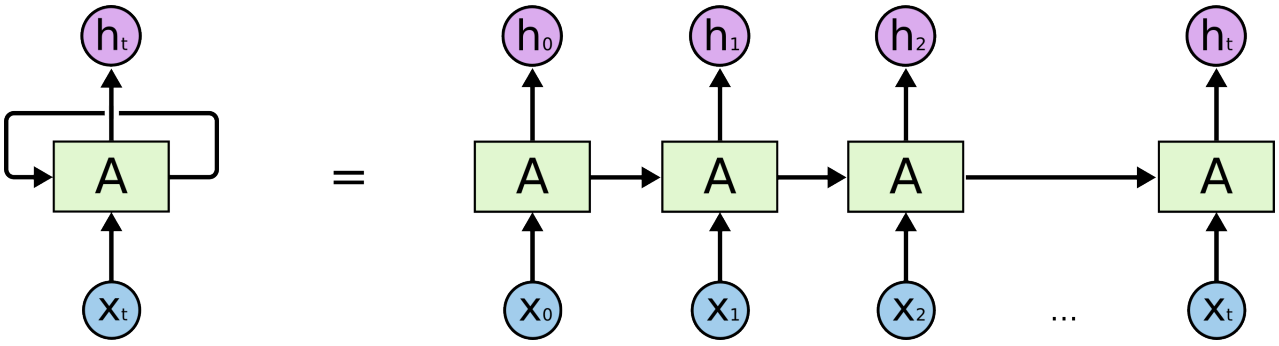


FIGURE 2. An unrolled recurrent neural network.

In Olah (2015)

Figure 2 depicts an RNN. x_i is a vector of feature dimension at time step i, A is some neural network, and h_i is a vector called the hidden state at time step i, which is the output of A. Note that when referring to the output of A, one is not referring to the output of some final model. Also note that the inputs for A at time step i are both x_i and h_{i-1}, when applicable. The output of a model itself which uses RNN will depend on the application. For instance, the concatenation of the hidden states could itself be the output of the model, or said concatenation could be used as input for a final output layer. Usually, for forecasting purposes, only the last hidden state (which contains processed information about all of the input) is used as input to predict the value of a variable in $t + 1$. Figure 3 is a summary of different ways to model output for RNN.

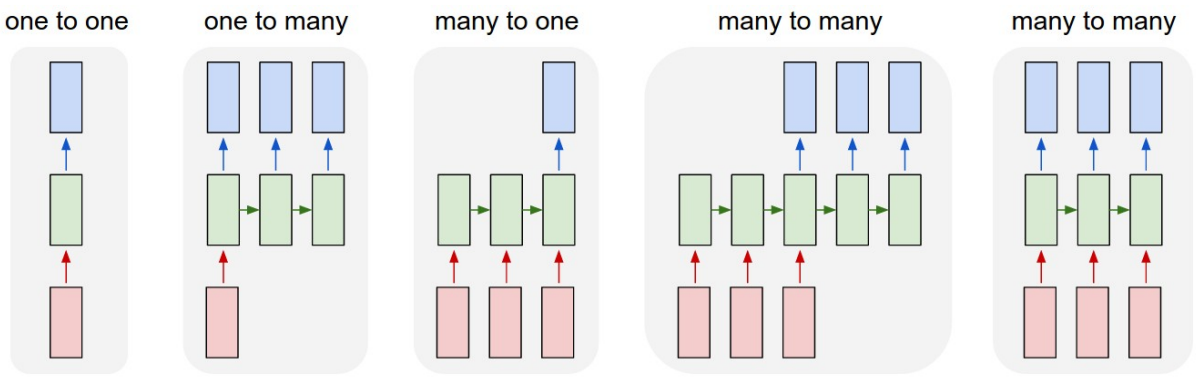


FIGURE 3. Input-to-output shape of standard ANN (left); Four different input-to-output shapes of RNN (right).

In Feng (2020)

Such RNN suffer from the issue of possibly having an exploding or vanishing gradient when performing gradient descent. Through the chain rule of differentiation, because there are several time steps, the partial derivatives for some parameters will contain products of long chains of repeating partial derivatives. In this way, through time, the gradient can grow or shrink exponentially, at least in certain directions (Brownlee, 2017). This makes the learning process unstable and ineffective. Figure 4 depicts one example of a basic RNN, with one hidden layer with *tanh* activation function.

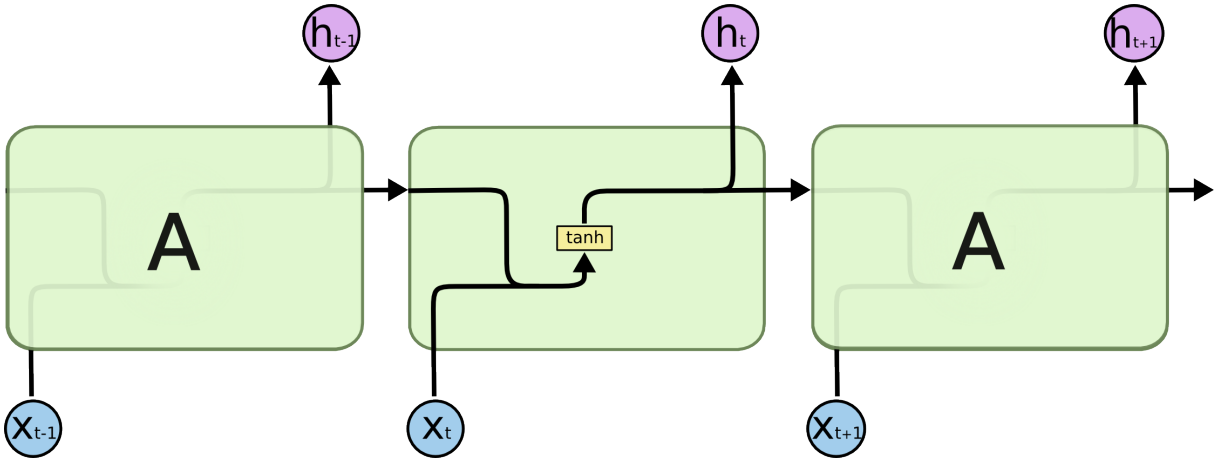


FIGURE 4. A repeating modulel of an RNN containing only a single layer.

In Olah (2015)

Some ways of alleviating or resolving this issue include the use of gradient clipping or weight regularization. Another way, which boasts other benefits, is to use a subclass of RNN called long short-term memory (LSTM). Figure 5 depicts an LSTM.

The design of the LSTM allows it to have selective memory. Figure 6 shows the cell state component of the LSTM, which serves as its memory. Information can be added or removed to and from the cell state at each time step. The changes to the cell state are

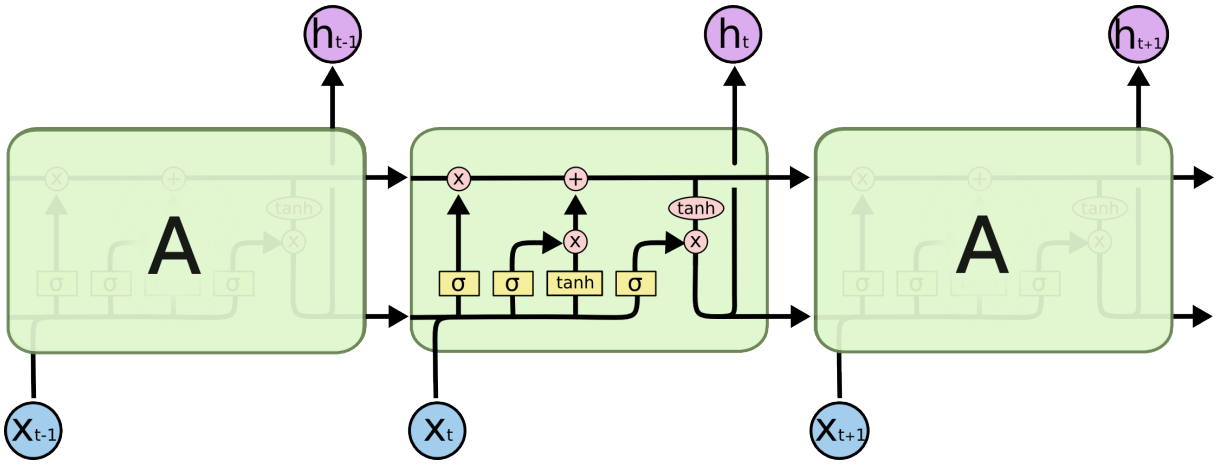


FIGURE 5. The repeating module of an LSTM.

In Olah (2015)

controlled by gates. The forget gate decides which values of the previous cell state should be remembered, and which should be forgotten (Figure 7). The input gate generates a new, candidate state for the cell state, and decides which values of the candidate state should be added as new information to the cell state (Figure 8). Figure 9 shows how the new cell state is generated based on these two gates. Finally, the ouput gate generates the output based on the previous output, and the current input and cell state (Figure 10).

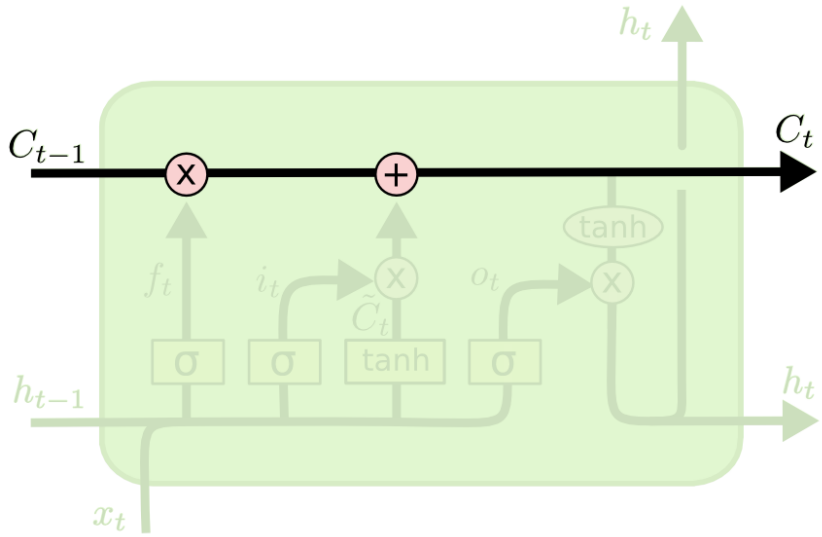
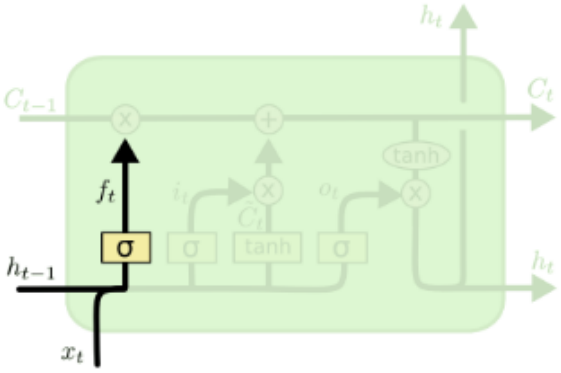


FIGURE 6. The cell state of an LSTM.

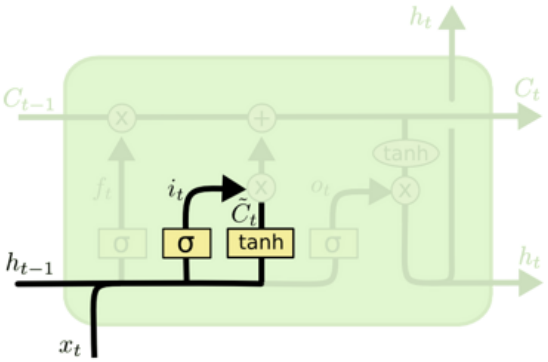
In Olah (2015)



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

FIGURE 7. The forget gate of an LSTM. f_t is a vector of values between 0 and 1. Element-wise multiplication is performed between C_{t-1} and f_t, such that the values of the previous cell state are remembered with strength given by f_t.

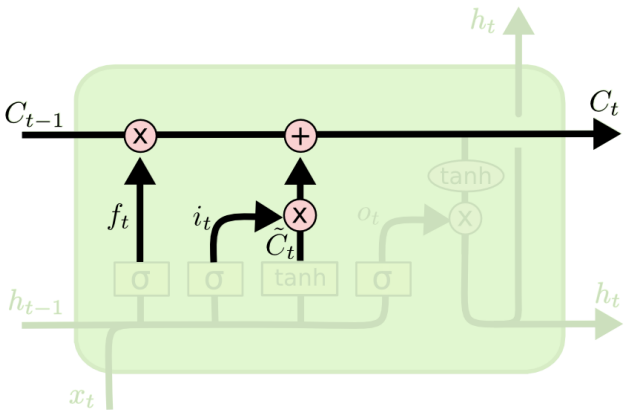
In Olah (2015)



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

FIGURE 8. The input gate of an LSTM. $\tilde{C}_t$ is the candidate state, with candidate information to be added to the cell state. i_t is a vector of values between 0 and 1, deciding how much of $\tilde{C}_t$ should be added to the cell state.

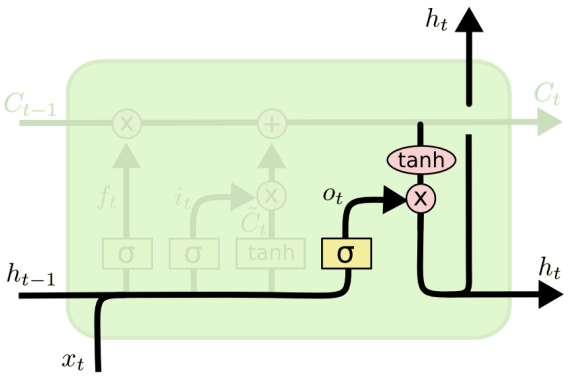
In Olah (2015)



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

FIGURE 9. The cell state is updated by the forget and input gates.

In Olah (2015)



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

FIGURE 10. The output gate of an LSTM. o_t is a vector of values between 0 and 1, deciding how much of the cell state should be output.

In Olah (2015)

3.3. Autoencoders

In ANN, an autoencoder is a neural network whose training labels are its input. An autoencoder can be logically divided into two consecutive parts: an encoder, and a decoder (Figure 11). This type of neural network is trivial when the encoder size is greater than that of the input. When it is smaller, asumming there is little enough colinearity, the model is forced to lose information about the input. By learning to recreate the input as best it can in spite of compression, it works well at both dimensionality reduction and denoising. Intuitively, it will cling to features in the data endowed with pattern, effectively filtering out noise.

After training an autoencoder, interest usually lies in the encoder half. The reconstruction weights are disregarded and the latent variables are used instead of the input, for the relevant application. This process can be likened to that of principal component analysis (PCA).

One variation of autoencoders is the stacked autoencoder (SAE). To add depth to the previous autoencoder, one can train a second autoencoder on the encoded variables, i.e., a second autoencoder will train to encode and decode the encoded variables of the first autoencoder. This process can be repeated an arbitrary amount of times. The resulting encoded layer has extracted deeper and deeper features in the data. Figure 15 depicts an SAE in its extent. Figures 12 through 14 illustrate the process of training each autoencoder step by step.

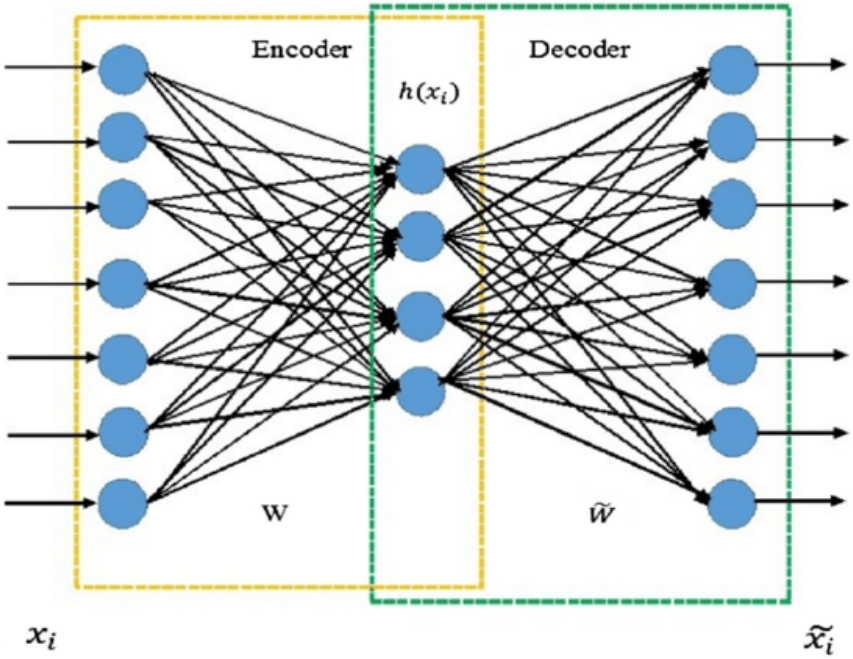


FIGURE 11. An autoencoder with a single layer, referred to as a sparse autoencoder. The latent variables contain compressed filtered information about the input.

In Ahmed, Wond, Nandi (2017)

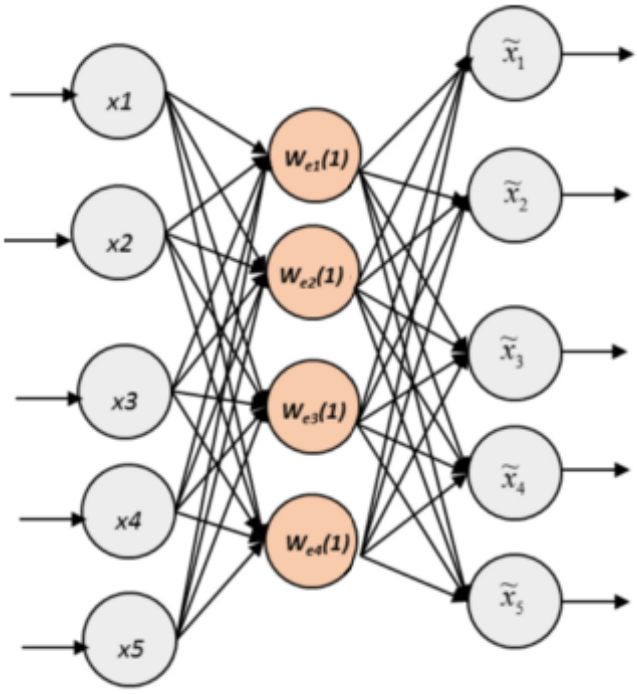


FIGURE 12. A first sparse autoencoder encodes and decodes the input.

In Jonnalagadda (2018)

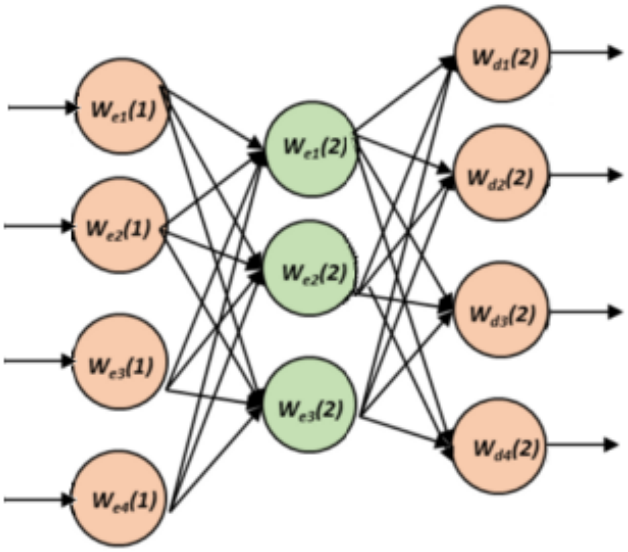


FIGURE 13. A second sparse autoencoder encodes and decodes the encoded input.

In Jonnalagadda (2018)

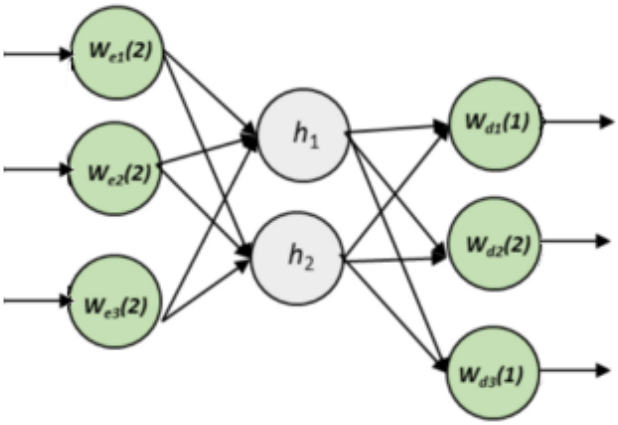


FIGURE 14. A third sparse autoencoder encodes and decodes the encoded encoded input.

In Jonnalagadda (2018)

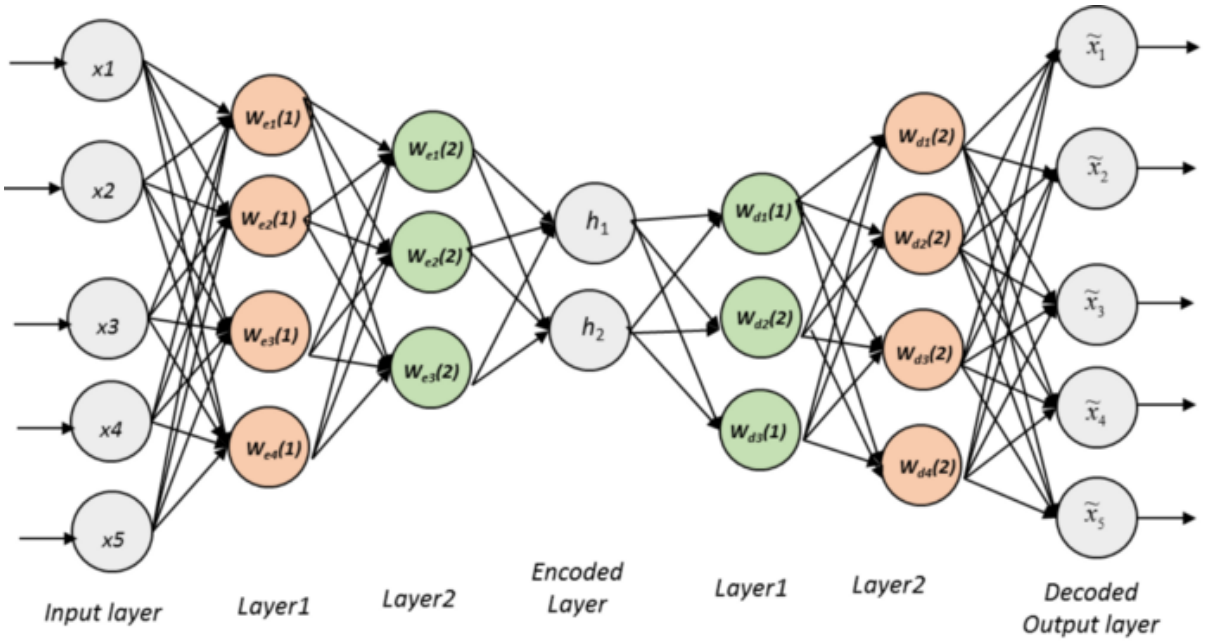


FIGURE 15. The full stacked autoencoder, made up of three sparse autoencoders.

In Jonnalagadda (2018)

CHAPTER 4

Methodology

This chapter describes the process of generating the next-day forecasts of S&P500. Three models are built as a combination of data preprocessing techniques, stacked autoencoders (SAE), and long short-term memory (LSTM). They all have the same basic framework, differing mostly in output. The 'price model' outputs its forecast for the next-day adjusted closing price. The 'change model' outputs the predicted relative change in price. Lastly, the 'binary model' solely outputs the probability that the price will increase. Because their output is different, they learn different things and in different ways. The quality of forecasts is evaluated in terms of prediction error and profitability. In the case of the price model and the change model, predictive power can be compared. There is interest in finding which of these two models generates better forecasts, and which of the three generates the most profit.

The models were written in Python 3.7 (Van Rossum & Drake, 2009) with the help of the Keras library for artificial neural networks (Chollet, 2015). The scripts used, along with the data files, are accessible from `https://github.com/LumpyJumbo/NeuralNet`.

Figure 3 provides a visual for the overall structure of the models.

4.1. Data

Daily data has been collected for the period of 01-01-2005 to 01-01-2020. With the primacy of recentness notwithstanding, this period was selected such that the dataset was large enough to train, validate, and test the models, while capturing different market behaviors and historical contexts in an effort to make the models generalize better. It has been split into three subsets: training set (first 80% of observations), validation set (next 10% of observations), and testing set (last 10% of observations). The training set is used for fitting the models for a given set of hyperparameters, the validation set is used for tuning, and the testing set is used to generate the results.

The data has been divided into three categories (Figure 1). Multiple stock indices from around the world were selected to exploit the potential correlations between them and with S&P500. For each index there are six daily variables: Open, Close, High, Low, and Adj. Close prices, and volume, with the exception that volume data for FTSE100 was discarded due to a long period of missing values. Following the practice of Bao, Yue,

& Rao (2017), different technical indicators used in trading were generated for each stock index, when applicable. Several macroeconomic variables were also included in order to exploit further correlation, and in the hope that the models could find some leading behavior for breaks in the series. In total, the dataset boasts 154 variables. Refer to Figure 2 for the data sources.

Stock Indices	**Technical Indicators**	**Macroeconomic Variables**
S&P500	Bollinger Bands	DAAA
DAX	EMA20	DCOILWTI
CAC40	MACD	DFF
NI225	MEMA	T10YIE
HSI	CCI	Treasury Yield 10 Years
DJIA	ATR	US30FUT
NASDAQ	Momentum	Dollar Index
FTSE100	ROC	
	SMI	
	AD	

FIGURE 1. The different variables used in the models.

Yahoo Finance https://finance.yahoo.com/	S&P500	DAX	CAC40	NIKKEI225
	HANG SENG	DJIA	NASDAQ	Treasury Yield 10 years
Federal Reserve Economic Data https://fred.stlouisfed.org/	DAAA	DCOILWTICO	DFF	T10YIE
Investing.com https://www.investing.com/	FTSE100	US30FUT	Dollar Index	

FIGURE 2. Data sources.

4.2. Preprocessing

Before the data can be used for the models, a few matters need to be addressed. First, while all variables have daily frequency, they do not all share a domain. To fix this, the domain for the dataset is fixed as S&P500's domain. Second, there are several missing values in the data that cannot be processed by the models. Linear interpolation is applied to fill these holes in the data. Third, the data is of the incorrect shape to be read by the models, which use LSTM. The input for LSTM is two-dimensional, whereas the data has one-dimensional observations. A time window of 35 days has been selected through trial and error (refer to the tuning chapter). The dataset is converted into a three-dimensional tensor, where each observation contains the values of each variable over 35 days. Each observation corresponds to a window that's rolling through the original dataset daily. Fourth, the data is not scaled appropriately. For each window, each variable is rescaled through a linear transformation such that its values lie in the $[0, 1]$ interval. For most

variables, this means subtracting their minimum in that window, and dividing by their maximum minus their minimum in that window. However, in an attempt to preserve and make evident for the models the spatial relationship between equivalently scaled variables (such as High and Low prices of a given index, or such prices and their respective upper and lower Bollinger bands), such equivalently scaled variables have been rescaled as a group. In other words, for every variable in a group, the minimum of the group for a given window is subtracted, and they are divided by the maximum minus the minimum of the group for that window. The result of all of these changes is a three-dimensional dataset, where the scale of each group of variables and the scale across different variables is the same each sequence of 35 days. This transformation is easily reversible at the output end.

4.3. Stacked Autoencoder

Bao, Yue, & Rao (2017) found that the use of SAE to extract deep features in the data before forecasting with LSTM generates better results in terms of both predictive accuracy and profitability, regardless of the stock index they tested (including S&P500). Following these results, this work uses an SAE built from three LSTM autoencoders. LSTM autoencoders were used so that the SAE encoded each 35 day sequence as a sequence, rather than encoding each daily observation by itself. The choice of three autoencoders was made as an arbitrary tradeoff between depth and parcimony. Training was performed using Adam to minimize mean square error. As will be detailed in the tuning chapter, not all hyperparameters related to she shape of neural networks were made variable to tuning, due to the computational constraints of this experiment.

To generate the size of each autoencoder hidden layer, a latent dimension term was defined, referring to the size of the final encoded data. This corresponds to the hidden dimension of the last autoencoder. The sizes of the first two autoencoders' hidden layers are defined such that the input dimension and the hidden dimensions of all autoencoders are equidistant. In other words, each encoder reduces the dimension of the data by the same amount. This facilitated tuning, as only one variable, the latent dimension, was optimized. A latent dimension of 85 was obtained through trial and error.

4.4. LSTM Forecaster

Finally, the processed data is input to a three layer LSTM network. The reason for three hidden layers is the same as for the SAE. All layers are set to be the same size to expedite tuning. A size of 120 was defined through trial and error for the price model and change model, and a size of 30 for the binary model. While the loss function for training was defined as the mean square error for both the SAE and the forecaster for the price model and change model, the loss function for the forecaster for the binary model was defined as the binary crossentropy function, which is more suited for binary classification (Brownlee,

2019). Adam was again used as the optimizer.

4.5. Training and Tuning

All the training and tuning were done using an Intel® Core i5-7400 processor and a cap of 8GB of RAM. As such, and without the help of GPU processing, tuning was very limited, and did not include every possible hyperparameter. First, hyperparameters were initialized arbitrarily within the ranges one would guess make sense. Second, one hyperparameter is changed incrementally in one direction, and then the other. Third, the hyperparameter is set to the value that perfomed best within those changes. This is repeated for all hyperparameters a few times per parameter. The final hyperparameter combination used for each model can be found in the Python scripts provided in `https://github.com/LumpyJumbo/NeuralNet`. To compare the predictive accuracy of each iteration, the mean absolute scaled error (MASE) was used. Since one of the hyperparameters being tuned is the time window, and since increasing the time window removes observations, it is important to use a measure that allows for the comparison of two models of different domain. MASE is calculated by dividing the mean absolute error of the model's forecasts by the mean absolute error of a naive predictor. This naive predictor is defined as the current-day price. One potential benefit of using MASE is that it provides insight, by itself, of how well the forecasting model is performing as compared to a naive prediction. If MASE$>$ 1, then the model gives, on average, worse results than just assuming the price will not change. Besides the MASE, models were also validated based on profitability. In general, profitability was favored as a criterion, as it was noted that lower MASE meant the models would approximate the naive predictor, as none achieved MASE$<$ 1 in any iteration in the validation set.

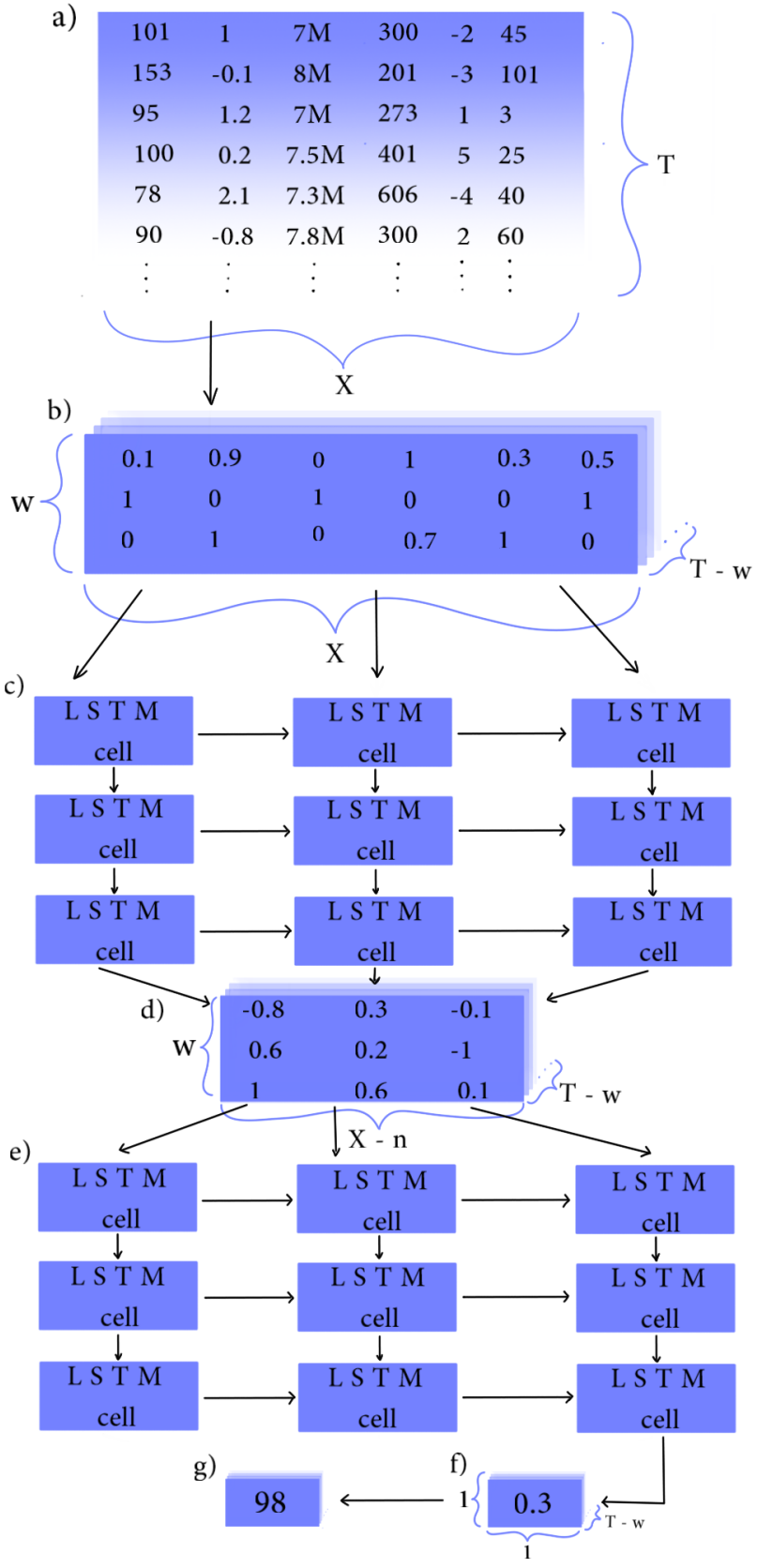


FIGURE 3. An overview of the basic framework used for each model. Legend: a) raw data; b) preprocessed (scaled) data; c) LSTM encoders; d) encoded data; e) LSTM predictor; f) scaled forecasts; g) raw scale forecasts (final output). X is the number of variables in the raw data, T is the number of observations in the raw data, W is the time window used, and n is dimensionality reduction.

CHAPTER 5

Results

5.1. Predictive Accuracy

Figure 1 shows S&P500 data for the testing period plotted alongside its forecasts by the price model and the change model. Not a lot can be gleaned from this graph alone aside that the change model's predictions seem to follow the real values closer than the price model's. Although a hasty glance might suggest a snug fit, a closer look reveals that, while the trajectory of the three curves is equivalent, the predictions are in most cases out of phase with the real values by one day, just like the naive predictor. This is especially true for the change model, which is why it seems to follow the data closer.

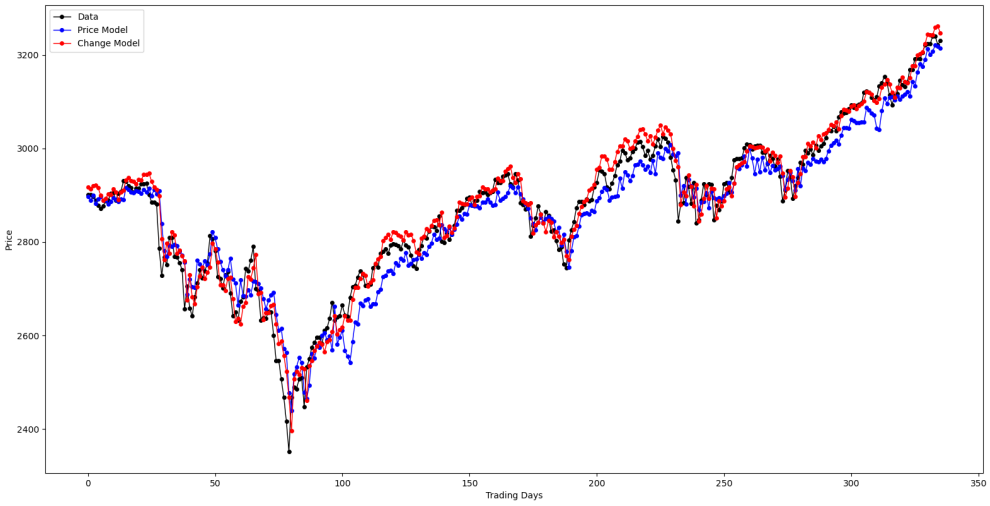


FIGURE 1. S&P500 and its forecasts for the testing period.

To quantify the predictive accuracy of the models, the mean absolute scaled error (MASE) and mean absolute error (MAE) were generated. As can be seen in Figure 2, neither of the models performed better on average than the naive predictor. During tuning, it was noted that neither model managed to have MASE< 1 for any iteration in the validation set. It was also noted that the lowest MASE were those who would get arbitrarily close to 1, due to the training finding a tempting minimum in approaching the naive predictor, shutting out most of the features in the data. This suggests that these model specifications are unable to find anything besides noise in most of the data features, as making effective use of them for a greater fit during training makes them generalize poorly.

	Naive Predictor	**Price Model**	**Change Model**
MASE	1	1.878	1.276
MAE	18.302	34.364	23.379
std(AE)	18.568	25.76	21.005
$\frac{std(AE)}{MAE}$	1.015	0.75	0.898

FIGURE 2. Mean absolute scaled error, mean absolute error, absolute error standard deviation, and absolute error relative standard deviation for naive predictor and forecasting models.

In other words, the relationships established between variables during training is spurious.

Another notable pattern in the models' performance can be seen in Figure 3, which shows the deviations of the forecasts from the actual values. With time out of the picture, it can be seen that forecasting error is greater for lower prices of S&P500. This is likely a consequence of how the neural networks are trained. Notice that the lower prices are also the most infrequent. Since the way the model is fit is by minimizing the average batch loss, these obsevations are largely ignored, behaving like outliers. However, larger prices are also quite infrequent (although admittedly not as much) and the error is noticeably lower for those. This is likely due to volatility. Breaks in the series appear most frequently as sharp drops in price rather than jumps. This suggests the models struggle with predicting breaks. Since these periods of high volatility are the most interesting and most profitable, it might behoove one to modify how the neural networks used learn in order to make them more sensitive to these breaks, either by adding to the data, modifying the loss function, or changing how it is optimized.

5.2. Profitability

While the models had higher MAE than the naive predictor, they might still hold useful information that a naive predictor would lack: information about direction. While on the one hand their deviation from the actual price is greater, on average, than that of the naive predictor, on the other hand they are able to model volatility and may perhaps get the direction of price change correctly. Each model was evaluated in terms of their profits following a simple trading strategy: buy when the price is predicted to increase on the next day, and sell when the price is predicted to decrease on the next day (the binary model is set to buy when the probability of a price increase is greater than 0.5). For the binary model, a second strategy was also used, consisting in investing in proportion given by the probability. The naive predictor for these results is turned into a buy and hold strategy (referred to as 'No Trading'). A second naive strategy is a 50/50 split in investment, to portray the expected value of a strategy which expects it to be equally likely for the price to increase or decrease each time. The results are shown in Figure 4.

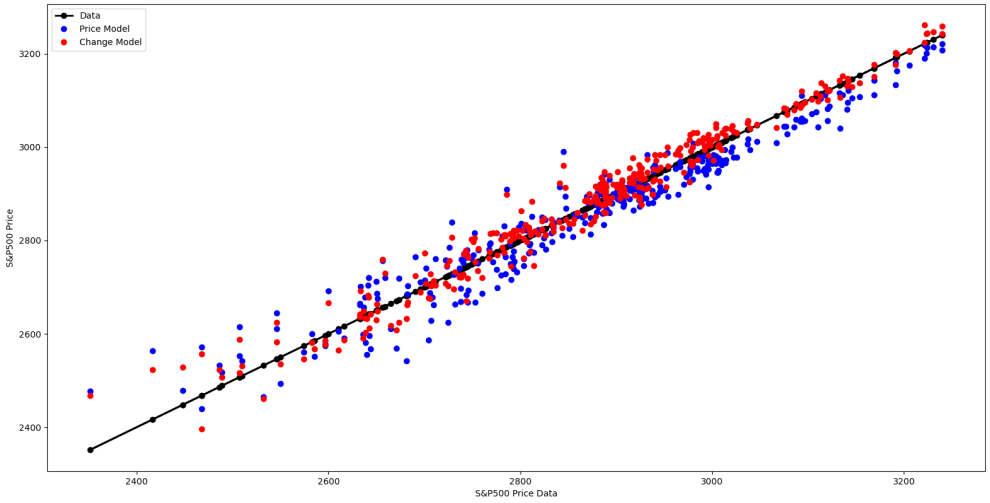


FIGURE 3. S&P500 timeless forecasting error.

	No Trading	**Price Model**	**Change Model**	**Binary Model**	**Binary Model Weighted Strategy**	**50/50**
Daily Return (avg)	0.03%	0.014%	0.032%	0.019%	0.025%	0.016%
Standard Dev	0.949 pp	0.744 pp	0.79 pp	0.725 pp	0.653 pp	0.475 pp
2σ Interval	[-1.868%, 1.928%]	[-1.474%, 1.501%]	[-1.548%, 1.612%]	[-1.431%, 1.468%]	[-1.28%, 1.331%]	[-0.933%, 0.965%]
Min - Max	[-3.286%, 4.959%]	[-3.286%, 4.959%]	[-3.286%, 4.959%]	[-3.286%, 4.959%]	[-2.88%, 4.897%]	[-1.643%, 2.48%]
Total Test Profit	10.87%	4.7%	11.68%	6.74%	9.08%	5.69%
Theoretical Max	214.81%	214.81%	214.81%	214.81%	214.81%	214.81%

FIGURE 4. Profitability results (any inflation, tariffs, comissions, or dividends are not considered). Average daily return is a geometric mean.

One result that is immediately apparent is that returns are less volatile when trading according to any of the models. This is less significant than it may seem, as all volatility lies in the asset price, being 0 when there is divestment. As such, average daily return is necessarily highest when there is no selling. By the same logic, it is expected that the 50/50 strategy be half as volatile, which is exactly what is verified. Because the Min-Max intervals are largely the same, it is unlikely the lower standard deviations imply virtuous trading rather than less trading, be it virtuous or vicious.

Note that it is extremely relevant to the results when the trading started and ended. If the trading ends when the price is greater than it was when it started, then the buy and hold strategy will yield positive return, and vice versa. This effect is present regardless of trading strategy. Take the binary model with the weighted strategy, for instance, and compare it with the 50/50 strategy. The resulting profits from the binary model are greater, but are they a consequence of the model correctly identifying the price movements, or of simply investing more, approximating the buy and hold strategy? This is likely also the explanation for the change model's performance, which is comparable to

that of the buy and hold strategy.

Notice that the binary model with the weighted strategy (henceforth BMWS) yields greater profit than with the full investment/divestment strategy (BM). By definition, BMWS will invest even when BM divests, and vice versa. This means it will yield greater return when BM is mistaken, but lower return when BM is correct. The net effect being positive is indicative of BM's inadequacies. Note that the greater return for BMWS is most likely due to avoidance of mistakenly investing too much when the price falls sharply, as can be observed in the Min - Max interval. In other words, and as can be observed for the other models, the biggest obstacle to greater returns lies in predicting those sharp breaks in the series when the price plummets, which the models were not able to do. Furthermore, they all fell quite far from the theoretical maximum achievable profit.

CHAPTER 6

Conclusions

Given the data and the architecture of the models, it seems through training they were not able to find a non-spurious relationship between the different features in the data that allowed them to generalize well. Those which performed best were those which approximated the naive predictor, i.e., those which ignored most features and settled for a low-hanging minimum. It is entirely possible that it was the only generalizable minimum available to them. The inability to find a better minimum could be evidence validating the efficient market hypothesis.

Out of the three models, only the change model accrued more profit than a buy and hold strategy, and not considerably so. Results revealed that the models are particularly poor at predicting breaks in the series, which were frequently characterized by sharp falls in price. Since these moments are also when there is the most potential for profit, the neural networks should be changed or trained such that they can effectively predict them, if possible.

The models can be further improved with better methods for hyperparameter training. With a faster way of training, using GPU processing for instance, a wider selection of hyperparameters can be defined for optimization, and the optimization process can be more rigorous (for instance, by using a more advanced optimization method and cross-validation). These changes can be affected in future studies, for which the files used in this research can be found in `https://github.com/LumpyJumbo/NeuralNet`.

References

Ahmed, H.O.A., Wong, M.L.D. and Nandi, A.K. (2018). Intelligent condition monitoring method for bearing faults from highly compressed measurements using sparse over-complete features. Mechanical Systems and Signal Processing, 99, pp.459-477.

Bao, W., Yue, J., & Rao, Y. (2017). A deep learning framework for financial time series using stacked autoencoders and long-short term memory. PloS one, 12(7).

Brownlee, J. (2017). A Gentle Introduction to Exploding Gradients in Neural Networks. Available at: `https://machinelearningmastery.com/exploding-gradients-in-neural-networks/`

Brownlee, J. (2019). A Gente Introduction to Cross-Entropy for Machine Learning. Available at: `https://machinelearningmastery.com/cross-entropy-for-machine-learning/`

Chollet, F. et al (2015). Keras. Available at: `https://github.com/fchollet/keras.`

Donoho, D. L., & Johnstone, I. M. (1995). Adapting to unknown smoothness via wavelet shrinkage. Journal of the american statistical association, 90(432), 1200-1224.

Estrella, A. (2007). Extracting business cycle fluctuations: what do time series filters really do?. FRB of New York Staff Report, (289).

Feng, C. (2020). 'Vanilla Recurrent Neural Network' in Machine Learning Notebook. Available at: `https://calvinfeng.gitbook.io/machine-learning-notebook/supervised-learning/recurrent-neural-network/recurrent_neural_networks`

Fernández-Cabán, P.L., Masters, F.J. and Phillips, B.M. (2018). Predicting roof pressures on a low-rise structure from freestream turbulence using artificial neural networks. Frontiers in Built Environment, 4, p.68.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press. Available at: `https://www.deeplearningbook.org/`

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.

Jonnalagadda, V.K. (2018). Sparse, Stacked and Variational Autoencoder. Available at: `https://medium.com/@venkatakrishna.jonnalagadda/sparse-stacked-and-variational-autoencoder-efe5bfe73b64`

Lachiheb, O., & Gouider, M. S. (2018). A hierarchical Deep neural network design for stock returns prediction. Procedia Computer Science, 126, 264-272.

Olah, C. (2015). Understanding LSTM Networks. Available at: `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`

Ramsey, J. B. (1999). The contribution of wavelets to the analysis of economic and financial data. Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences, 357(1760), 2593-2606.

Van Rossum, G. & Drake, F.L. (2009). Python 3 Reference Manual, Scotts Valley, CA: CreateSpace.

White, H. (1988). ECONOMIC PREDICTION USING NEURAL NETWORKS: THE CASE OF IBM DAILY STOCK RETURNS. Proceedings of International Conference on Neural Networks, 451-II-458-II