NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATION

BSc THESIS

Stock Market Predictions using LSTM Neural Networks

Konstantinos G. Tsiaras

Supervisor: **Panagiotis Stamatopoulos,** Assistant Professor

ATHENS

FEBRUARY 2020

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Χρηματιστηριακές Προβλέψεις με χρήση Νευρωνικών Δικτύων LSTM

Κωνσταντίνος Γ. Τσιάρας

Επιβλέπων: **Παναγιώτης Σταματόπουλος,** Επίκουρος Καθηγητής

ΑΘΗΝΑ

ΦΕΒΡΟΥΑΡΙΟΣ 2020

BSc THESIS

Stock Market Predictions using LSTM Neural Networks

Konstantinos G. Tsiaras

S.N.: 1115201500165

SUPERVISOR: **Panagiotis Stamatopoulos,** Assistant Professor

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Χρηματιστηριακές Προβλέψεις με χρήση Νευρωνικών Δικτύων LSTM

Κωνσταντίνος Γ. Τσιάρας

Α.Μ.: 1115201500165

ΕΠΙΒΛΕΠΩΝ: **Παναγιώτης Σταματόπουλος,** Επίκουρος Καθηγητής

ABSTRACT

The purpose of this thesis is to analyze the extent to which the historical financial data of a stock suffice to make meaningful predictions about its future price with the use of Machine Learning. The intuition behind our task is that, as the price of a stock fluctuates, it is assumed to follow certain patterns which we hope to capture using Deep Learning and utilize for future predictions.

Firstly, we will expand on the necessary theoretical background information regarding Machine Learning, focusing particularly on the neural networks that will later be used. Following that, we will examine how existing research regarding stock market forecasting using similar techniques fared in the past and we will proceed to propose a regression model using Long short-term memory (LSTM), a Recurrent Neural Network architecture most suitable for this kind of tasks. Finally, using data from Athens Stock Exchange, we will attempt to make predictions about the future trajectories of the stocks' prices and draw conclusions from them.

SUBJECT AREA: Pattern Recognition

KEYWORDS: Machine Learning, Recurrent Neural Networks (RNN), Long short-term memory (LSTM), Stock Market Prediction, Regression

ΠΕΡΙΛΗΨΗ

Ο σκοπός αυτής της πτυχιακής είναι να αναλυθεί ο βαθμός στον οποίο τα ιστορικά οικονομικά δεδομένα μια μετοχής επαρκούν για να πραγματοποιηθούν ουσιώδεις προβλέψεις της μελλοντικής της τιμής με τη χρήση Μηχανικής Μάθησης. Η διαίσθηση πίσω από την εργασία μας είναι ότι, καθώς η τιμή μιας μετοχής κυμαίνεται, θεωρείται πως ακολουθεί κάποια μοτίβα τα οποία ελπίζουμε να αντιληφθούμε χρησιμοποιώντας Βαθιά Μάθηση και να τα αξιοποιήσουμε για μελλοντικές προβλέψεις.

Αρχικά, θα παραθέσουμε το απαραίτητο θεωρητικό υπόβαθρο σχετικό με τη Μηχανική Μάθηση, εστιάζοντας ιδιαίτερα στα νευρωνικά δίκτυα που θα χρησιμοποιηθούν στη συνέχεια. Έπειτα, θα εξετάσουμε τις ήδη υπάρχουσες έρευνες σχετικά με τις προβλέψεις χρηματιστηριακών αγορών που χρησιμοποιούν παρόμοιες τεχνικές και θα προτείνουμε ένα μοντέλο παλινδρόμησης χρησιμοποιώντας Μακροχρόνια βραχυχρόνια μνήμη (LSTM), μία αρχιτεκτονική Επαναλαμβανόμενων νευρωνικών δικτύων (RNN) που είναι πλέον κατάλληλη για τέτοιου είδους προβλήματα. Τέλος, χρησιμοποιώντας δεδομένα από το Χρηματιστήριο Αθηνών, θα επιχειρήσουμε να πραγματοποιήσουμε προβλέψεις για τις μελλοντικές πορείες των τιμών των μετοχών και να εξάγουμε συμπεράσματα από αυτές.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Αναγνώριση Προτύπων

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Μηχανική Μάθηση, Επαναλαμβανόμενα Νευρωνικά Δίκτυα (RNN), Μακροχρόνια βραχυχρόνια μνήμη (LSTM), Χρηματιστηριακές Προβλέψεις, Παλινδρόμηση

ACKNOWLEDGEMENTS

I would like to wholeheartedly thank all people near and dear to me that supported me both in completing this thesis and throughout my undergraduate studies in general.

Furthermore, I would also like to thank my professor Panagiotis Stamatopoulos both for his guidance in choosing the subject for this thesis as well as his immediate responses whenever I required further help or advice.

CONTENTS

1. BACKGROUND 14

1.1. Types of Machine Learning 14
1.1.1. Supervised Learning 14
1.1.1.1. Classification 14
1.1.1.2. Regression 14
1.1.2. Unsupervised Learning 15
1.1.2.1. Clustering 15
1.1.2.2. Association 16
1.1.3. Reinforcement Learning 16

1.2. Neural Networks 17
1.2.1. Network Architecture 17
1.2.1.1. Neuron 17
1.2.1.2. Activation Function 19
1.2.1.3. Layers 20
1.2.2. Training risks 21
1.2.3. Cost function 23
1.2.4. Gradient Descent 24
1.2.5. Backpropagation 25
1.2.6. Types of Neural Networks 26
1.2.6.1. Feed-forward Neural Networks 26
1.2.6.2. Convoluted Neural Networks 26
1.2.6.3. Recurrent Neural Networks 26
1.2.6.4. LSTMs 28

1.3. Stock Market Information 32
1.3.1. Stock 32
1.3.2. Stock Market and Stock Exchanges 33
1.3.3. Stock Market Index 33

1.4. Stock Market Prediction 33
1.4.1. Random Walk Hypothesis 33
1.4.2. Methods of Market Analysis 33
1.4.3. Related Work 34
1.4.3.1. Stock Data-related approaches 34
1.4.3.2. Predictions using NLP 34

2. PROPOSED APPROACH 35

2.1. Task Definition 35

2.2. Tools and Technologies **35**

2.3. Dataset **35**

2.4. Focusing on Stock Indices **36**

2.5. Data Preprocessing **36**
2.5.1. Data Cleaning 36
2.5.2. Normalization 36

2.6. Training Strategy **37**
2.6.1. Training window 37
2.6.2. Training labels 38

2.7. Model Design **40**

2.8. Parameter tuning **40**

3. RESULTS **42**

3.1. Metrics **42**

3.2. Types of prediction **42**
3.2.1. Single-step prediction 42
3.2.2. Sequence predictions 45
3.2.2.1. Multi-step prediction 45
3.2.2.2. Single-step sequence prediction 46

4. CONCLUSION **53**

4.1. Conclusions **53**

4.2. Future Work **53**

ABBREVIATIONS – ACRONYMS **54**

REFERENCES **55**

LIST OF FIGURES

Figure 1: A Classification example 14
Figure 2: A Regression example 15
Figure 3: A Clustering example, using K-Means algorithm 16
Figure 4: Output graph (right) of a Sigmoid Neuron without bias (left)...........................18
Figure 5: Output graph (right) of a Sigmoid Neuron with bias (left)................................18
Figure 6: Frequently used Activation Functions..20
Figure 7: Examples of Underfitted (left), Optimal (center) and Overfitted (right) models ... .22
Figure 8: Testing and Training Errors plotted against Training Steps............................22
Figure 9: Simulation of Gradient Descent's convergence after multiple iterations 24
Figure 10: A stock's price candlesticks graph (left) and its respective “Close” line graph (right) 42
Figure 11: Single-step prediction of General Index...43
Figure 12: Model loss for Figure 11 43
Figure 13: Last 30 days of Figure 11 44
Figure 14: Multi-step prediction of General Index 45
Figure 15: Model loss for Figure 14 46
Figure 16: Prediction Sequences for General Index, after 90 epochs of training...........48
Figure 17: Model loss for Figure 16 48
Figure 18: Prediction Sequences for General Index, after 10 epochs of training...........49
Figure 19: Prediction Sequences for General Index, after 30 epochs of training...........50
Figure 20: Prediction Sequences for EEE's stock..51
Figure 21: Model loss for Figure 20 51

LIST OF IMAGES

Image 1: Brain Neuron (left) compared to Artificial Neuron (right) 17
Image 2: A simple Neural Network 19
Image 3: An example of a NN before (left) and after (right) applying dropout................23
Image 4: A Recurrent Neural Network, folded (left) and unfolded (right) 27
Image 5: A bidirectional RNN 28
Image 6: The repeating module of a standard RNN (above) compared to that of an LSTM (below)..29
Image 7: Dissection of an LSTM cell 30
Image 8: Flow of information in an LSTM cell...31
Image 9: A GRU cell...32
Image 10: Example of training windows with single outputs 38
Image 11: Example of training windows with prediction window outputs 39
Image 12: Example of testing windows generating a prediction sequence....................47

LIST OF TABLES

Table 1: Loss for General Index's predictions using different features 50
Table 2: Loss for EEE's predictions using different features..52

PREFACE

This thesis was conducted as part of my Bachelor's Degree in Computer Science at the Department of Informatics and Telecommunications of the National and Kapodistrian University of Athens, spanning about 6 months. The first half of these was devoted entirely to research, both regarding neural networks in general as well as specifically for related forecasting problems. The latter half was focused on the development of the necessary code and the actualization of the predictive experiments.

1. BACKGROUND

1.1. Types of Machine Learning

1.1.1. Supervised Learning

Supervised Learning is the task of generating a model that can map any valid input to some specific output. In order to achieve this, we split our available data – which should be in the form of a vector of inputs along with the desired output (label) for each – into two sets, the training set and a smaller test set. We then train our model and finally, in order to validate its accuracy, try it against the test set. If the results are unsatisfying then the model's parameters should be tweaked and the above process be repeated.

The most common classes of problems that are usually solvable using Supervised Learning are the following:

1.1.1.1. Classification

Classification is the problem of discovering to which of a set of known categories a new input belongs. Those categories – or classes – can be two (**binary classification**) or more (**multiclass classification**). A simple example of binary classification can be seen in Figure 1 where we have multiple datapoints of both healthy and diseased people and we're trying to draw a line so as to be able to predict a new person's health based on their Gene 1 and Gene 2 values.

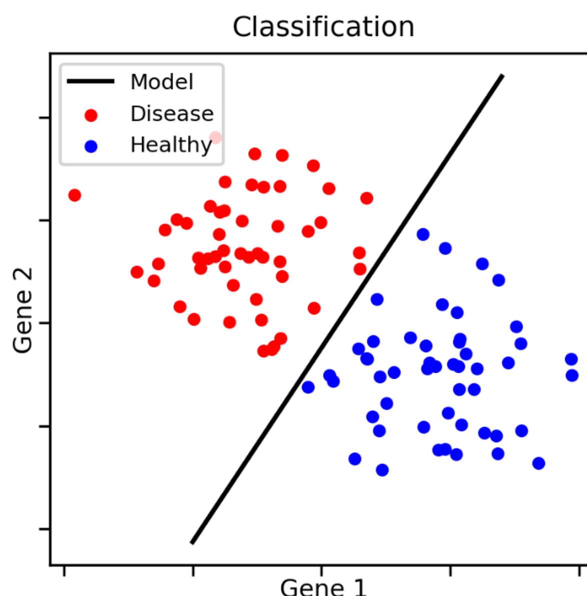


Figure 1: A Classification example

1.1.1.2. Regression

Regression is also used for predicting and forecasting but for continuous values as opposed to classification's discrete ones.

Continuing the theme of the above example, a regression one could be the following:

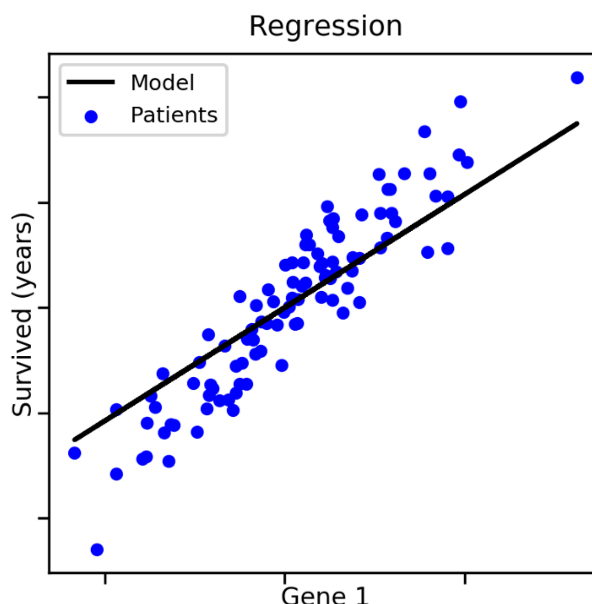


Figure 2: A Regression example

Here, if for a new patient we know either the years they've survived or their Gene 1 value, we can predict the other one with the help of the plotted line. It should be noted that in this simple case a mere line seemed enough – that's called **linear regression**. In cases where greater predictive power is needed, one may have to use more complex non-linear regression models.

Time series prediction, cases of which are weather forecasting as well as our task, stock market prediction, is a common type of regression problem and we will be revisiting it in later chapters.

1.1.2. Unsupervised Learning

Contrary to Supervised Learning, Unsupervised Learning does not require output variables for the provided input data, but it's also used in cases where the optimal end result is not predetermined. Its aim is instead to help find previously unknown patterns or correlation in the data.

Unsupervised Learning methods include:

1.1.2.1. Clustering

Clustering is the task of discovering inherent groupings in the available data. This is done by grouping data objects in such a way that objects in the same cluster are more similar (in some regard) to each other than those in other clusters.

Let's see one clustering example by plotting flowers based on their petal length and width (Figure 3). Here, contrary to supervised learning, we don't know beforehand how many different flower categories those belong to. We just provide a clustering algorithm (such as K-Means) with a desired number and it tries to best differentiate our data into this many clusters.

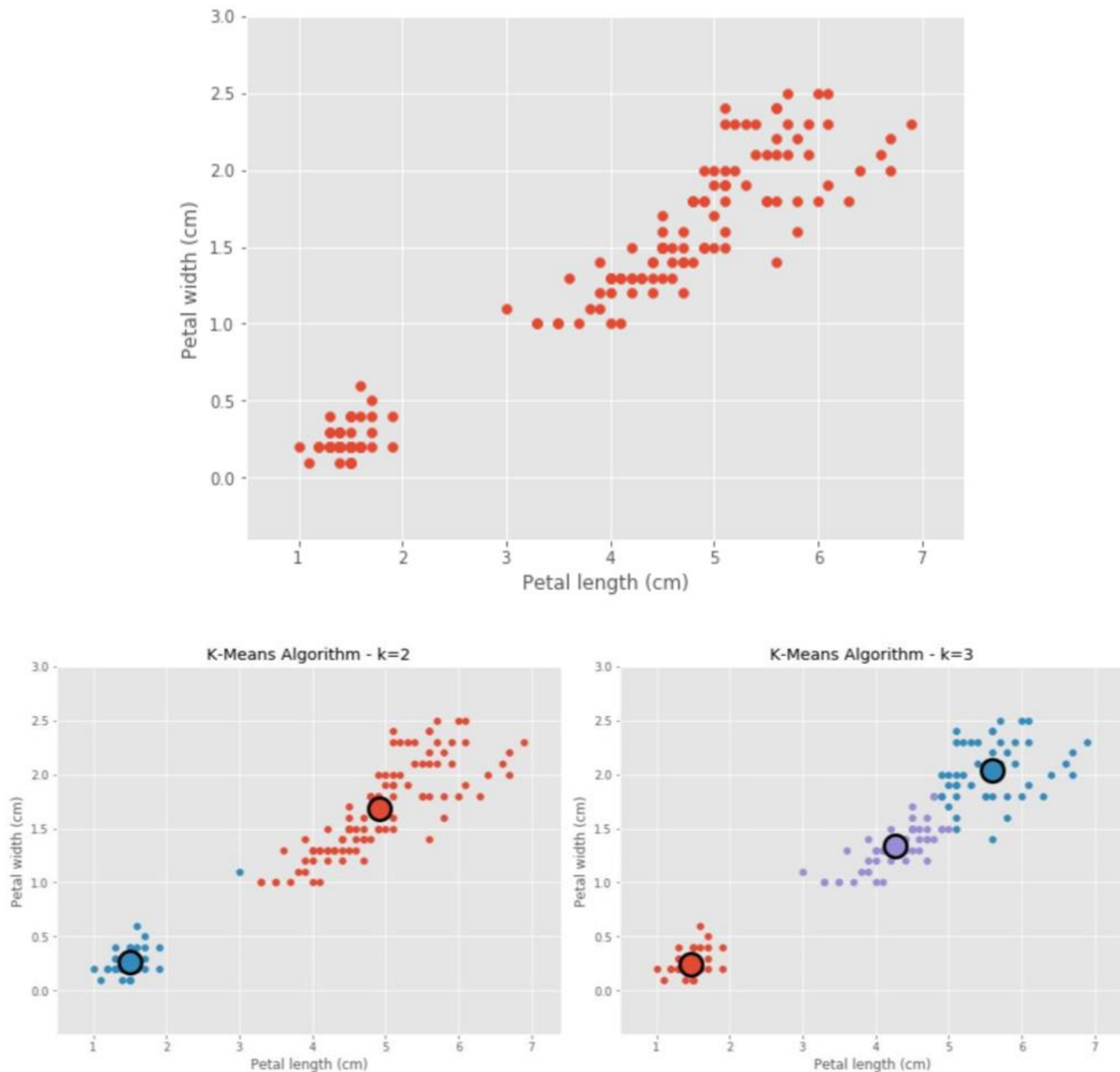


Figure 3: A Clustering example, using K-Means algorithm

1.1.2.2. Association

Association rule learning is a method for discovering relations between variables of the available data.

Association rule problems aren't exactly describable with a single figure and given that they're unrelated to our subject there's no reason to analyze them further, but one good case for association would be knowing the products that the customers of a certain store bought and trying to find out which ones people have a tendency to buy together.

1.1.3. Reinforcement Learning

This type of learning differs from the previously mentioned ones and it's concerned with how software agents should act in a specific environment, aiming to maximize some predefined reward function. Reinforcement Learning is not Supervised Learning since it does not require a labeled set of input data but it also differs from Unsupervised Learning in that we know the expected reward beforehand and model our agent with it in mind.

Their generality allows them to be applicable in a vast range of fields, such as genetic algorithms, game theory, as well as any problem that could also be tackled using Supervised Learning – in that case, the reward is given when the output for a training set input matches its given label.

1.2. Neural Networks

Artificial Neural Networks (ANNs) were originally designed with the purpose to mimic the architecture of the human brain and its ability to process and learn information. While newer advances in ANNs have weakened that original link between the two, the fundamental idea remains that both human brains and ANNs learn by example and improve over time. The basic unit of computation in both is the neuron, often called a node or unit in ANNs.

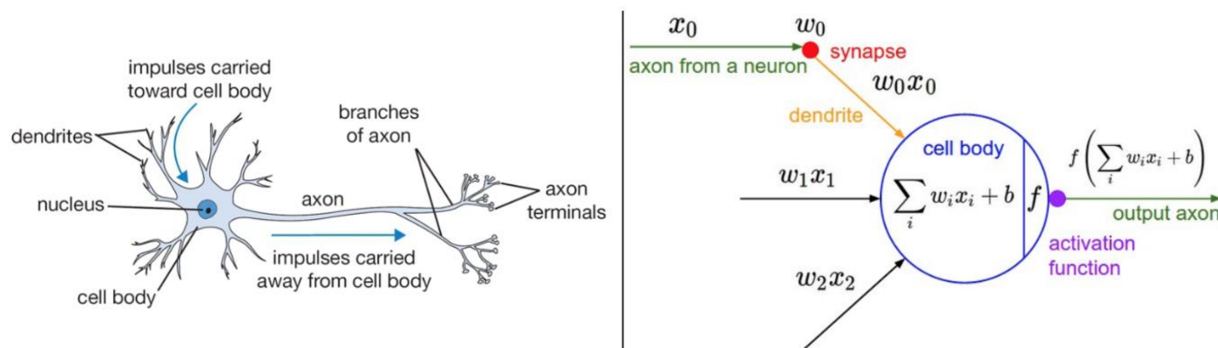


Image 1: Brain Neuron (left) compared to Artificial Neuron (right)

From this point onwards we will focus solely on Artificial Neural Networks and the abbreviation "NN" will refer to them.

1.2.1. Network Architecture

1.2.1.1. Neuron

First of all, let's focus on the design of a single neuron as depicted in Image 1 (right), often also referred to as a **perceptron**. It receives any number of inputs along with a real-valued weight for each one and constructs a linear combination, possibly adding a constant value for that specific neuron – known as **bias** – in the form of $\sum_i \ w_i x_i + b$. Then, this output value is inputted to an activation function, the purpose and characteristics of which we'll explain next up.

Before doing that however, let's examine the purpose of the bias and why its utility cannot be replaced by the weighted inputs. We'll do that by example, using the simplest possible neuron with just one input and a sigmoid activation function (which, while we haven't defined yet, does not affect our observations which would hold true with any other in its place). This neuron and the graph depicting its output value relative to its input *x* can be seen in Figure 4.

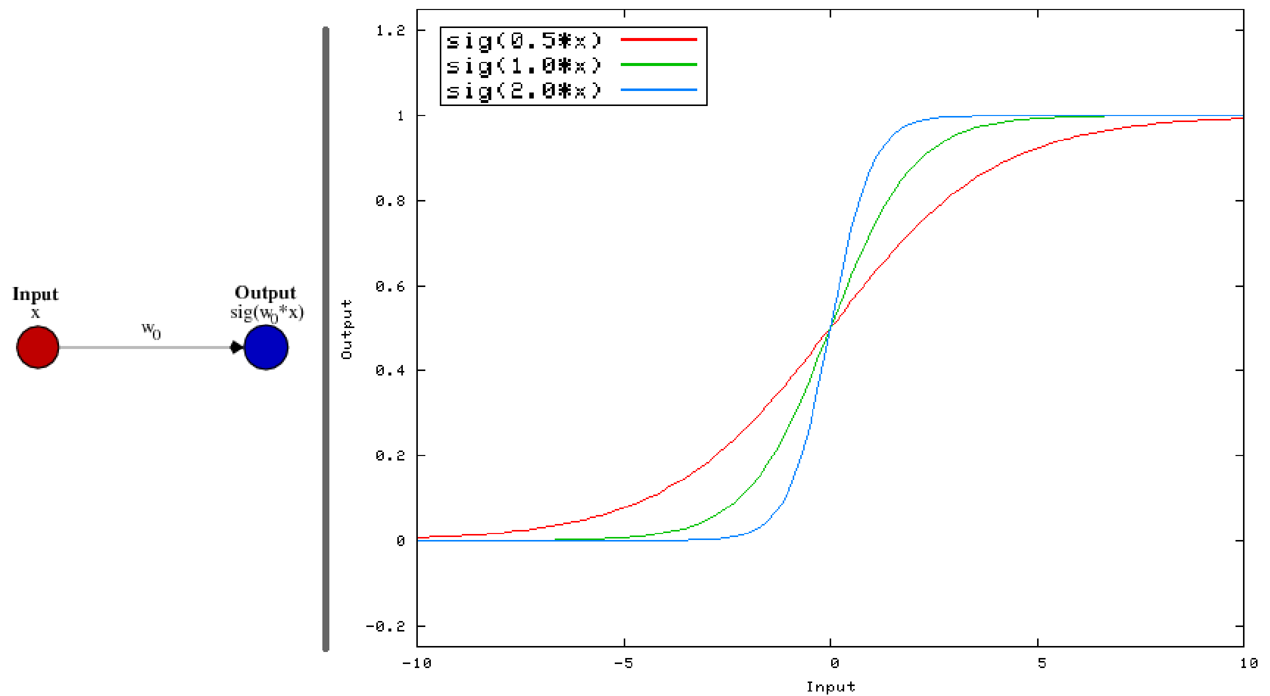


Figure 4: Output graph (right) of a Sigmoid Neuron without bias (left)

The problem here is that regardless of the input's and the weight's values the neuron's output is always zero-centered. Obviously, for flexibility's sake, we would like to be able to overcome that limitation. That is where the bias comes in:

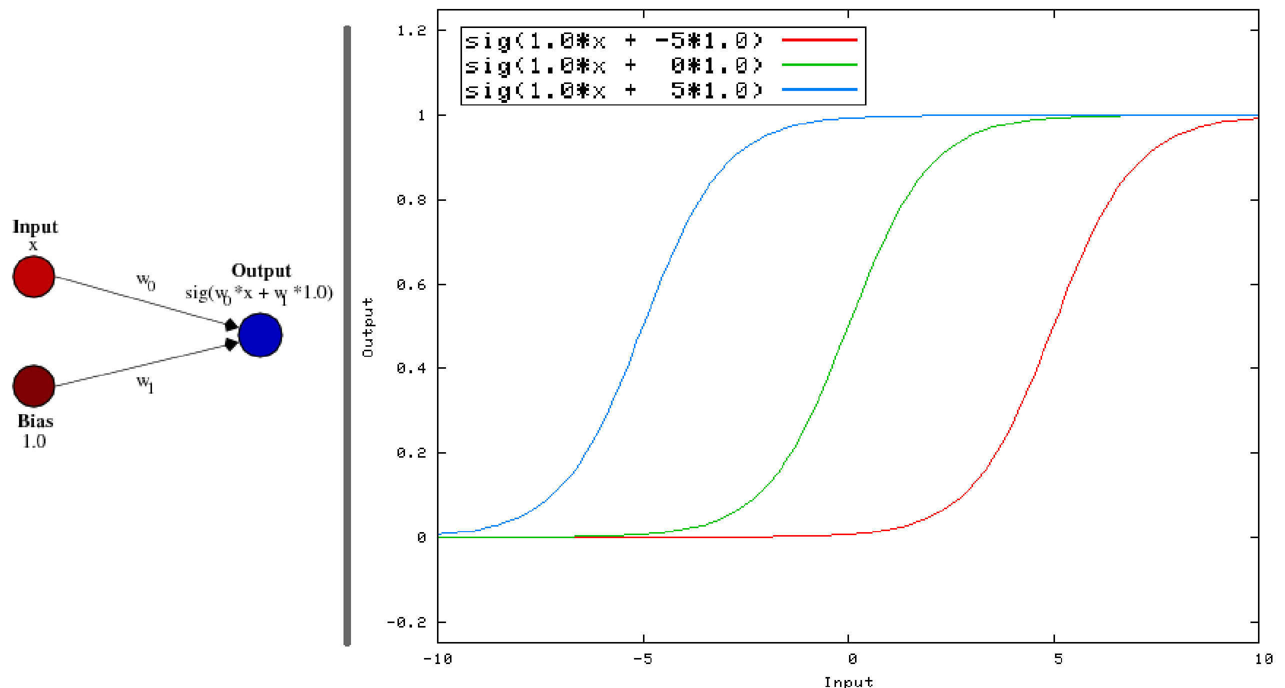


Figure 5: Output graph (right) of a Sigmoid Neuron with bias (left)

As we can see, changing the bias's value effectively allows us to shift the neuron's output to the left or right.

Despite all that, it is worth noting that sometimes the bias is not considered a fundamental part of the neuron and, depending on the task as well as the size of the overall network, it may be omitted.

1.2.1.2. Activation Function

The output of the **activation function** of a neuron is, for all intents and purposes, the output of that neuron. With some activation functions, such as *Sigmoid* or *tanh*, that output may always be in a closed interval (such as $[0, 1]$ or $[-1, 1]$ respectively), with values near the right end signifying that the neuron should to be activated, while those near the left one meaning that the neuron is to remain inactive. For other activation functions that interval may be left-closed as is the case for *ReLU*, or it can even be open – an example of which is *Leaky ReLU*.

At this point it's not yet evident why we would need an activation function. Besides, if our only requirement was to have a result in some interval (a, b), it would be much simpler to, say, normalize the neuron's output to that interval and be done with it. For that purpose, let's examine the final output of the simple NN depicted below:

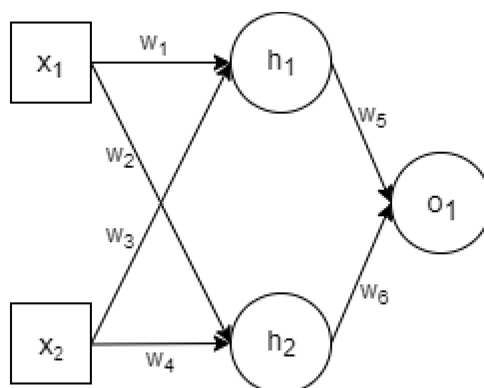


Image 2: A simple Neural Network

$$o_1 = y(w_5 h_1 + w_6 h_2) = y\big(w_5 \cdot y(w_1 x_1 + w_3 x_2) + w_6 \cdot y(w_2 x_1 + w_4 x_2)\big)$$

Assuming a hypothetical linear activation function $y(x) = kx$, where k is a constant, the above can be rewritten as: $o_1 = k\big(w_5 k(w_1 x_1 + w_3 x_2) + w_6 k(w_2 x_1 + w_4 x_2)\big) = k^2(w_5 w_1 x_1 + w_5 w_3 x_2 + w_6 w_2 x_1 + w_6 w_4 x_2)$.

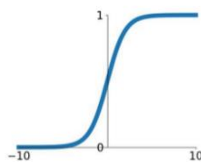
As we can observe, were we to use a linear activation function, the output we'd get would simply be a polynomial of first degree in regards to the input values. And while this example was rather small-scale, that linearity would hold true regardless of the number of neurons and layers in the network. While a network comprised solely from linear equations would undoubtedly be computationally cheap, it would also have limited power and, consequently, limited performance especially in complex problems. As such, we deduce that activation functions are not only necessary, but it is a requirement for them to be non-linear.

What this all comes down to is the necessity for NNs to be able to learn and represent any arbitrary complex function which maps inputs to outputs. That ability is indeed achieved by using non-linear activation functions that allows NNs to be considered **Universal Function Approximators** [1].

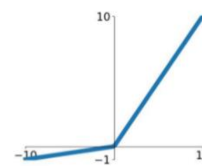
When it comes to choosing an activation function for a neuron there isn't a one-size-fits-all answer. The requirement is just a non-linear function that is also differentiable, which is needed to perform an optimization algorithm called **backpropagation** that will be explained later on.

The most frequently used ones are the following:

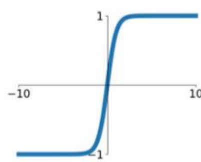
Sigmoid
$\sigma(x) = \frac{1}{1+e^{-x}}$



Leaky ReLU
$\max(0.1x, x)$

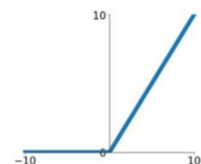


tanh
$\tanh(x)$



Maxout
$\max(w_1^T x + b_1, w_2^T x + b_2)$

ReLU
$\max(0, x)$



ELU
$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$

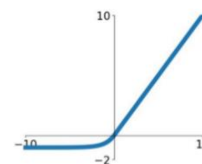


Figure 6: Frequently used Activation Functions

Indicatively, ReLU is often preferred especially in deep NNs due to its speed, while the once ubiquitous sigmoid has decreased in popularity. Still, each one has its advantages and its drawbacks and the optimal one will depend on the task at hand. Also note that an activation function is a characteristic of the node itself meaning that, if for whatever reason that is deemed desirable, it is possible to have neurons with different activation function within the same network.

1.2.1.3. Layers

Neural Networks are comprised of layers each of which is comprised by multiple neurons. Every NN has exactly one **input layer** through which the input data enters the network and exactly one **output layer** which serves as the result of the NN itself.

The number of neurons of the input layer is usually equal to the number of features of the input data, while that of the output layer depends on the task. For instance, for a NN serving as a binary classifier the output layer may contain a single neuron where a value below in the interval $[0,0.5)$ indicates classification in class A and, conversely, a value in $(0.5,1]$ indicates classification in class B. In n-class classification however, n output neurons may be preferred, with the value of each one signifying the probability of the input belonging to the nth class.

Between the input and the output there can exist 0 or more **hidden layers**. NNs that contain multiple hidden layers are referred to as **deep** – and hence the subfield of Machine Learning that employs their use is called **Deep Learning**. Networks without a single hidden layer are only capable of representing linear separable functions or decisions (for example, in binary classification that would include the “AND” and “OR” problems but not the “XOR” problem) which are obviously rather trivial and the sort of problems that are usually tackled with simpler means than NNs like Support Vector Machines (SVMs). Networks with exactly 1 hidden layer can approximate any function that contains a continuous mapping from one finite space to another while 2 layers suffice to represent an arbitrary decision boundary to arbitrary accuracy with rational activation functions and can approximate any smooth mapping to any accuracy [2].

Based on that final fact it would stand to reason that one would never need to use more than 2 hidden layers for any NN but, beside that mathematical fact, they are indeed

sometimes used as they have in some cases been observed to increase accuracy. Even then, 3 hidden layers is the greatest number of layers that is practically used and even that has been found to significantly increase the network's training time [3]. As such, if training time is a major factor the current application one should seek to minimize the number of network layers [4].

However, apart from choosing the number of the hidden layers themselves, another non-trivial task is to determine the correct number of neurons to use in each one. An initial thought, given that we're indifferent to the training time required, could be to have a multi-layer network with a very large number of neurons in each hidden layer in order to have our network adapt as much as possible for our input data. Besides the complexity and the obvious computational difficulties of that approach, it turns out that its results would actually also be suboptimal. To understand why we need to familiarize ourselves with a required property of NNs called **Generalization**: the network's ability to make decisions about previously unknown data based on what it has been trained on. When there is a great number of parameters (weights) that are modifiable during training, as would be the case of a NN with multiple hidden layers containing multiple neurons, the network tends to adjust to specific details of the training set, which inevitable leads to overfitting [5].

There actually are many rule-of-thumb methods for determining an acceptable number of neurons to use in the hidden layers, such as the following [2]:

- The number of hidden neurons should be between the size of the input layer and the size of the output layer.
- The number of hidden neurons should be 2/3 the size of the input layer, plus the size of the output layer.
- The number of hidden neurons should be less than twice the size of the input layer.

To conclude, the selection of a good enough architecture for any neural network will come down to trial and error. For most cases, it's suggested to strive to have a network be sufficiently small to adapt to the important and primary characteristics of the data used for its training, yet also able to ignore any secondary details that would hinder its ability to generalize.

1.2.2. Training risks

The concept of training has already been mentioned so far and it would thus be a good idea to further expand on it before proceeding. As mentioned in 1.1.1, the training process generally requires splitting our initial dataset comprised of labelled inputs into two sets, named *training* and *test*, with only the training set being initially provided to our model.

The caveat here is that we do not want our model to adapt to that training set too closely as that would cause **Overfitting**. Overfitting occurs when the trained model is adapted too closely – or even exactly – to the data points with which it was trained. Consequently, despite high measures of accuracy in the training set, its accuracy when used for new data will be significantly lower and worse than it would have been with an optimal model (with less training set accuracy).

The opposite phenomenon is called **Underfitting**, where the trained model is unable to adequately capture the underlying structure of the data. It can either be caused by lack of training or by using too simple of a model for the task at hand – that could be, for instance, trying to fit a linear model to non-linear data as depicted in Figure 7 (left).

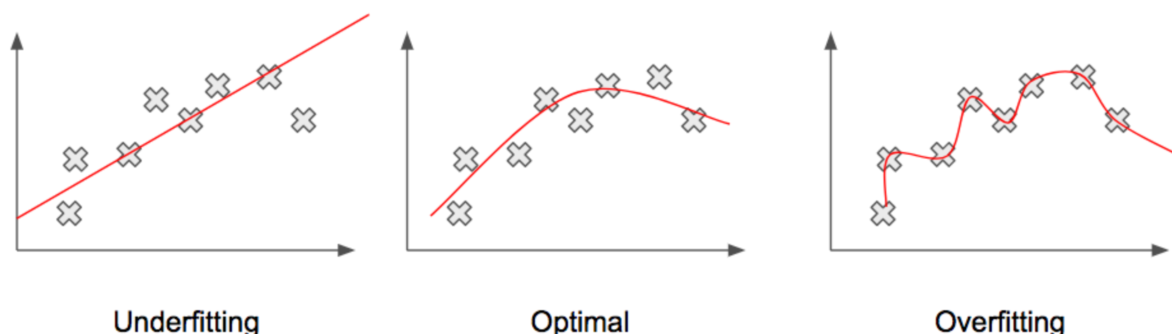


Figure 7: Examples of Underfitted (left), Optimal (center) and Overfitted (right) models

Overfitting on the other hand can occur when a model is too complicated or may simply be the result of overtraining. A visual representation of the latter can be seen in Figure 7 (right).

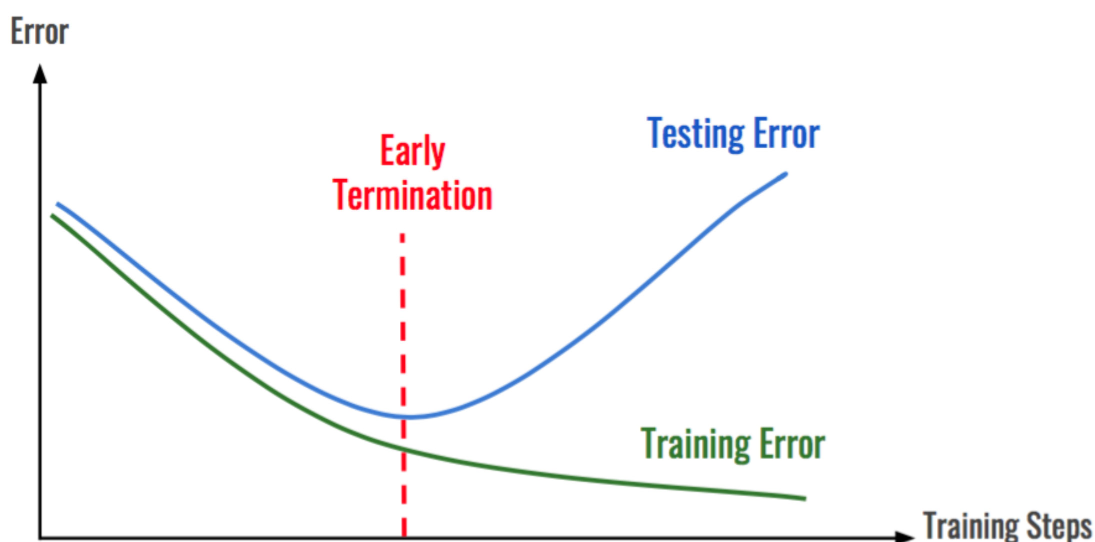


Figure 8: Testing and Training Errors plotted against Training Steps

In Figure 8 we see the calculated training and testing errors for some model as training progresses through time. Both errors keep decreasing until a certain point, after which the training error keeps decreasing (albeit more slowly) while the testing error starts increasing. The red dotted line signifies the "ideal" model, one that is sufficiently trained to generally make predictions about new data with an accuracy that isn't far worse than the training one.

In order to prevent overfitting, a technique named **Regularization** is used. Its exact purpose is to significantly reduce the variance of the model, without substantial increase in its bias. There are different approaches when it comes to regularization, one of which is **early stopping** of training, which was described above. Due to its ease of use, early stopping should almost always be used, unless there's a specified reason not to for any particular problem [6].

In NNs in particular, a regularization method that is frequently used is called **Dropout**. What that does is randomly deactivate neurons and/or connections during the training phase, as shown in Image 3.

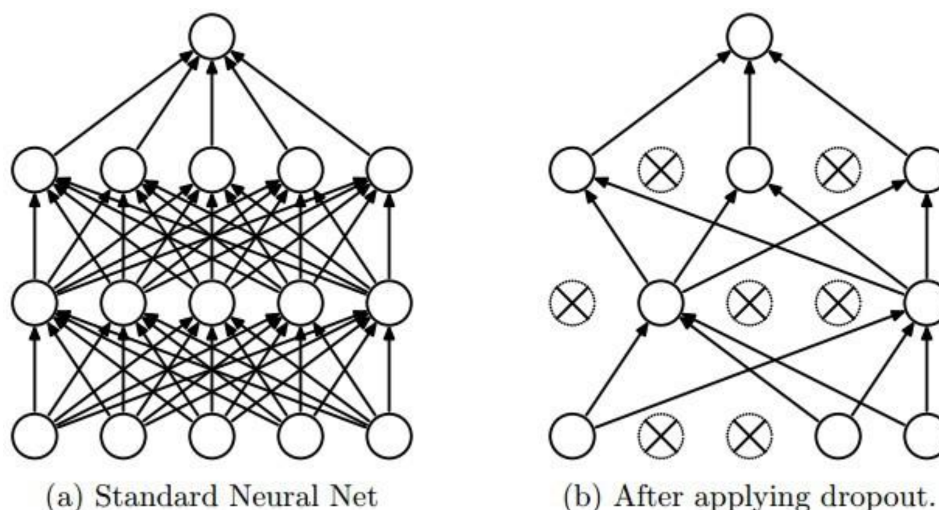


Image 3: An example of a NN before (left) and after (right) applying dropout

This forces the network to not rely on specific neurons or connections to extract specific features while training. Once the training is complete, all neurons and connections are restored to their original locations.

It is worth noting that using dropout layers negatively affects the overall training time: The number of epochs required to converge will be increased even though the training time of each epoch will be shortened [7].

1.2.3. Cost function

In the beginning, the weights in between our network's layers will be initialized to some arbitrary values. While research has been made to suggest strategies with which to initialize them [8], it is to be expected that our first outputs will diverge significantly from the optimal ones. So, in order to be able to quantify by how much the network's output was wrong we need to employ a **Cost Function** (also known as **Loss Function**). This function's utility is the knowledge that by minimizing its value we achieve higher prediction accuracy.

The most frequently used cost function is called **Mean Squared Error (MSE)** and is computed as:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2$$

where Y is the vector of observed predicted values and $\hat{Y}$ the vector of their desired ones.

Other widely used cost functions include **Mean Error (ME)**, **Mean Absolute Error (MAE)**, **Mean Percentage Error (MPE)**, **Mean Absolute Percentage Error (MAPE) and Cross-Entropy Loss**. Factors to consider when choosing a particular cost function over another include the activation function chosen for our network's neurons, whether we value a metric (such as precision or recall) more than just accuracy as well as how disproportionately harshly we want to cost larger errors. In any case, a good rule of thumb is that, unless we have a justification for preferring another one, MSE will suffice for most practical uses.

The exact way the cost function is used to iteratively improve the network is using the **backpropagation** algorithm with the help of **gradient descent**, both of which terms we'll examine next up.

1.2.4. Gradient Descent

As we've just mentioned, in order to maximize our model's performance, the requirement is to minimize the cost function. The way this is generally done in NNs is using an algorithm called **Gradient Descent (GD)**.

The mathematics of it are admittedly fairly complicated, but an intuitive explanation can easily be given by imagining the graph of any arbitrary function. Regardless of its complexity, there is a simple way to find a local minimum of it: Choose a random point of it as a starting point and then calculate the derivative (gradient) there. If it is negative then choose another point to the right of the function relative to the value of the gradient ("step") while if it's positive respectively choose a point that is a step to the left. We repeat this process until either the gradient is 0 – in which case we found our minimum – or until the gradient changes sign, in which case we decrease the step even more and keep moving towards the minimum. Even though this iterative algorithm provides no guarantee that the minimum we'll find will be a global one and in fact it most likely won't be, finding a local minimum has been proven to yield good enough results without being computationally daunting.

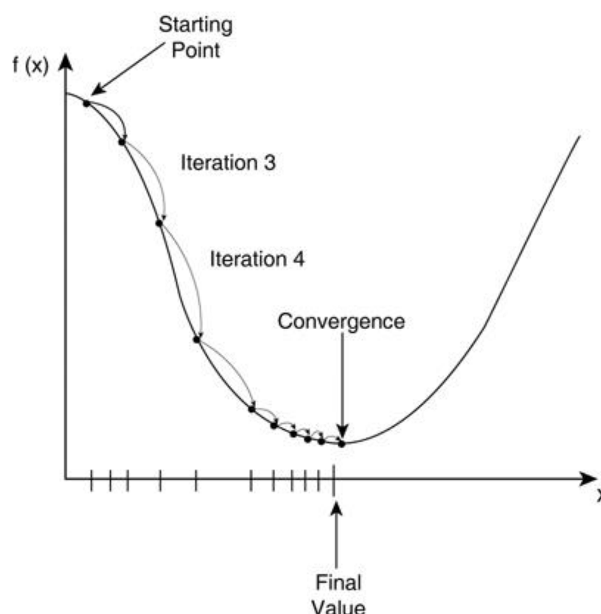


Figure 9: Simulation of Gradient Descent's convergence after multiple iterations

Gradient descent can be generalized to functions with several variables like most widely used cost functions have. In such cases however, we cannot visualize the direction towards the local minimum schematically. In these the result we get is a vector with equally many variables, the sign of each one showing whether that particular cost function variable should be increased or decreased and with its absolute value indicating how much should the variable change.

When using GD in a NN we use our entire dataset in each iteration – that is, in order for the network's weights to update once we need to run through every sample in our training set and repeat that for every iteration. What that entails is the fact that for a single improvement numerous calculations (possibly millions, depending on the training set's size) need to be done.

That is why in practice in most NNs today variations of the above are used, mostly **Stochastic Gradient Descent (SGD) and Minibatch Gradient Descent**. Their difference is that they instead use only one or a subset of the training set's data (often 50) at random respectively, pass them through the network, and update the network's

weight right away. Both those variations provide faster convergence than GD, but they are more likely to end in a worse minimized error function than it. Minibatch GD in particular is sometimes presented as the middle-ground between GD and SGD combining most of their advantages, but ultimately choosing the best Gradient Descent variation comes down to the training set's size, our training time limitations and our necessity for optimal accuracy.

As a last note, it may be useful to keep in mind that, unlike the deterministic GD, both SGD and Minibatch GD are, as the former's name suggests, stochastic. Consequently, it's fully possible and even likely that, due to their inherent randomness in choosing the training examples, these two will produce different local minima each time they're run. If precise consistency in our results is required then GD should be preferred.

1.2.5. Backpropagation

The "natural" way of information flow in a NN that we've been describing so far has only been forward. That is called **forward propagation** and describes exactly the path an input follows since entering the input layer until it exits through the output layer, passing any and all hidden layers in between in order.

Backpropagation (or **Backpropagation Through Time – BPTT**, to be precise), on the other hand, is an algorithm almost universally used in the training of ANNs for supervised learning. At the end of each **iteration** (that is, a full pass of a batch/part of the training set though the network) we can compare the model's outputs with the correct values. Given a cost function (as described in 1.2.3) we are able to quantify the divergence between actual and expected values and with the help of gradient descent (or one of its variants as detailed in 1.2.4) we can calculate how much the weights that resulted in those values should be changed and then change them accordingly. This entire process is repeated for as many **epochs** (passes over the entire dataset) as we want, but usual factors to take into account for when to stop it include:

- When the model's validation accuracy (or whatever metric is more important for the current problem) has not increased in X epochs (where often $X \in [5, 20]$)
- When the validation accuracy starts to drop (which can happen due to overfitting – see 1.2.2)
- Time and/or hardware constraints

Note that whenever we refer to accuracy above as a factor to stop training, we emphasize *validation* accuracy – that is of the testing set – as opposed to training accuracy. The reason for that is simply that backpropagation is designed to always keep increasing training accuracy, at least until it reaches a local minimum of the error function after which it will stagnate. However, as it was explained in 1.2.2, it is possible and even likely for the optimal testing accuracy to have been reached before then, at which point the training process should halt.

Without delving too deep into the math of backpropagation (which tends to quickly get fairly daunting even for small scale networks), the formula for an updated weight w_i' after backpropagation is $w_i' = w_i - a\frac{\partial Error}{\partial w_i}$, where w_i is the weight's last value and $\frac{\partial Error}{\partial w_i}$ is the partial derivative calculated with gradient descent.

The term $a \in (0,1]$ which we have not yet talked about is called **learning rate** and it is a constant value on which depends how quickly backpropagation will converge to a minimum. This constant need to be small enough so as to guarantee convergence, but the smaller it gets the slower that convergence is bound to happen. The optimal learning rate depends both on the cost function and the problem in question [9], with a common

practice being starting from a relatively large one and, if the training criterion diverges, gradually decrease the learning rate until no divergence is observed [10].

1.2.6. Types of Neural Networks

Most of what we've discussed so far, applies to all kinds of NNs. There are, however, some differences among different types of NNs that we'll expand on here.

1.2.6.1. Feed-forward Neural Networks

The most widely recognized kind of NN and the one discussed so far is that of **Feedforward Neural Network (FNN)**. In these, the information moves only in a single direction – forward – which is evident from the fact that the neurons of one layer connect only to neurons of the next layer. An FNN containing at least one hidden layer (in addition, of course, to the input and the output layers) is more specifically called a **Multilayer Perceptron (MLP)**. FNNs have no notion of order in time, and the only input they consider to provide an output is the one they are directly exposed to.

1.2.6.2. Convoluted Neural Networks

A special case of MLPs are **Convoluted Neural Networks (CNNs)**. The simplest way to describe their usefulness is that they use various regularization techniques to decrease the number of parameters that the network needs to process. This is very clearly understood in an example for the most frequent use case of CNNs, image recognition: Given a $48x48$ image, where each pixel has 3 separate values of interest (Red-Green-Blue), we get $48 \cdot 48 \cdot 3 = 6912$ of parameters for the input layer which is not necessarily unreasonable. However, for a higher resolution image, say, $1000x1000$ the number of input parameters rises to $1000 \cdot 1000 \cdot 3 = 3 \cdot 10^6$ which is certainly not computationally manageable. This is where various filters and specific layers are used in CNNs to effectively group together many of those parameters before the actual training on them begins. Other uses of CNNs include image classification as well as video recognition.

While we won't delve deeper into the above since we won't be using them for our stock market prediction problem, studying them provides a clearer understanding of NNs in general and it is useful to juxtapose them with the ones that we will encounter later on.

1.2.6.3. Recurrent Neural Networks

Recurrent Neural Networks (RNNs) expand the functionality of MLPs in that they allow the output of nodes to be fed as input to these nodes themselves or even ones in previous layers. In other words, the output of the hidden layers for any input point is directed, apart from the output layer, again in the hidden layer for the next input point in order – that is called the **hidden state**. The necessity for that functionality stems from an underlying assumption in traditional NNs, that all inputs (and, by extension, outputs) are independent of each other. While this is indeed the case in a plethora of problems, there are some like series prediction where any output depends on previous ones. In general, whenever there is a sequence of data that is sequentially inputted in the network and the order of it is important, that is when RNNs are most useful. A common use case is speech recognition, where in order to predict the next word in a sentence the previous words are required and there's a need for our NN to remember them. Stock prices are just another example of such timeseries.

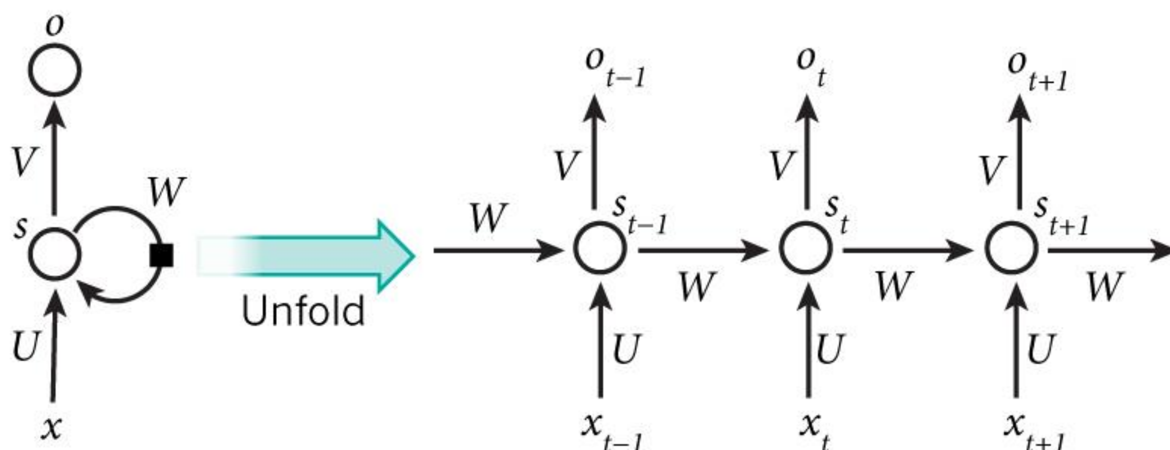


Image 4: A Recurrent Neural Network, folded (left) and unfolded (right)

An RNN is often depicted like on the left of Image 4 for simplicity's sake, while on the right it can be seen *unfolded* (or *unrolled*) into a full network. What is meant by that is that, say, for a sequence of *K* words, the network would be unfolded into a K-layer NN, with each layer corresponding to a single word. Let's further clarify the notation of this image:

- *W, V* and *U* are various, possibly different between them, weights. It's worth noting that those weights are the same across all steps, reflecting that each layer performs the exact same task, just with different inputs.
- x_t is the input at the time step t – which would be a word in our example.
- s_t is the hidden state at the time step t which is calculated using the formula $s_t = f(Ux_t + Ws_{t-1})$ where f is an activation function. s_{-1} which is required to calculate the following hidden states is typically initialized to 0.
- o_t is the output at time step t, which is what we would expect to be the network's prediction for the next word in a sentence, having seen the previous $t-1$ words. If we don't care about such partial outputs and we merely require, for instance, a prediction at the end of the sentence then all those outputs but the last could be omitted. [11]

A simple extension of RNNs is that of **bidirectional RNNs** (Image 5). These are effectively two conventional RNNs stacked on top of each other and are based on the intuition that for certain problems, mainly revolving around text data due to grammatical and syntactical rules, require knowledge of future states to make better predictions for the current one. While they're significantly useful for these problems, bidirectional RNNs cannot be used for predictions in timeseries-related tasks – since we obviously cannot base our predictions for the present time on future data – and as such we will not be exploring them further.

Image 5: A bidirectional RNN

Let's now discuss the problems that RNNs face. A major one which is also commonly found in other deep NNs is known as the **vanishing gradient problem.** Vanishing gradients make it difficult for the model to learn long-term dependencies. We'll again illustrate this through an example, where an RNN is given this sentence:

The brown and black dog, which was playing with the cat, was a german shepherd.

(x_2) (x_4) (x_5) (x_{14}) (x_{15})

In this, were the RNN try to predict the last two words "german" and "shepherd", it would need to take into account the inputs "brown", "black, and "dog" which are the nouns and adjectives that describe a german shepherd. However, the word "brown" is quite far from the word "shepherd" and to calculate the backpropagation error of the "shepherd" back to "brown" we need to calculate the partial derivative $\frac{\partial h_{15}}{\partial h_2}$ which, due to the chain rule, can be written as $\frac{\partial h_{15}}{\partial h_2} = \frac{\partial h_{15}}{\partial h_{14}} \frac{\partial h_{14}}{\partial h_{13}} \ldots \frac{\partial h_2}{\partial h_1}$. The problem lies in that these chains of gradients, when their values are lesser than 1, can cause the loss from "shepherd" with respect to "brown" to approach 0, thereby "vanishing". This makes it difficult for the weights to take into account words that occur at the start of a long sequence. As a result, during forward propagation the word "brown" may not have any effect in the prediction of "shepherd" because the weights weren't updated due to the vanishing gradient [12].

A similar and, in a way, "opposite" problem to the above that can also arise in RNNs is that of **exploding gradients**, which occurs by repeatedly multiplying gradients with values greater than 1. Exploding gradients can make learning unstable that may not be able to learn from the training data or even results in NaN (Not A Number) weight values during training [13]. To some extent, this can be mitigated by gradient clipping, that is simply clipping the gradients if their norm exceeds a given threshold [14], [15].

Several approaches have been proposed to overcome both these problems, with some performance improvements in discrete cases [16]. Still, achieving that in traditional RNN architectures – especially in the case of the vanishing gradients – has proven largely difficult with conventional methods [17].

1.2.6.4. LSTMs

Long short-term memory (**LSTM**) is an RNN architecture introduced in 1997 by Hochreiter and Schmidhuber [18] (and later refined by many others) in order to provide a

viable alternative to those domains that traditional RNNs are lacking, by changing how outputs and the hidden state are computed.

The factor that has probably contributed the most in their popularity is that LSTMs specialize in so-called “**long-term dependencies**” – in a simple text recognition example that could mean having to predict a word based on information from paragraphs or even pages before it instead of a just few words before it in the same phrase. While RNNs are too supposed to be theoretically capable of tackling such problems, in practice this has been shown to be significantly challenging [19].

Let’s begin our dive into LSTMs by directly comparing them to RNNs (Image 6).

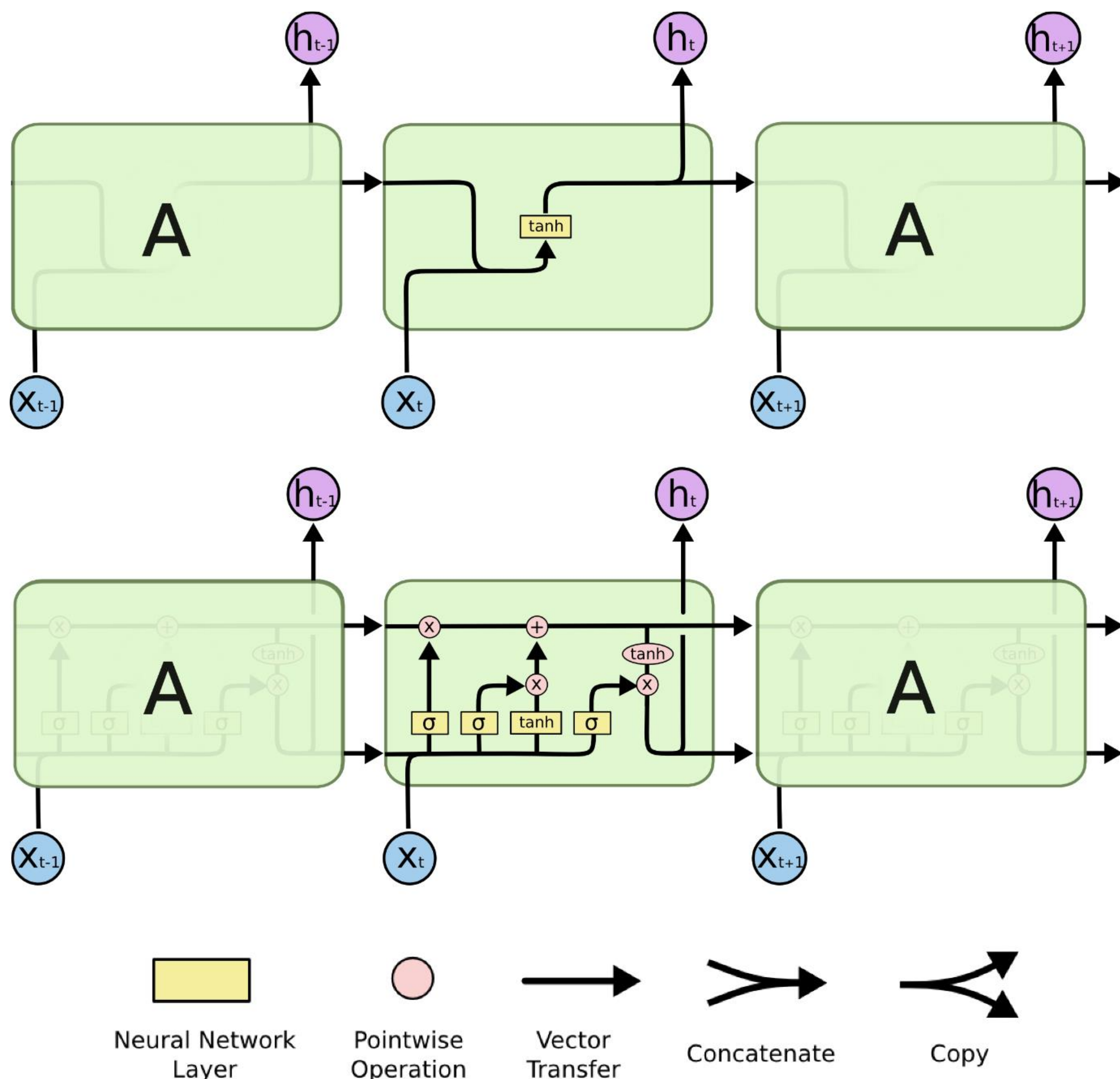


Image 6: The repeating module of a standard RNN (above) compared to that of an LSTM (below)

It is immediately obvious that the rationale of having a repeating module (cell) in a chainlike layout remains unchanged between the two. What does change – and significantly so – is the inner structure of each such module. A standard RNN’s cell contains just a single NN layer (such as a *tanh* layer) whereas in LSTM a single cell consists of four NN layers, interacting in a specific way. We will further analyze all of them next up by dissecting a single LSTM module.

A key characteristic of LSTMs that sets them apart from many other types of RNNs is that of the **cell state** (the horizontal line running through the top of the diagram) in addition to the module's output (hidden state). The cell state is what effectively acts as the "memory" of the network and inside each cell there are ways to add or remove information to it in a carefully regulated manner. It is also important that information can potentially flow unchanged through the module and that is what allows LSTMs to be free of the **vanishing gradient problem** discussed in 1.2.6.3.

It's now time to talk gates. Gates are structures composed out of a sigmoid NN layer and a pointwise multiplication operation that provide a way to optionally let information through. More specifically, the sigmoid layer outputs a number in the range $[0, 1]$ where 0 means "let nothing through" while 1 means "let everything through".

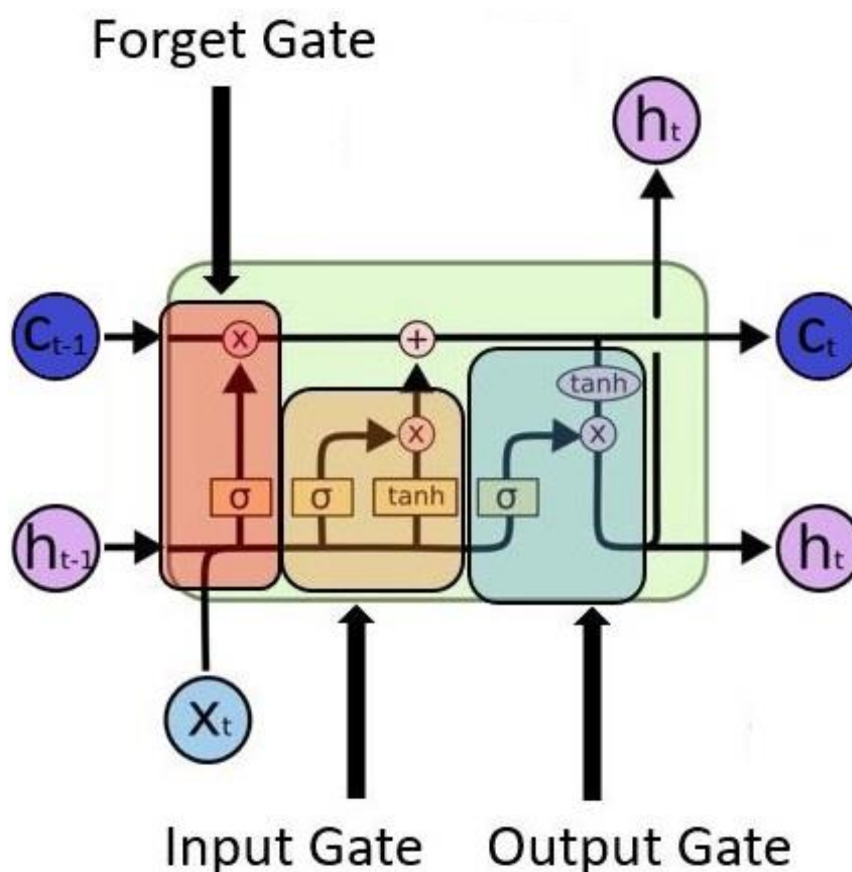


Image 7: Dissection of an LSTM cell

LSTM cells each use three of those gates, as annotated in Image 7. All three of them take as their input both the previous hidden state h_{t-1} and the current data input x_t. From left to right we have the following:

i. The **forget gate** which outputs how much of the new input should be added to the cell state. At first, the notion of a forget gate might seem redundant or even counter-productive since the tasks that LSTMs are used for explicitly require being able to remember previous information. It does however improve the prediction capabilities of the network, with a simple example of that being again text recognition: if the subject of a text excerpt changes after a full stop, it would be helpful to omit/forget information about the previous one going forward.

ii. The **input gate** which is further comprised of two parts. First, a sigmoid layer decides which values should be let through. Then, a second NN layer gives weight to the values that will be passed, signifying their level of importance (a tanh layer sets these weights in the range $[-1, 1]$).
iii. Last but not least, the **output gate**. The sigmoid layer, as before, decides which parts of the input should be included in the output and that result is then multiplied with the cell state after the latter is put through *tanh*. That is how the combined result of the new input and the cell state is finally outputted (h_t).

To recap, from beginning to end an LSTM memory cell performs 3 basic functions which correspond to the 3 gates we just described respectively:

i. Decide what information from the previous cell state is worth remembering and forget that which is deemed irrelevant.
ii. Selectively update the cell state using the new input.
iii. Decide which part of the cell state should be outputted as the new hidden state.

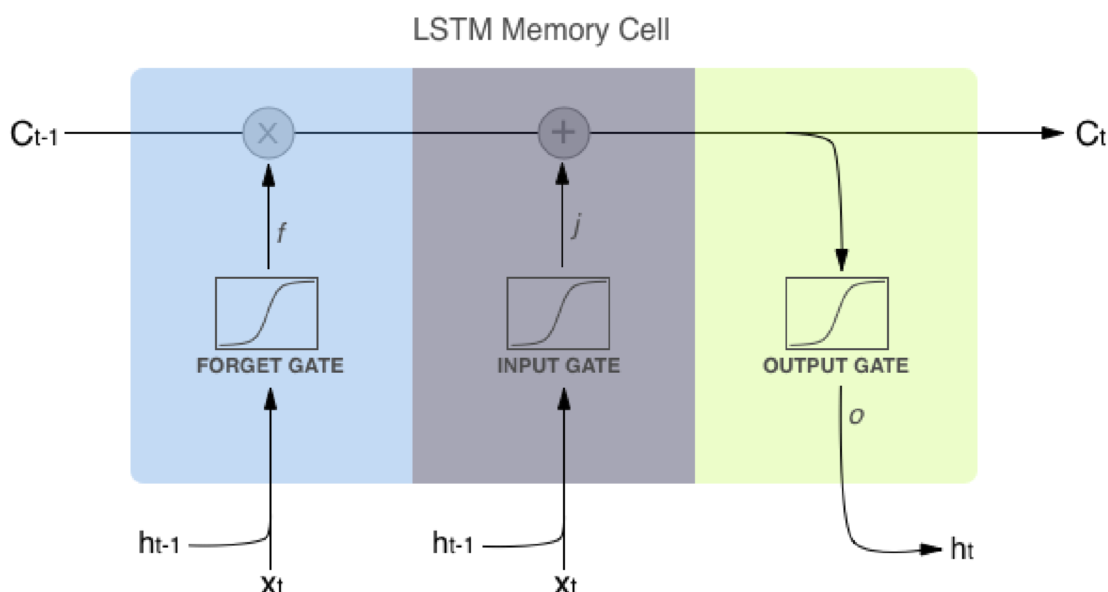


Image 8: Flow of information in an LSTM cell

It should be noted that what we've just described is the "vanilla" and most fundamental version of LSTMs, but this type of network often appears with slight differences in literature. Some variants add "peephole connections" to some or all of the gates that allow them to also get the cell state as an input, while others for instance couple the forget and input gates, so that the cell can only decide to forget a piece of information if it also inputs some new in its place.

A notable and more significant variation of LSTM is called **Gated Recurrent Unit (GRU)**, introduced just in 2014 by Cho et al [20]. As can be seen in Image 9, a GRU cell is simpler than an LSTM one since it doesn't only combine the forget and input gates into a single *update gate* but it also merges the cell state with the hidden state. While GRUs are currently rising in popularity and research into their tradeoffs is still ongoing, so far neither they nor any other commonly used LSTM variant seem to have exhibited any significant improvements compared to the vanilla one [21].

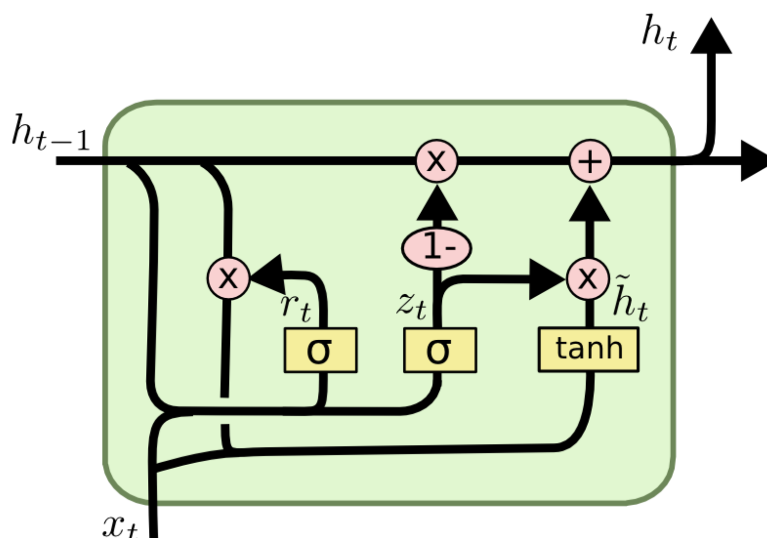


Image 9: A GRU cell

1.3. Stock Market Information

1.3.1. Stock

The **stock** (also known as "**capital stock**" or "**equity**") of a corporation is comprised of all of the shares into which ownership of the corporation is divided. A single **share** of the stock represents fractional ownership of the corporation in proportion to the total number of shares. This typically entitles the stockholder to that fraction of the company's earnings, proceeds from liquidation of assets, or voting power, often dividing these up in proportion to the amount of money each stockholder has invested. Not all stock is necessarily equal, as certain classes of stock may be issued for example without voting rights, with enhanced voting rights, or with a certain priority to receive profits or liquidation proceeds before or after other classes of shareholders.

The price of a stock fluctuates fundamentally due to the theory of supply and demand, even though there are many factors that influence the demand for a particular stock. The price of every stock changes on a daily basis, on days that the stock market operates (which mainly excludes weekends and some holidays). The data that is extracted at the end of a day for a particular stock are the values "Open", "Close", "High", "Low" which represent the price that the stock had at the start of the day, the price that it had at the end of the day, the highest price that it reached during that day and the lowest one respectively. Finally, there is also a value for "Volume", which represents the number of shares traded in that day.

Some people merely purchase particular stocks and hold onto them as a long-term investment, hoping that their price will increase in the distant future while others – referred to as **traders** – constantly buy and sell various stocks, aiming to profit by buying when the stock price is at its lowest and selling it at its highest before it starts declining again. Some are actually doing that as the prices change during the day – called Day Traders – who also tend to use shorter timeframes for the previously mentioned values (such as hour or even 5-minute live intervals) but most only use the values extracted after the stock market closes – called End-of-Day Traders. The latter ones are also the reason why for most practical purposes one can get sufficient results by solely focusing on the "Close" value of the stocks. Regardless of their preferred way of trading, traders are the core users of all kinds of software for analyzing and predicting the stock market.

1.3.2. Stock Market and Stock Exchanges

By **stock market** we don't mean any physical place but rather the aggregation of buyers and sellers of stocks. By contrast, a **stock exchange** is the institution or organization – accompanied by an actual venue – where traders are able to buy and sell shares of stocks, as well as bonds and other financial assets that are beyond our current interest.

The stock exchange of Greece is known as the **Athens Stock Exchange** (**ATHEX**). It was founded in 1876 and it is based in the capital city of Athens [22]. For reference, the two noteworthy stock exchanges in the U.S. are the **New York Stock Exchange** (**NYSE** – 1792) and the **NASDAQ** (1971) while one of the oldest stock exchanges is England's **London Stock Exchange** (1571).

1.3.3. Stock Market Index

A stock market index is a measurement of a section of the stock market and is computed from the prices of selected stocks (typically a weighted average). Three of the most major U.S. ones are Dow Jones, NASDAQ and S&P 500, while for ATHEX those would be the General Index and FTSE/XA. Since they fundamentally act as benchmarks, it is not feasible to directly buy shares of an index. Despite that, this nature of stock market indices makes them a frequent focus for analytical and machine learning purposes.

1.4. Stock Market Prediction

1.4.1. Random Walk Hypothesis

It has been suggested – and one could even argue it makes intuitive sense – that stock markets largely behave in random ways and are thus inherently impossible to predict. The term for that has been coined as "Random Walk Hypothesis" (RWH) and can be traced back to the 19th century. Extensive research has been made on this topic, analyzing long-spanning periods (and subperiods within them) and has long concluded that this claim does not hold true [23], [24].

1.4.2. Methods of Market Analysis

The first of the two major methods of market analysis is called **Technical Analysis** and it is a means of examining and predicting price movements in the financial markets, by using historical price charts and market statistics. It is based on the idea that if a trader can identify previous market patterns, they can form a fairly accurate prediction of future price trajectories. Those traders have a lot of dedicated software at their disposal like *Metastock*, *TradingView* and *TC2000* to name a few.

The counterpart to technical analysis is **Fundamental Analysis** which focuses on external events and influences, as well as financial statements and industry trends. Essentially, contrary to technical traders who derive all their information they need from charts with historical price data, fundamental traders look at factors outside of the price movements.

While both technical and fundamental analysis are currently used today – often together – by traders around the world, if we were to categorize our own attempts at predicting the stock market they would undoubtedly fall into the former category.

1.4.3. Related Work

1.4.3.1. Stock Data-related approaches

Predicting stock prices data with deep learning using past prices as input is a decently common task around the world, by Computer Science students and researchers alike. The abundance of such research served as a valuable resource both for references as well as ideas regarding which approach to take.

While limited attempts have been made with mere FFNs [25], it was soon evident that for this task more specialized types of NNs are preferable. There have been methodologies that were fairly successful using CNNs [26], but the majority of existing research on this subject utilized RNNs and specifically LSTMs which were what led the focus of the experimental part for this thesis in that direction. Most of them were pretty straight-forward, revolving around single-day predictions on NIFTY (an Indian stock index) [27], various American companies' stocks [28] as well as Chinese ones [29], predictions which we will too try to replicate for ATHEX's General Index.

Lastly, it bares mentioning that there exist stock data-related ML approaches that do not involve NNs, but rather SVMs or Random Forests [30]. Nevertheless, as was presented by another paper which compared such approaches [31], LSTMs still appear to be the superior choice for this task.

1.4.3.2. Predictions using NLP

A different approach than ours and one that does qualify as fundamental analysis, is that of using Natural Language Processing (NLP) to analyze relevant excerpts from current affairs regarding companies that participate in the stock market and then perform predictions based on the information gathered.

In one such paper [32], the researchers used news excerpts – short phrases, labelled "events" – from Reuters and Bloomberg and simulated a real stock trader, achieving an accuracy better than state-of-the art baseline methods by a factor of nearly 6%.

Similar techniques were employed in another paper [33], where a merged model was used combining stock price prediction using an RNN with textual – sentiment – analysis, again from news articles.

However, for this approach to be of any practical use, a plethora of data in necessary. In the first paper used 10 million events for 15 companies and even then, they found that for the lower fortune ranking companies for which they had fewer news available the prediction results were significantly worse compared to the rest. The latter one did not report significant improvements in accuracy, despite having a dataset in the hundreds of thousands of news articles.

Consequently, given the relative scarcity of detailed coverage on Greek companies in English and the limited state-of-the-art NLP algorithms capable of analyzing news sources in Greek, a conscious decision was made to limit our work to simpler "conventional" predictions using just the stocks' data.

2. PROPOSED APPROACH

2.1. Task Definition

Quite simply, we want to take a given stock or stock index and attempt to predict the future behavior of its price, though not necessarily its exact future value, by identifying patterns that appeared on its data in the past.

In accordance to what we described in 1.3.1, there is a handful of values that comprise a stock's daily data. As we explained there, "Close" is the one that better encapsulates the concept of "value" for the stock in that day and as such will be the target of our forecasting attempts.

As for the forecasting itself, there are two main ways to approach it. Those are:

a) Predicting the price of the following day, based on the data gathered up to the current one.
b) Make predictions in increments, which would mean performing predictions for a prespecified period of time, starting from a given day. This could be repeated multiple times throughout the test set.

Both of these will be explored our experiments, with the focus being particularly on the latter.

2.2. Tools and Technologies

The code development was done entirely in Python 3.6, using a prominent library specializing in NNs named Keras. It runs Google's TensorFlow behind the scenes (and can also utilize a variety of other backends like Theano) but is also designed so as to be significantly user-friendly without sacrificing the functionality these lower-level libraries provide, a fact that has led to its growing popularity among both researchers and large organizations. Keras allows building powerful NNs with a high level of abstraction, allowing one to specify layers, activation and loss functions, compile and finally train a model in a matter of a few lines. For these reasons, Keras with TensorFlow were selected for the implementation of this thesis' experiments. Other notable Python libraries that were used due to their well-known usefulness in a multitude of tasks such as file manipulation, scientific calculations and more include pandas, numpy and scikit-learn.

2.3. Dataset

Stocks' and stock indices' historical data, especially when going back more than a couple of years, is generally not too easy to find freely available. That was also the case for the Greek Stock Exchange, the official site of which only provides data for the last 30 days (https://www.athexgroup.gr/el/web/guest/index-historic/). Following some online research, the necessary data was found and taken from https://gr.investing.com/. For each of the stocks and indices there was at least 5 years' worth years of daily data ("Open", "Close", "High", "Low" and "Volume", starting from 2013 up to and including 2019), which resulted in csv files of around 1600 rows each.

The amount of data practically needed is a matter of discussion. In fact, some research indicates that, after a certain point, more data may actually increase the test error [34]. This research, however, did not contain any testing using LSTMs which are known to perform better with more data and, after some limited experimentation with our final model, it was evident that using more data consistently yielded better results than using

subsets of it. It is of course possible that if we had decades of data available then it may well have been preferable to omit some of them.

2.4. Focusing on Stock Indices

Intuition dictates that, even if one of the stocks that comprise it were to behave out of the ordinary, the underlying patterns of an index's movement would not be considerably affected. The apparent benefit of this fact is that it would render the movement of indices more predictable than that of individual stocks which in turn should entail greater prediction accuracy. For that reason, we will focus our predictions on ATHEX's General Index, although we will additionally perform some of the same tests on Coca-Cola's stock (EEE) which is ATHEX's stock with the higher market capitalization at the time of writing [35].

Note that whenever "stock" is being mentioned in later chapters it will refer to either *stock* or *stock index* unless otherwise specified, since they don't differ in any other way that is relevant in the context of this thesis.

2.5. Data Preprocessing

2.5.1. Data Cleaning

This is usually quite an important step in ML experiments, even though in our particular dataset no significant filtering or cleaning was required. In our case, the data for each stock was in csv format and it contained the following issues:

a) Unnecessary quotation marks around certain values.
b) Commas instead of periods as decimal separators (difference between Greek and English conventions) which prevented their parsing in code.
c) 'K' and 'M' to represent thousands and millions respectively in "Volume" values which again prohibited their parsing as numbers.
d) Visibly invalid data, such as unexpected zeroes or missing values.

The last one was luckily very rare and only appeared on a couple of "Volume" values which were replaced with the average volume of the surrounding few days, while the rest were easily corrected by finding and replacing them in-place with a simple Python script.

2.5.2. Normalization

There is one major reason for the necessity of data normalization and **feature scaling** in particular that we'll examine here and that is variable ranges. A simple example of that can be given by assuming a hypothetical ML task where a model used to classify a company's employees takes as an input two features about them, their age (roughly in the range $[20, 70]$) and their yearly salary (the values for which could vary from a few thousand euros to possibly hundreds of thousands). Given how ML algorithms work, the salary would be given much greater gravity, resulting in the age feature being effectively ignored. A similar problem could arise in our data, where we have for example "Close" stock prices (with values in the tens or hundreds) next to their "Volume" (ranging from tens of thousands to tens of millions). In order to prevent that, we rescale all features in one particular range (normally $[0, 1]$ or $[-1, 1]$) using feature scaling.

As is the case with many kinds of related tasks, a paper showed that the normalization method of stock data which is to be used in NNs for forecasting purposes can have a significant impact on prediction accuracy [36]. However, in that same paper it is concluded

that no one technique consistently outperforms the others and, for simplicity's sake, the common min-max rescaling was chosen for our experiments. Of course, there are cases that bear mentioning where more elaborate preprocessing techniques were used, such as in a paper [37] where the stocks' prices were treated as a wave function which allowed them to be transformed in a way that stripped them of any noise, before again feeding them to an LSTM.

While we're in the topic of normalization and regardless of the method chosen, it is worth mentioning a typical trap. In code, the dataset is often first loaded in its entirety and then split into the different sets, as described in the beginning. It's very easy – and fairly tempting, since it would also result in a couple of less lines of code – to perform the normalization of the data before splitting it. This though would be a clear mistake, since it effectively includes information from the validation and test sets, to which we are not yet supposed to have access, and would incorporate that into the training – that is sometimes referred to as **data leakage**. Instead, it is imperative that we first generate a normalization scale by fitting only the training set's data and then simply transform the rest of the data in that range (both of which can be easily achieved in Python using *sklearn.preprocessing.MinMaxScaler*).

2.6. Training Strategy

The way in which we train our NN, like any NN, is of paramount importance but when first tackling the challenge of training a NN using stock market data, one realizes that the decision for what exactly to provide to it as a training set is not trivial.

Our approach involves using data "windows" which will be promptly explained below:

2.6.1. Training window

In supervised learning – as is the case here – we use labeled training data, typically a vector of pair values: an input accompanied with the desired output for it. Here, the data that we have consists of float values (stock prices) for various consecutive dates – notice that our explanation here will assume our only having a single price feature for our stocks (e.g. "Close") but the exact same methodology can be also applied when having more features.

A single price in itself does not contain enough information to predict anything meaningful, and as such it's obvious that plainly providing every single data point (price) in our training set as an input wouldn't be of any use. Here's where the concept of the **training windows** needs to be introduced. Simply enough, a training window is but a fancy term referring to a vector of values as single input during training. To expand on this, we first make sure that our datapoints are in order (sorted by date from oldest to newest) and then we define a fixed size for our training window which we will be calling $trainingWindowSize$. Following that, we get the first (oldest) $trainingWindowSize$ stock prices and insert them into a vector which becomes our first training input. The second one includes an equal number of stock prices, but this time starting from $prices[1]$ and ending with $prices[trainingWindowSize]$, assuming that our stock prices exist in a (zero-indexed) array named $prices$ with size $pricesSize$. Similarly, the next training input will be the vector with the values from $prices[2]$ to $prices[trainingWindowSize + 1]$ and so on, until the last one which will contain the prices $prices[pricesSize - trainingWindowSize - 1] .. prices[pricesSize - 1]$.

After this process, we are left with an array of input vectors but, as we previously mentioned, an *output label* as it's called is of course needed to perform any sort of training, the structure of which will be our next consideration.

2.6.2. Training labels

Here, there are two main potential angles to examine, both of which will end up being used in some way later in our tests.

Everything depends on our assumption of what information the values inside a training window can provide. The intuition behind them is that some patterns that appeared within the prices in a single window are the ones that contributed to what follows (which is the output) and, as such, if they were to reappear in our test set, they should also result in the same output.

The first and simpler approach is to consider the single value that follows a window as its output. That would make $prices[trainingWindowSize]$ the output of the window with the values $prices[0]..prices[trainingWindowSize - 1]$ and in general for some x, a window with the values $prices[x]..prices[x + trainingWindowSize - 1]$ would have the single value $prices[x + trainingWindowSize]$ as its output. This way, we construct a vector of all those outputs which, along with the training window inputs, comprise our training set, as such:

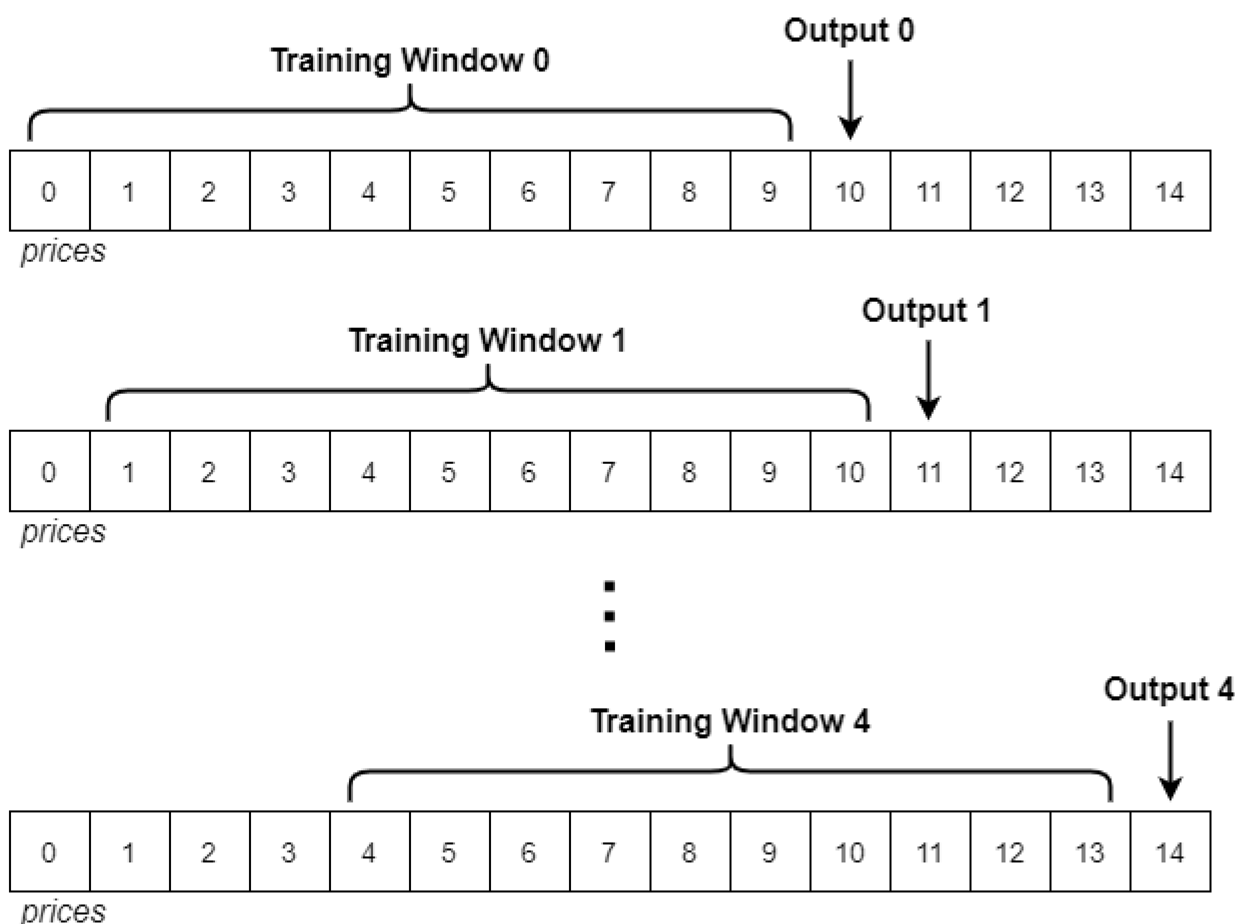


Image 10: Example of training windows with single outputs

The alternative, as one could guess, is to provide multiple values as a label for each training window input, instead of just one. This is known in timeseries prediction tasks as

"multi-step" forecasting (as opposed to the "single-step" above) and again involves constructing a window of values, this time those that trail the training window. Assuming a $predictionWindowSize$ of size greater than 1, here the output for the $prices[0]..prices[trainingWindowSize - 1]$ training window will be a prediction window containing the values $prices[trainingWindowSize]..prices[trainingWindowSize + predictionWindowSize - 1]$ and so on, until every training window is accompanied by a prediction window, the bundle of which will represent the training set.

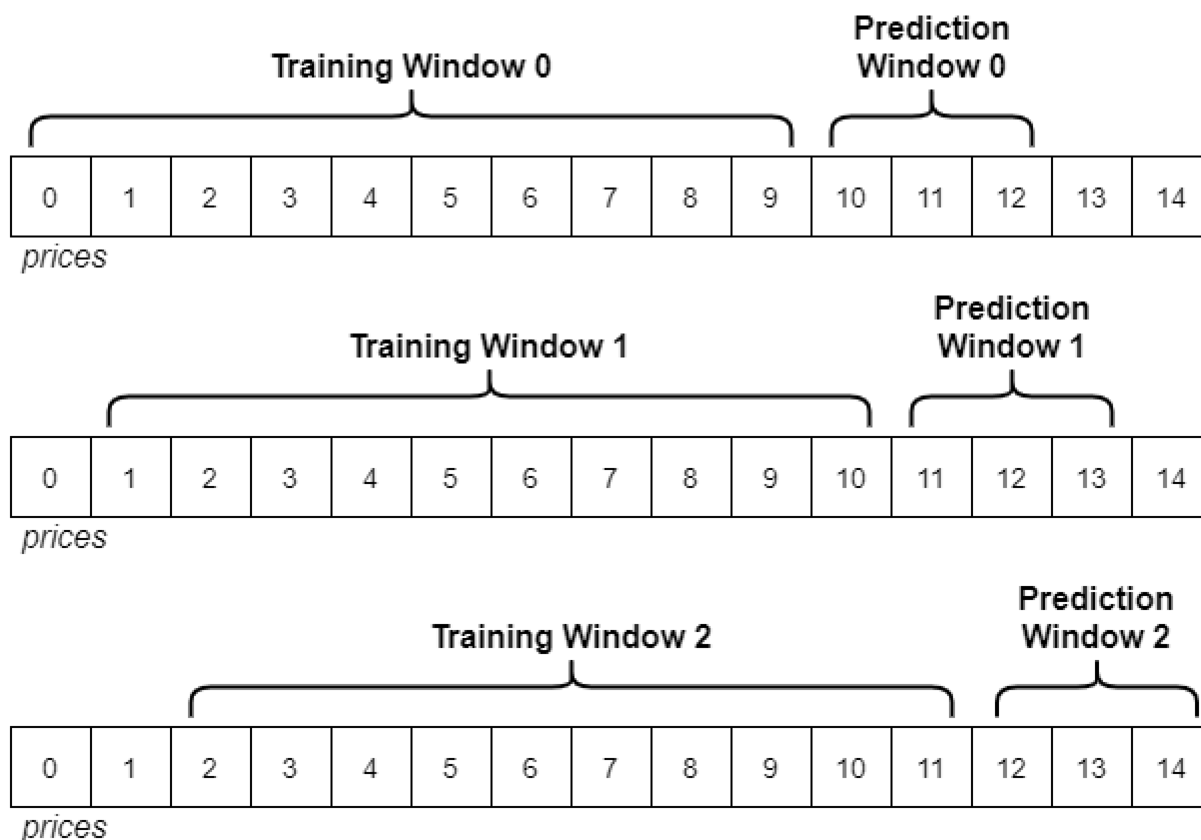


Image 11: Example of training windows with prediction window outputs

For both of these methods, it should be noted that the structure of the training inputs and outputs ought to be preserved throughout the validation and test sets as well (even using the same $predictionWindowSize$ for the latter method) since that will naturally also be the shape of the model's predictions. Furthermore, due to the need for values even after the last training window as depicted above, both methods result in fewer overall data point inputs than the training set's size (1 and $predictionWindowSize$ fewer respectively in quantity).

Lastly, it is important to underline that the training window's size is one of the most impactful design choices in our model. Too small of a window will be unable to pick up significant patterns and would result in our predictions being barely better than random ones. On the contrary, a window that is too large will not only provide us with less overall training data but, perhaps more importantly, will also miss subtle patterns that may merely appear in a handful of prices as those would be drowned down by the more overreaching ones within that window. As is often the case with such parameters, experimentation was needed here to arrive at a "just right" value for $trainingWindowSize$.

2.7. Model Design

The code that generates the model itself is the following:

```
1. self.model = Sequential()
2. self.model.add(LSTM(units=model_params["neurons"], return_sequences=True,
           input_shape=(params["training_window"], model_params["features_num"])))
3. self.model.add(Dropout(model_params["dropout_rate"]))
4. for _ in range(model_params["hidden_layers"]):
5.     self.model.add(LSTM(units=model_params["neurons"], return_sequences=True))
6.     self.model.add(Dropout(model_params["dropout_rate"]))
7. self.model.add(LSTM(units=model_params["neurons"]))
8. self.model.add(Dropout(model_params["dropout_rate"]))
9. self.model.add(Dense(units=model_params["dense_units"]),
10.        activation=model_params["activation"]))
11. self.model.compile(optimizer=model_params["optimizer"], loss=model_params["loss"])
```

The approach of using stacked LSTM layers was chosen, with a variable number of them, as evident by the $model_params["hidden_layers"]$ variable. The merit that multiple such layers provide is that complex relations between the data can be identified and learned, although there exists the danger of also doing so for characteristics of the training set alone that are not representative of the expected data in general (overfitting). After experimenting with up to 5 total LSTM layers, a number of 4 was deemed most appropriate for our later experiments. Each one of those is immediately followed by a Dropout layer whose value of 0.2 (that is 20% of neurons are to be randomly deactivated each time) is very commonly chosen as a good compromise between retaining model accuracy and preventing overfitting.

Another configurable part of our architecture was the variable $model_params["neurons"]$, which represents the number of input neurons each LSTM layer receives. A greater number of input neurons improves the capabilities of the model but only up to a certain degree and furthermore, as is also the case with more layers, leads to increased training time. They also have an inverse correlation between them, as in that models with fewer layers tend to require more neurons and vice versa. Solid values for this parameter were found to be around the value of 100.

Lastly, in the last line of the excerpt above, $optimizer$ and $loss$ parameters can be seen for which there was no reason to diverge from the usual choices for them, “adam” which is regarded as a fairly robust general purpose optimizer and “mse” (mean squared error) respectively.

2.8. Parameter tuning

Apart from the tuning of the parameters regarding our model’s architecture itself, there is also a number of other values, mainly regarding the training phase, that are customizable and have from little to paramount effect to our results. Namely:

- Training window size
- Prediction window size (when applicable)
- Choice of features
- Learning rate
- Batch size
- Number of training epochs

It is realistically impossible to manually test every possible combination of all reasonable values for each of these parameters, as is to give detailed reasoning for each and every one of them. It is well established in the field of ML that a lot of trial and error is required

regarding hyperparameters and that is how the choices were made here as well. Some will be mentioned later on when presenting the results, but generally:

- Training window size, being a particularly impactful parameter, was part of a lot of testing. Small values (around 30 or less) proved to be inefficient, and the best results which are presented below were produced using a training window of size 64.
- For simplicity but also since it would be the most relevant, initially only the “Close” price (feature) was used. The comparison between just “Close” and more features being used can be seen in the last part of 3.2.2.2.
- The learning rate, perhaps being the single most important hyperparameter to tune in a neural network [38], was also the topic of extensive testing. While the default value of 0.001 produced decent results, a value of 0.0005 was ultimately preferred.
- For the batch size, which is often a not too large power of 2, the value of 64 was chosen. This change, along with the learning rate one, of course also affected the number of training epochs required.

3. RESULTS

3.1. Metrics

Before presenting the results themselves, it is necessary to shortly explain the metrics on which we'll be evaluating the accuracy of our predictions.

"Accuracy" itself as a metric, which is possibly the most common metric used in ML in general, is in fact not applicable in regression problems, the category in which ours belongs. The reason for that being that we cannot compute a value for $\frac{correct\ outputs}{total\ outputs}$ or rather if we did it would always be equal to 0 since we are not trying to predict exact outputs but rather values that are as "close" to it as possible (according to our cost function). Instead, our focus will be on the loss – both during training and validation. The combination of these two also enables the prevention of overfitting through the use of early stopping, as shown in Figure 8 of 1.2.2.

Of course, we will also be seeing the plotted price predictions on top of the actual stock's price in a single graph, to be able to easily compare them. For that part, should one have knowledge of the financial field, they may be familiar with candlestick graphs to project stock data. However, the utility of those derives from their ability to simultaneously display "Low", "High", "Open" and "Close". Since only "Close" is of interest here, a simple line graph will be preferred instead.

Figure 10: A stock's price candlesticks graph (left) and its respective "Close" line graph (right)

3.2. Types of prediction

In all of the cases that we will be presenting, the figures depicting predictions were made on the test set of that stock's dataset and each such figure will be followed by a second one illustrating the validation loss along with the loss during the training of the model.

3.2.1. Single-step prediction

Let's start by a simple prediction, using just 2 LSTM layers and using the training window with single outputs, as described in 2.6.2 and similar to the ones made in [27], [28] and [29]. After training our model, we use it to predict the last 15% of our dataset (our test set) and the plotted results are as follows:

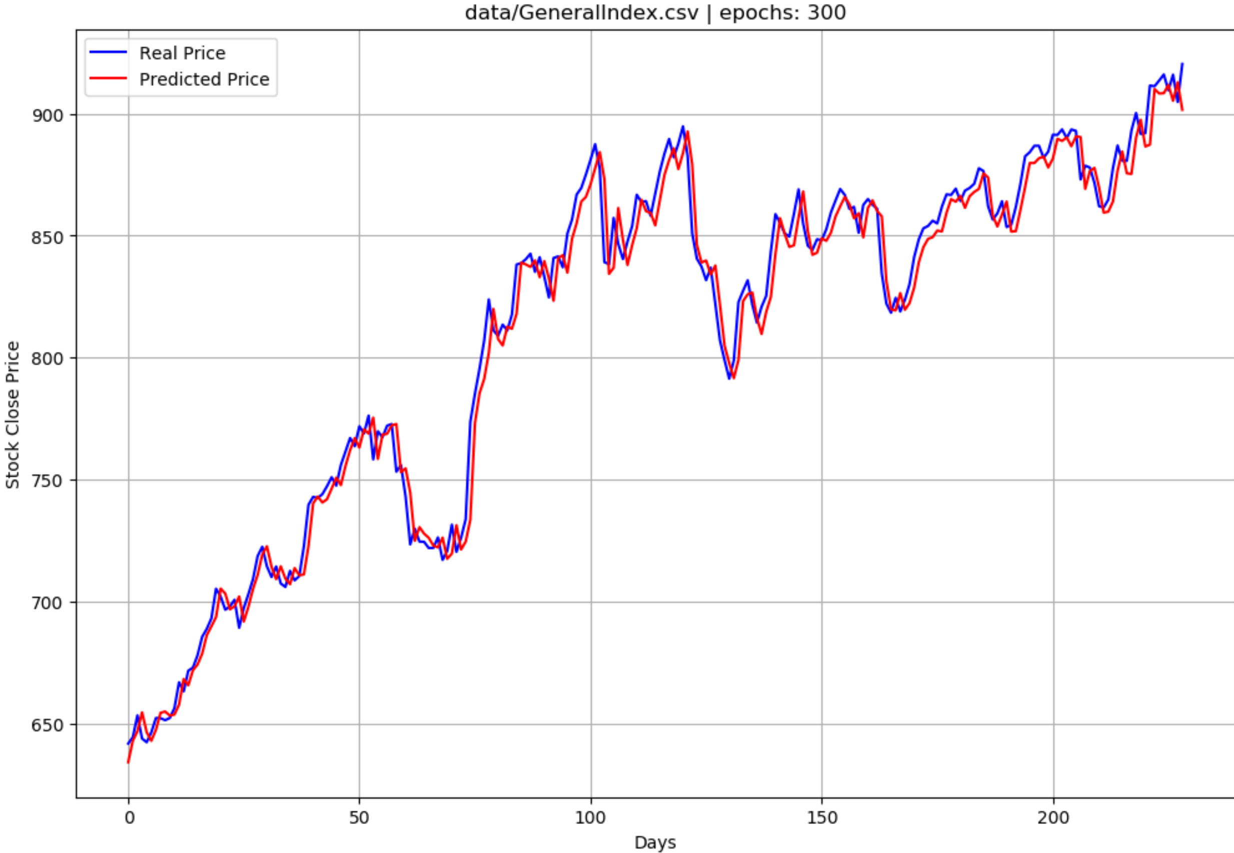


Figure 11: Single-step prediction of General Index

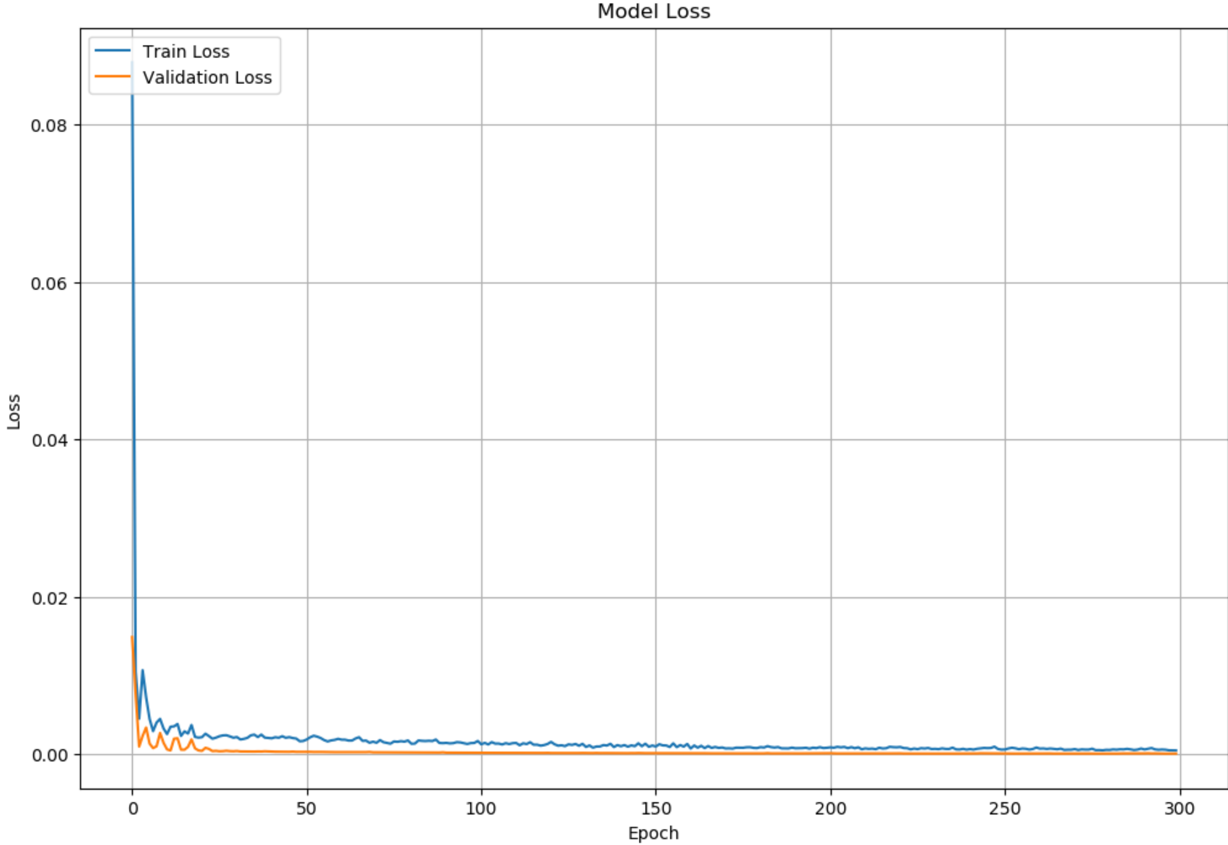


Figure 12: Model loss for Figure 11

We will be seeing graphs like Figure 12 depicting our model's loss later as well, similar also to the respective loss figures in [29], so let's focus on that figure first. The fact that both train and validation loss alike are steadily decreasing indicates a good fit and that our model is not overfitting to our training data. As for why the validation loss is (almost) constantly lower than the training one, that is something commonly explainable with two reasons. Firstly, the training loss for an epoch is calculated as the average of the losses over each batch of training data (that is, *during* the epoch) as opposed to the validation loss which is computed at the end of the epoch. Secondly and perhaps most notably, any regularization used (a case of which are the dropout layers of our model) are applied during training but not during validation and that fact also contributes to the plotted loss above.

So, given that we are not overfitting, how can our predictions be so close to the real prices as they seem at first glance? After all, predicting the exact future prices of stocks, let alone for long time periods is supposed to be a difficult and highly erroneous task. In order to understand what exactly is depicted in Figure 11, we need to inspect our graph more closely, for instance just the last 30 days:

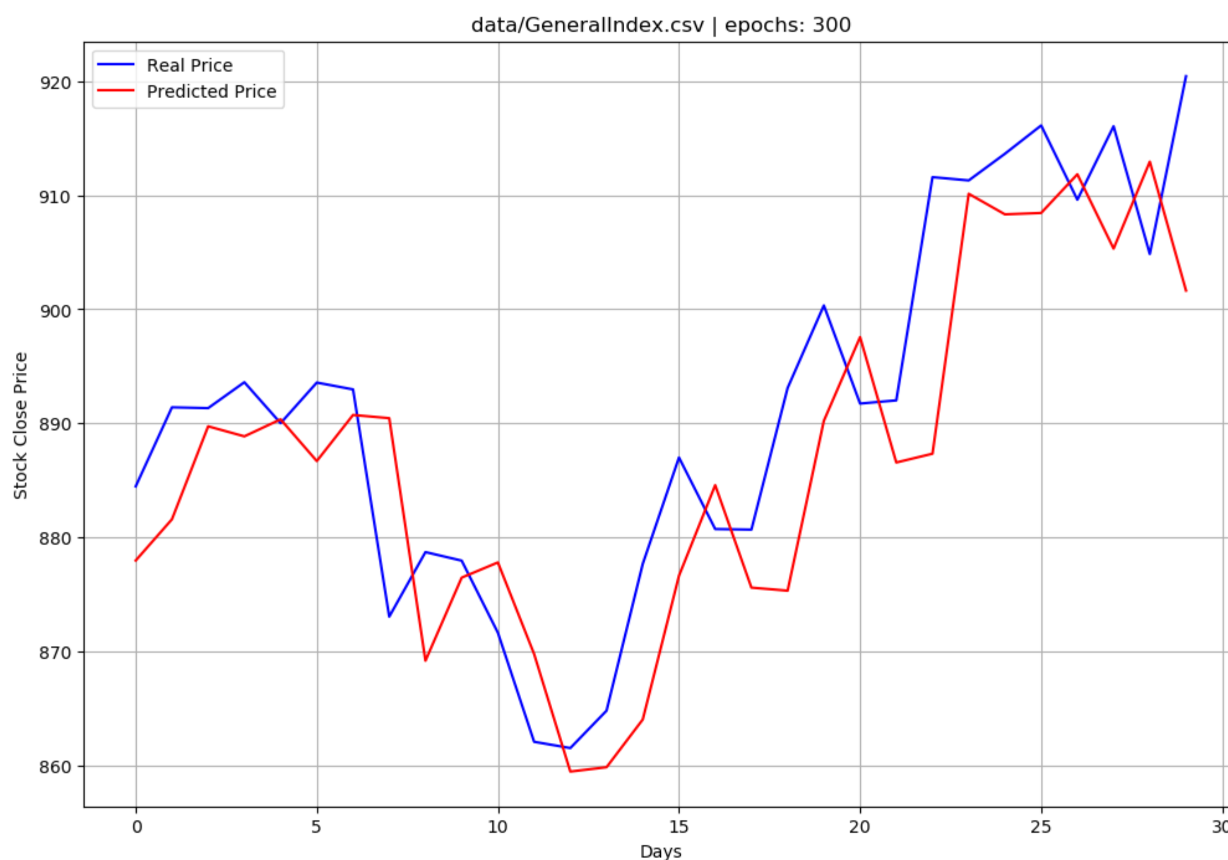


Figure 13: Last 30 days of Figure 11

Here, it is evident that our predicted price does not overlap with the real one, as one could be mistaken to believe when first seeing Figure 11. Instead, it is quite similar to a 1-day shifted version of the real price. Remember, the line of the "predicted price" above was not generated at once at the beginning of the test set, but is rather the product of multiple 1-day predictions. Essentially, this means that our model does indeed learn, but what it really learns is that a successful prediction of a day's price is generally very near to that of the previous day.

This of course does not entail that this kind of prediction lacks any practical use. One such could be anomaly detection, where if a day's price diverges too much from the

predicted one an automatic mechanism could be triggered, such as a notification system. Nonetheless, we would like to further experiment with different kinds of predictions, which we will be doing in the rest of this thesis.

3.2.2. Sequence predictions

A fundamental drawback of the single-step prediction is that it can only ever predict at most one day in the future after the last available in our data. And while this may be enough for some uses, we would like to examine how possible it is to perform decent predictions for periods of time greater than a day. This is what we will be trying next up, opting to make predictions for periods spanning some days, iteratively starting at various points in our test set. For the number of days for those sequences we will select 15 as a number that is large enough to prove our model's abilities without it being too large to be impossible to predict, but this is obviously easily parameterized.

3.2.2.1. Multi-step prediction

Moving away from single-step predictions, one could think to use the previously established training window, but use it instead to predict a number of future stock prices instead of just one. This is the approach described in the second half of 2.6.2, using the concept of prediction windows which are sometimes used for timeseries-related problems, although no existing research for their use in this context was found. The results of that can be seen below:

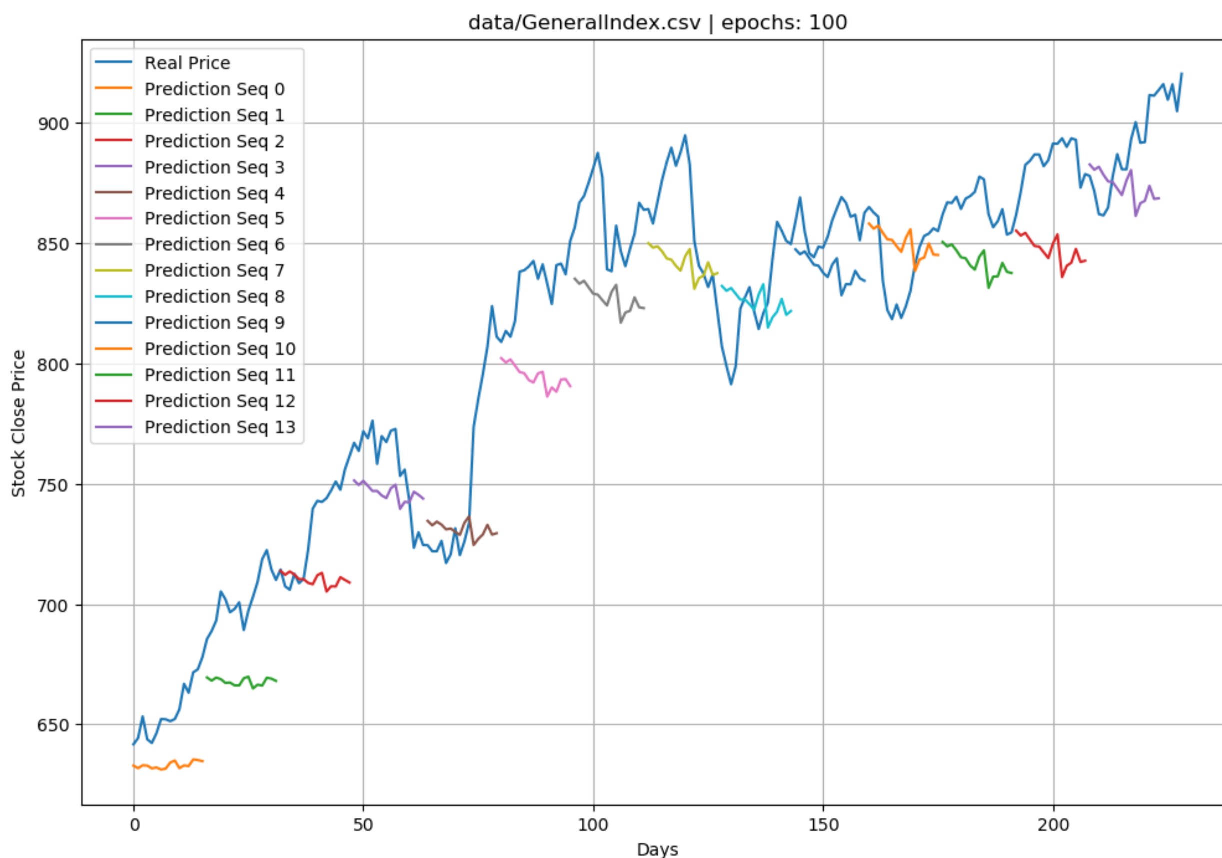


Figure 14: Multi-step prediction of General Index

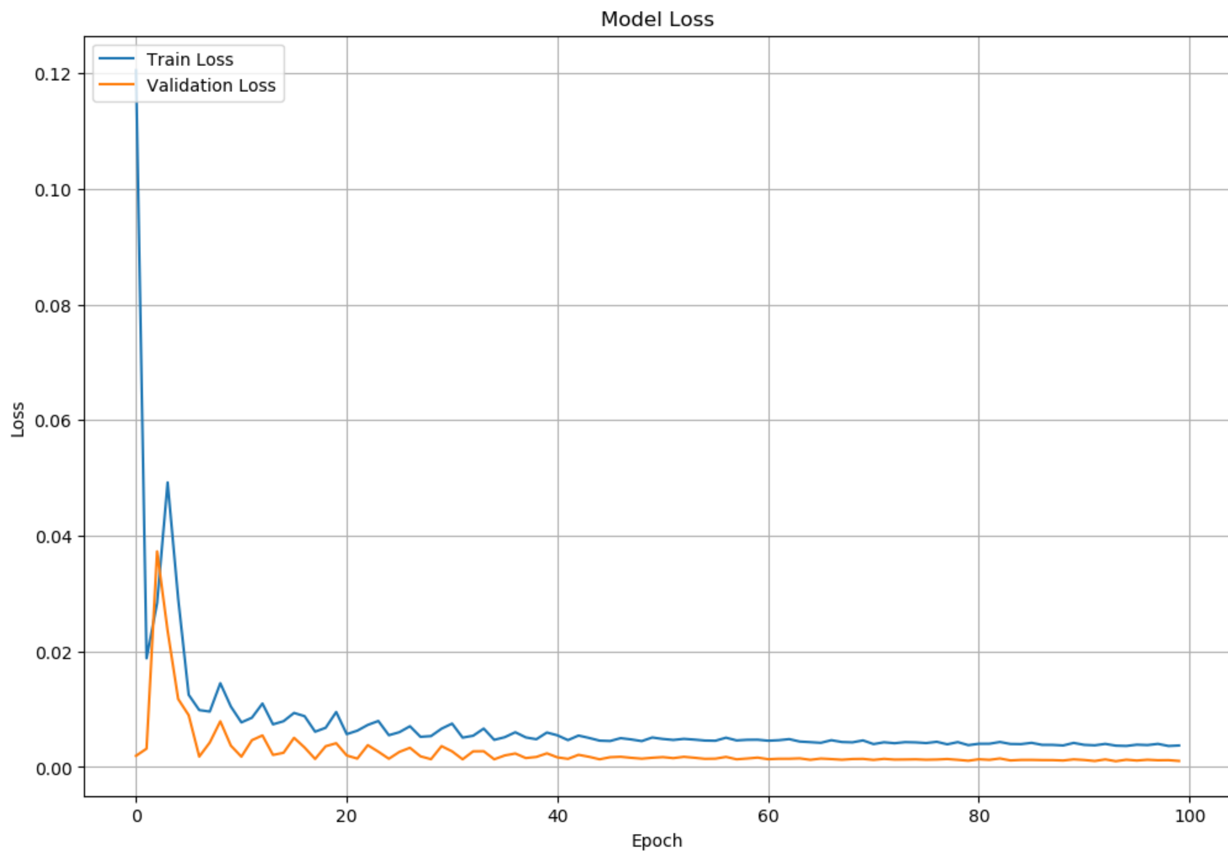


Figure 15: Model loss for Figure 14

What is immediately obvious here is that all predictions have very similar “shapes”. Evidently, during training the model learned an average trend that stock prices tend to follow in $predictionWindowSize$ steps and then applied it to every prediction it was asked to perform, resulting in the different predictions only differentiating in regards to the y-axis based on the input test data, while preserving an almost identical shape.

Results using this method were always similar to the above, irrespective of the number of epochs and various different values for other parameters. Given that such outputs differ significantly from what we were trying to accomplish, a different method to predict sequences of future prices had to be sought after.

3.2.2.2. Single-step sequence prediction

The next approach involved resorting back to the initial training methodology where a training window of multiple values was again used to predict a single value in the future. The difference with the single-step prediction in 3.2 is that we now employ a different prediction method and instead we now make predictions for a prespecified number of consecutive values, sequentially, with our trained model up to that point – the inspiration for which was an article with similar work on this subject [39]. This is done by gradually replacing values of the input window with predicted ones and performing new predictions using the old ones as part of the input. For example, if we wanted to predict sequences spanning 4 days, each one would be generated using our test set as follows:

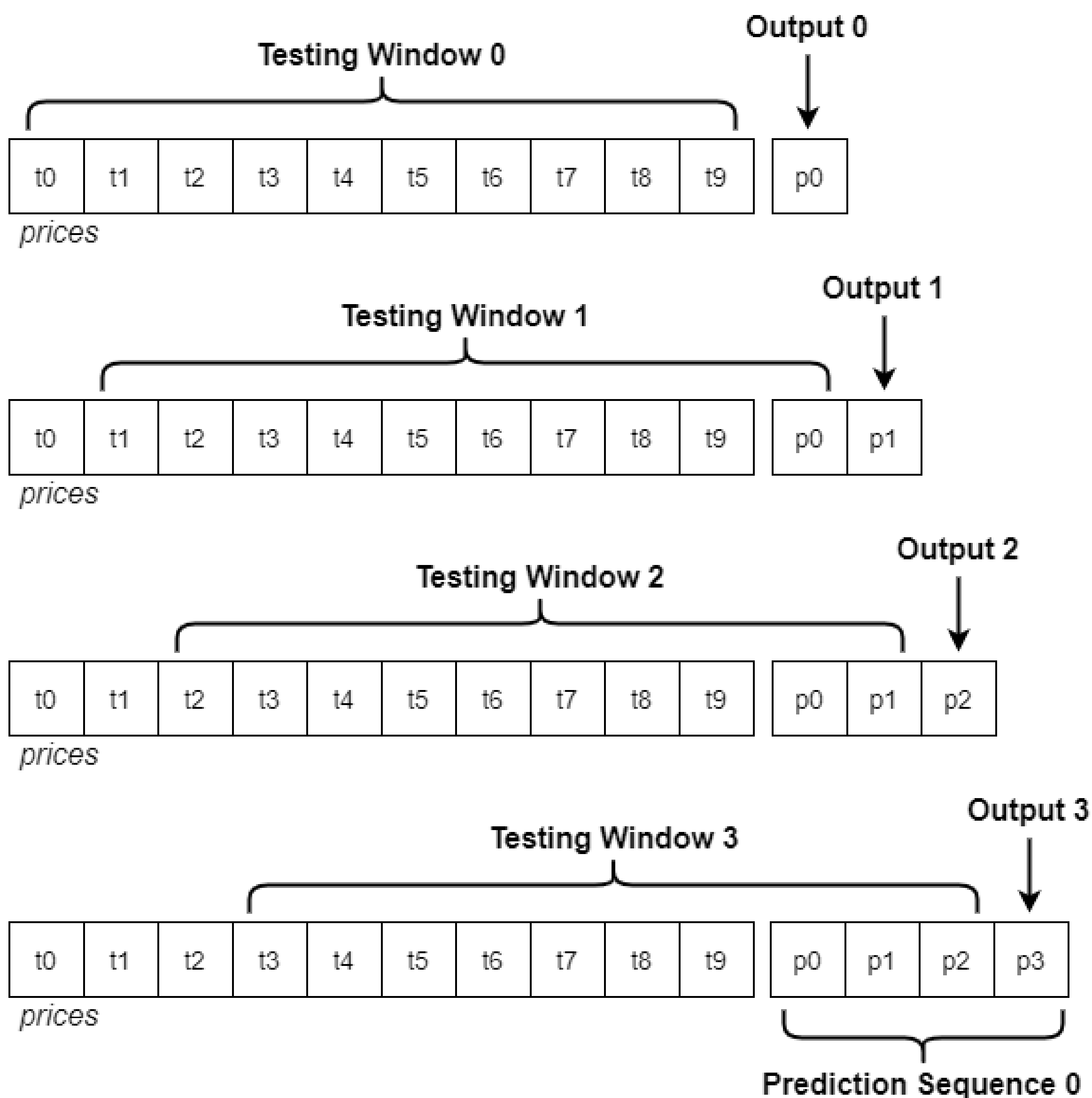


Image 12: Example of testing windows generating a prediction sequence

Here, were we to expect to see exact price predictions we would merely set ourselves up for disappointment as performing predictions based – even partly – on past predictions isn't likely to yield precise numerical results since prediction errors will inevitably accumulate. What we hope to achieve here is to predict a smooth trajectory of the price for the next 15 days, which should give us a general indication of whether the price is expected to rise, fall or remain about the same. So let's apply that technique again to the same stock we've been experimenting on so far, the General Index:

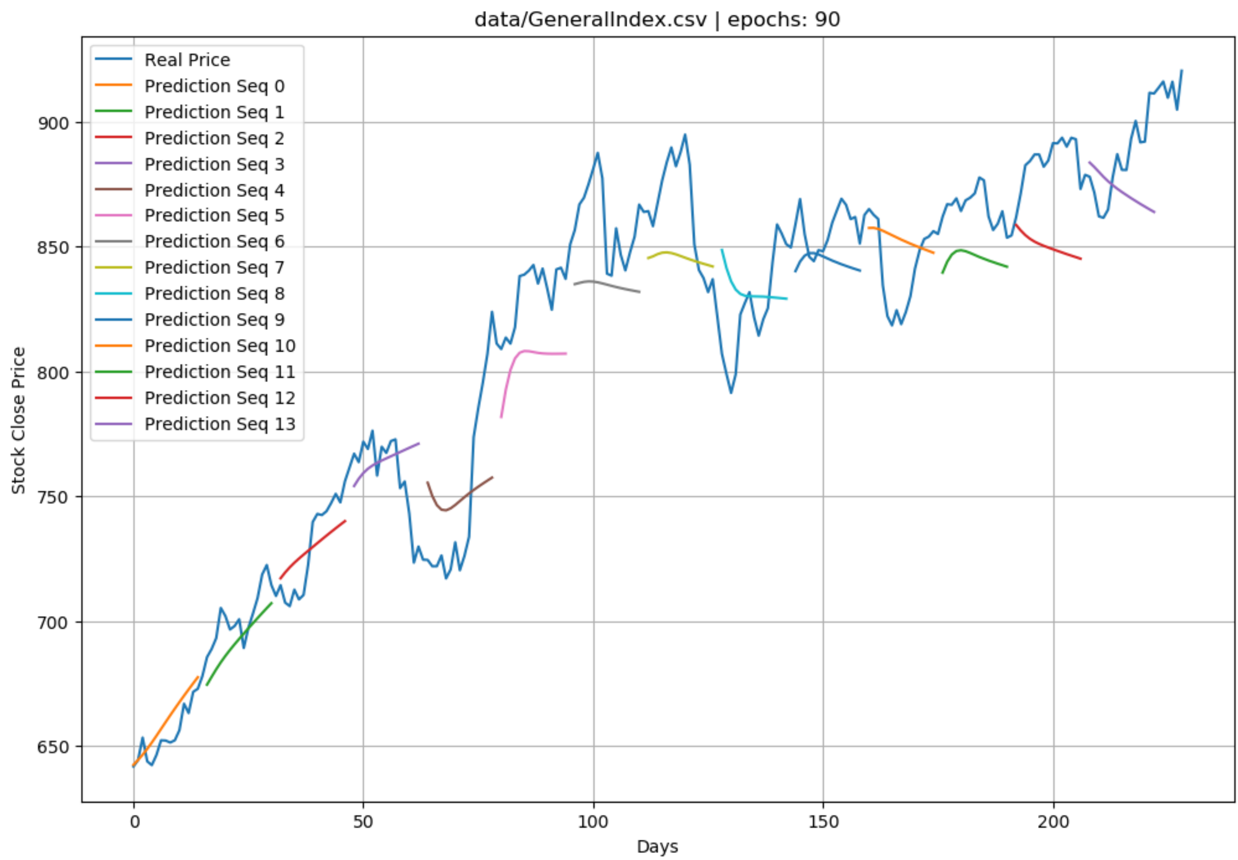


Figure 16: Prediction Sequences for General Index, after 90 epochs of training

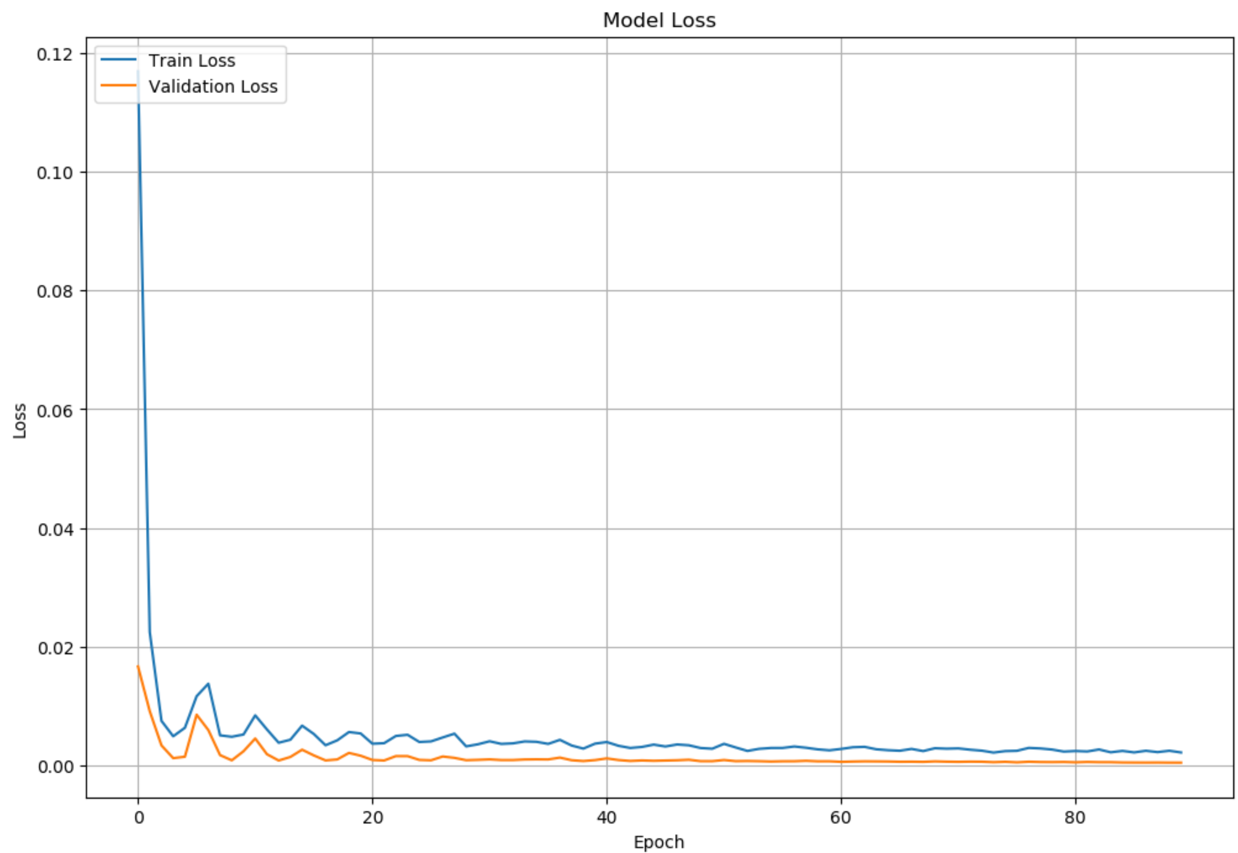


Figure 17: Model loss for Figure 16

As we can see, most prediction sequences turned out fairly accurate with few missing entirely like Seq 12 but others even capturing some curves in the prices' graph such as Seq 5 and 11.

Moreover, it is interesting to see how our model gradually learns, by examining the same prediction sequences but with less epochs of training:

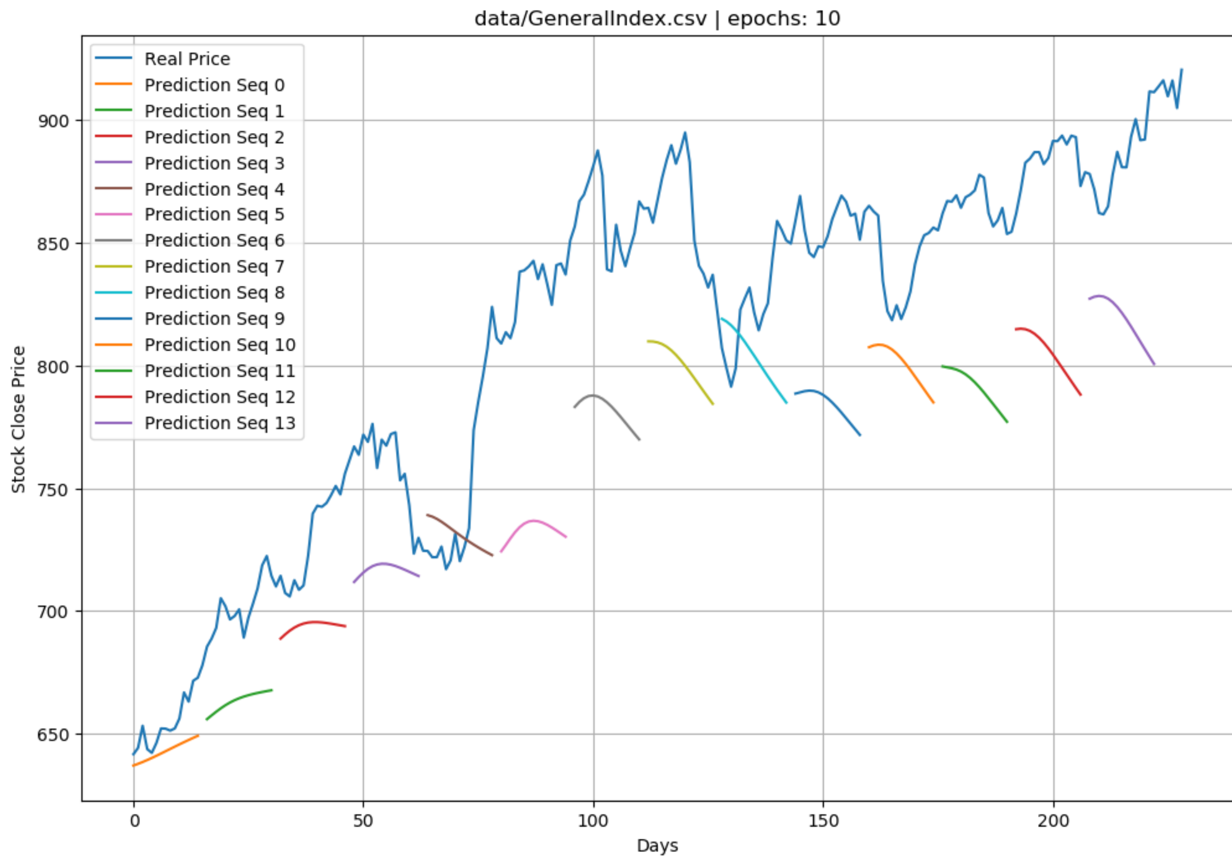


Figure 18: Prediction Sequences for General Index, after 10 epochs of training

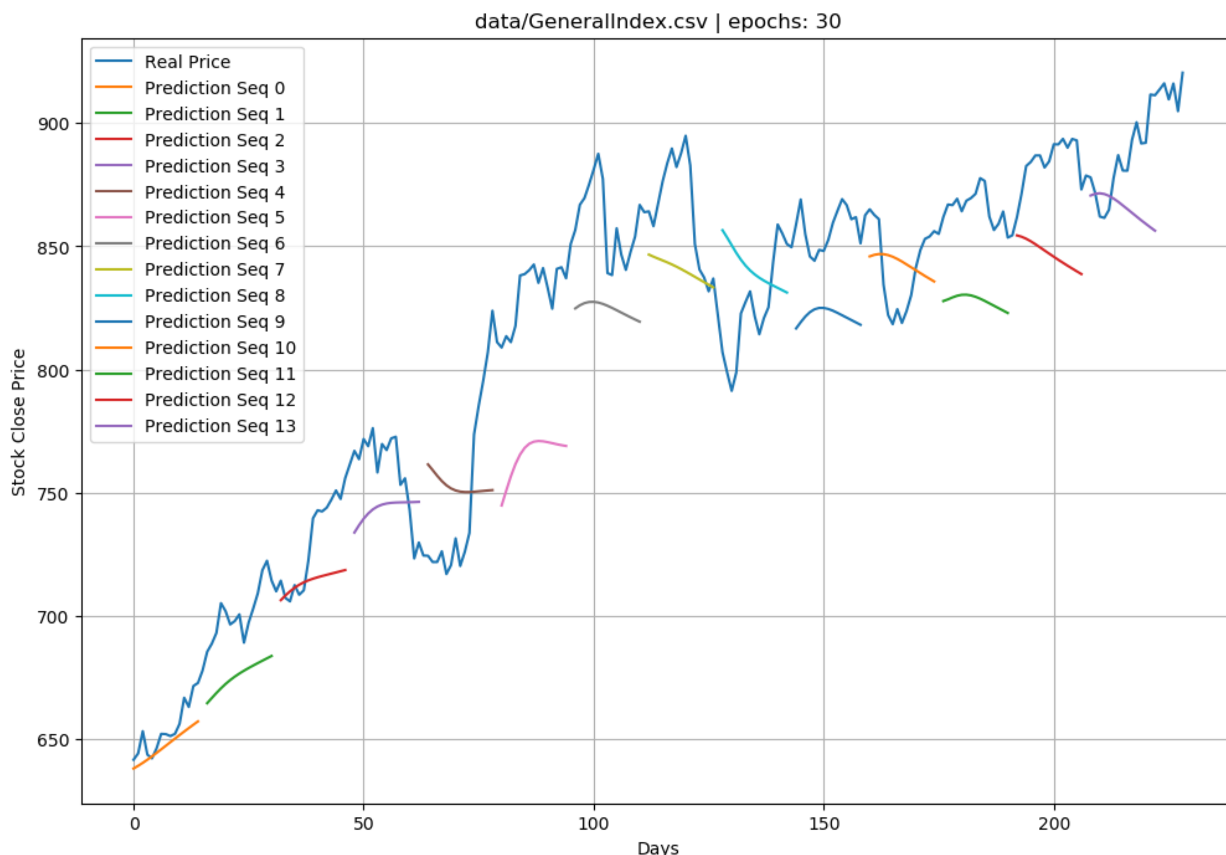


Figure 19: Prediction Sequences for General Index, after 30 epochs of training

Additionally, up until now we have only been training on "Close" prices, the same ones that we are trying to predict. Using the method of training we just used, let's also attempt to retrain out model, this time using more features, and juxtapose the results:

Table 1: Loss for General Index's predictions using different features

Features	Loss
["Close"]	0.00019
["Close", "Volume"]	0.00075
["Close", "Open", "Low", "High"]	0.00026
["Close", "Open", "Low", "High", "Volume"]	0.00043

The second column represents the lowest possible validation loss that we were able to get each time in a reasonable number of epochs, employing early stopping.

Before drawing any conclusions and in order to further test our model's capabilities, let's also try the same on Coca-Cola's stock (EEE) without altering any hyperparameters and again using the last 15% of the dataset as the test set.

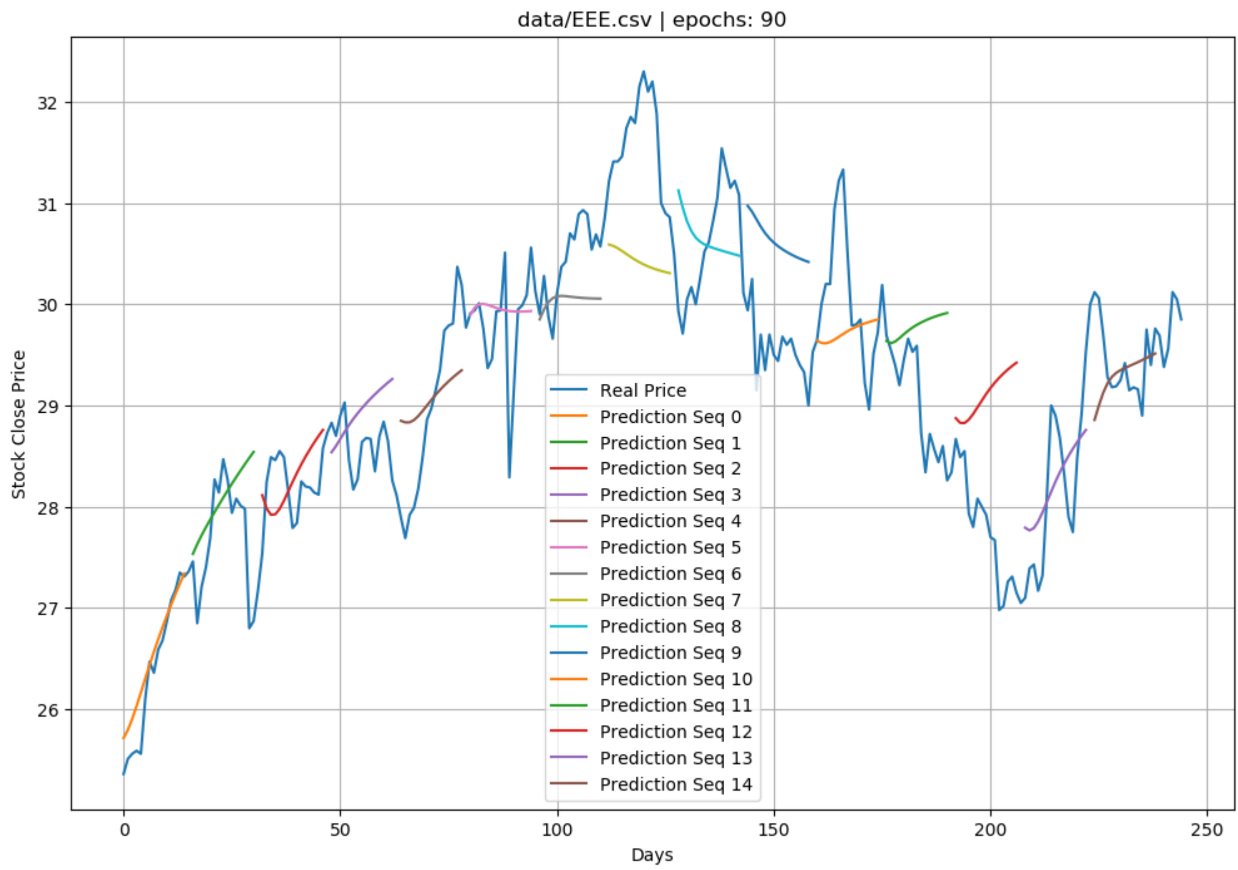


Figure 20: Prediction Sequences for EEE's stock

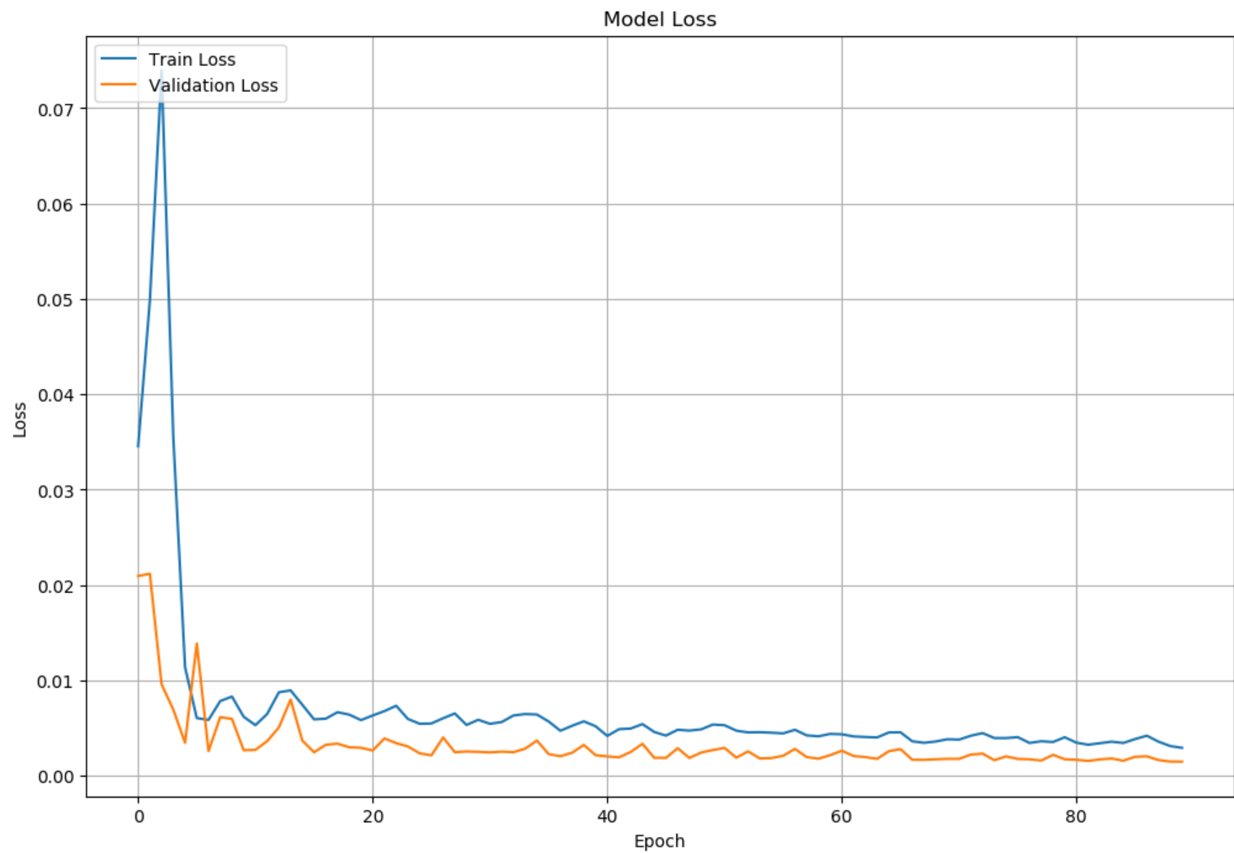


Figure 21: Model loss for Figure 20

Here we again see that many prediction sequences managed to capture the general movement of the stock in their time period, even though one could argue that this time more of them failed to visibly succeed in that.

Table 2: Loss for EEE's predictions using different features

Features	Loss
[“Close”]	0.0010
[“Close”, “Volume”]	0.0016
[“Close”, “Open”, “Low”, “High”]	0.0017
[“Close”, “Open”, “Low”, “High”, “Volume”]	0.0020

By comparing the two tables above we can observe that, using features other than “Close” in fact worsened our model’s loss. A possible explanation for that is that “Open”, “Low” and “High” are values that are approximate to “Close” and tend to function more like noise – and the effect of that noise was lesser in the index’s loss where fluctuations between those 4 values in any day are, naturally, minor compared to those of any single stock. On the other hand, “Volume” operates in an entirely different manner and any association between it and the “Close” price seemingly failed to be captured by our model.

Finally, regardless of the features used and despite having about the same number of data inputs to train, EEE’s loss was consistently found to be higher than that of the General Index. This further reinforces, even if not categorically, our initial assumption regarding the connection between stocks’ and stock indices’ predictability made in 2.4, that it is in fact comparatively easier to perform predictions on indices than on any particular stock.

4. CONCLUSION

4.1. Conclusions

It is reasonable to say that we succeeded in our goal which was to show that it is indeed feasible to make reasonable predictions regarding stock price movements using LSTM neural networks and even such that can span multiple days in the future.

It should, however, go without saying that betting money based on these predictions would be at the very least reckless. Stock trading should be done with at least some decent knowledge of stock market proceedings and there is no shortage of sophisticated analytical tools for that purpose, as described in 1.4.2. Of course, creating an actually profitable model was never the purpose of this thesis but rather exploring the forecasting capabilities of NNs which we achieved.

4.2. Future Work

The domain of stock market forecasting using deep learning is extremely vast and undoubtedly extends far beyond the scope of a single thesis.

Even in the model presented here and despite the efforts made for its optimization, as is often the case in ML it is entirely possible that a different combination of its parameters and/or a slightly different architecture could yield better results.

Future work could also involve training with more data, not only from stock exchanges around the world, but also from the significantly more volatile and ever-changing domain of cryptocurrencies which has entered the mainstream in the last couple of years.

Lastly, a more meaningful extension of what is presented here would be to train a similar model using data from multiple stocks and stock indices, thus attempting to create a unified predictor that could even perform predictions on stocks that it hasn't been trained on. Such a model, although it probably wouldn't outperform stock-specific ones, could prove valuable for predictions regarding newer stocks or generally in cases where limited past data is available.

ABBREVIATIONS – ACRONYMS

ANN	Artificial Neural Network
ATHEX	Athens Stock Exchange
CNN	Convoluted Neural Network
EEE	Coca-Cola’s stock
FNN	Feedforward Neural Network
GD	Gradient Descent
GRU	Gated Recurrent Unit
LSTM	Long Short-Term Memory
ML	Machine Learning
MLP	Multilayer Perceptron
MSE	Mean Squared Error
NLP	Natural Language Processing
NN	Neural Network
ReLU	Rectified Linear Unit
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
SVM	Support Vector Machine

REFERENCES

[1] Kurt Hornik, Maxwell Stinchcombe, Halbert White, "Multilayer Feedforward Networks are Universal Approximators", *Neural Networks*, vol. 2, pp. 359–366, 9 Mar. 1989.

[2] Jeff Heaton, "Feedforward Backpropagation Neural Networks" in *Introduction to Neural Networks* with Java, 2nd Edition, 2008, ch.5, pp. 128–131.

[3] Saurabh Karsoliya, "Approximating Number of Hidden layer neurons in Multiple Hidden Layer BPNN Architecture", *International Journal of Engineering Trends and Technology*, vol. 3, issue 6, 2012.

[4] Gaurang Panchal, Amit Ganatra, Y. P. Kosta, Devyani Panchal, "Behaviour Analysis of Multilayer Perceptrons with Multiple Hidden Neurons and Hidden Layers", *International Journal of Computer Theory and Engineering*, vol. 3, Apr 2011.

[5] Theodoridis Sergios and Koutroumbas Konstantinos, "Choosing the Network's Size" in *Pattern Recognition*, 2009, ch. 4.9, pp. 191–193.

[6] Ian Goodfellow, "Early Stopping" in *Deep Learning*, 2016, ch. 7.8, pp. 239–242.

[7] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting", *Journal of Machine Learning Research 15*, Jun 2014.

[8] Mercedes Fernández-Redondo and Carlos Hernández-Espinosa, "Weight Initialization Methods for Multilayer Feedforward", *European Symposium on Artificial Neural Networks*, Bruges (Belgium), Apr 2001.

[9] Leslie N. Smith, "Cyclical Learning Rates for Training Neural Networks", *2017 IEEE Winter Conference on Applications of Computer Vision (WACV), Santa Rosa, CA, 2017,* pp. 464–472.

[10] Yoshua Bengio, "Practical Recommendations for Gradient-Based Training of Deep Architectures" in *Neural Networks: Tricks of the Trade*, *Lecture Notes in Computer Science*, Sep. 2012, ch. 19, pp. 437–478.

[11] http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/

[12] https://medium.com/towards-artificial-intelligence/whirlwind-tour-of-rnns-a11effb7808f/

[13] Ian Goodfellow, "Optimization for Long-Term Dependencies" in *Deep Learning*, 2016, ch. 10.11, pp. 401–404.

[14] Yoav Goldberg, "Vanishing and Exploding Gradients" in *Neural Network Methods in Natural Language Processing*, 17 Apr 2017, ch. 5.2.4.

[15] Razvan Pascanu, Tomas Mikolov, Yoshua Bengio, "Understanding the exploding gradient problem", *arXiv:1211.5063v1 [cs.LG],* 21 Nov. 2012.

[16] Razvan Pascanu, Tomas Mikolov, Yoshua Bengio, "On the difficulty of training Recurrent Neural Networks", *ICML'13: Proceedings of the 30th International Conference on International Conference on Machine* Learning, vol. 28, pp. 1310–1318, Jun. 2013.

[17] Sepp Hochreiter, "The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions", *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 6, issue 2, Apr 1998.

[18] Sepp Hochreiter, Jurgen Schmidhuber, "Long Short-Term Memory", *Neural Computation*, vol. 9, issue 8*, 1735–1780*, 1997.

[19] Yoshua Begio, Patrice Simard, Paolo Frasconi, "Learning Long-Term Dependencies with Gradient Descent is Difficult", *IEEE Transactions on Neural Networks*, vol. 5, no. 2, Mar. 1994.

[20] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, Yoshua Bengio "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation", *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP),* pp. 1724–1734, Oct. 2014.

[21] Klaus Greff, Rupesh K. Srivastava, Jan Koutnik, Bas R. Steunebrink, Jurgen Schmidhuber, "LSTM: A Search Space Odyssey", *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, pp. 2222–2232, Jul. 2016.

[22] http://www.helex.gr/history/

[23] Andrew W. Lo, A. Craig MacKinlay, "Stock Market Prices Do Not Follow Random Walks: Evidence From A Simple Specification Test", *The Review of Financial Studies*, vol. 1, issue 1, Jan. 1988.

[24] Natividad Blasc, Cristina Del Rio, Rafael Santamaría, "The Random Walk Hypothesis in the Spanish Stock Market: 1980–1992", *Journal of Business Finance & Accounting*, vol. 24, Issue 5, Jun. 1997.

[25] F. W. Op't Landt, "Stock Price Prediction using Neural Networks", *Leiden University*, Master Thesis, 4 Aug. 1997.

[26] Sreelekshmy Selvin, Vinayakumar R, Gopalakrishnan E.A, Vijay Krishna Menon, Soman K.P, "Stock Price Prediction using LSTM, RNN and CNN-Sliding Window Model", *Procedia Computer Science 132*, pp. 1351–1362, Jan. 2018.

[27] Murtaza Roondiwala, Harshal Patel, Shraddha Varma, “Predicting Stock Prices Using LSTM”, *International Journal of Science and Research (IJSR)*, vol. 6, issue 4, Apr. 2017.
[28] Raghav Nandakumar, Uttamraj K. R., Vishal R., Y. V. Lokeswari, “Stock Price Prediction Using Long Short Term Memory”, *International Research Journal of Engineering and Technology (IRJET)*, vol. 5, issue 3, Mar. 2018.
[29] Guangyu Ding, Liangxi Qin, “Study on the prediction of stock price based on the associated network model of LSTM”, *International Journal of Machine Learning and Cybernetics*, Nov. 2019.
[30] Chia-Cheng Chen, Chun-Hung Chen, Ting-Yin Liu, “Investment Performance of Machine Learning: Analysis of S&P 500 Index”, *International Journal of Economics and Financial Issues*, vol. 10, issue 1, pp. 59–66, 2020.
[31] Manish Agrawal, Asif Ullah Khan, Piyush Kumar Shukla, “Stock Price Prediction using Technical Indicators: A Predictive Model using Optimal Deep Learning”, *International Journal of Recent Technology and Engineering (IJRTE)*, vol. 8, issue 2, Jul. 2019.
[32] Xiao Dingy, Yue Zhang, Ting Liu, Junwen Duan, “Deep Learning for Event-Driven Stock Prediction”, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI)*, 2015.
[33] Tipirisetty Abhinav, "Stock Price Prediction using Deep Learning", *San Jose State University*, Master Thesis, 2018.
[34] Aroshine Munasinghe, Dajana Vlajic, “Stock market prediction using artificial neural networks”, *KTH Royal Institute of Technology*, Jun. 2015.
[35] https://www.naftemporiki.gr/finance/capitalization?market=ATH/
[36] S. C. Nayak, B. B. Misra, H. S. Behera, “Impact of Data Normalization on Stock Index Forecasting”, *International Journal of Computer Information Systems and Industrial Management Applications*, vol. 6, pp. 257–269, 2014
[37] Xiaodan Liang, Zhaodi Ge, Liling Sun, Maowei He, Hanning Chen, “LSTM with Wavelet Transform Based Data Preprocessing for Stock Price Prediction”, Hindawi, *Mathematical Problems in Engineering*, vol. 2019.
[38] Ian Goodfellow, “Manual Hyperparameter Tuning” in *Deep Learning*, 2016, ch. 11.4.1, pp. 416-419.
[39] https://www.altumintelligence.com/articles/a/Time-Series-Prediction-Using-LSTM-Deep-Neural-Networks/