NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

BSc THESIS

Deep Learning for cryptocurrency assets: Employing series forecasting models for price prediction and uncertainty quantification

Dimitrios A. Gangas

Supervisor: **Ioannis Emiris,** Professor

ATHENS

August 2020

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Βαθιά μάθηση για κρυπτονομίσματα: Χρησιμοποιώντας μοντέλα πρόβλεψης σειρών για εκτίμηση τιμών και ποσοτικοποίηση αβεβαιότητας

Δημήτριος Α. Γάγγας

Επιβλέπων: **Ιωάννης Εμίρης**, Καθηγητής

ΑΘΗΝΑ

Αύγουστος 2020

BSc THESIS

Deep Learning for cryptocurrency assets: Employing series forecasting models for price prediction and uncertainty quantification

Dimitrios A. Gangas
S.N.: 1115201400024

SUPERVISOR: **Ioannis Emiris,** Professor

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Βαθιά μάθηση για κρυπτονομίσματα: Χρησιμοποιώντας μοντέλα πρόβλεψης σειρών για εκτίμηση τιμών και ποσοτικοποίηση αβεβαιότητας

Δημήτριος Α. Γάγγας
Α.Μ.: 1115201400024

ΕΠΙΒΛΕΠΩΝ: **Ιωάννης Εμίρης,** Καθηγητής

ABSTRACT

Price prediction is one of the main challenges of quantitative finance. Within the realm of time series, cryptocurrencies are some of the most volatile and speculative, which makes the task of predicting them even more troublesome. This thesis presents a series of Neural Networks to provide a deep machine learning solution to the price prediction problem. Despite the difficulties, this research is concerned with two separate applications. The first focuses on predicting the next day's Opening, High, Low and Closing prices for Bitcoin and Litecoin; whilst the second one estimates prediction intervals for Bitcoin's Closing price. For the first application, a wide range of networks were tested and among them the one that gave the best results overall, proved to be a hyperparameter Bayesian optimized Long Short Term Memory (LSTM) network. For the second task, an approximate Bayesian Neural Network was created to provide an uncertainty estimation for Bitcoin. Uncertainty in this case is represented using prediction intervals, estimating the range within which future outcomes will fall. The research focuses specifically on Bitcoin for both tasks and Litecoin for the first, but could easily be extended for any other cryptocurrency or stocks using the methodologies that are presented.

SUBJECT AREA: Machine Learning

KEYWORDS: Deep Learning, Neural Networks, Bayesian Neural Networks, LSTM, Bayesian optimization, Cryptocurrencies

ΠΕΡΙΛΗΨΗ

Η πρόβλεψη τιμής θεωρείται μια από τις μεγαλύτερες προκλήσεις στον κλάδο της χρηματοοικονομικής και ποσοτικής ανάλυσης. Στο πλαίσιο των χρονοσειρών, τα κρυπτονομίσματα είναι από τα πλέον πιο ασταθή και ριψοκίνδυνα, γεγονός που καθιστά το έργο της πρόβλεψης τους ακόμη πιο προβληματικό. Η παρούσα πτυχιακή παρουσιάζει μια σειρά από νευρωνικά δίκτυα με σκοπό την επίλυση του προβλήματος της πρόβλεψης των τιμών τους βασισμένη στη Βαθιά Μάθηση. Παρά τις όποιες δυσκολίες, η συγκεκριμένη έρευνα εστιάζεται σε δύο ξεχωριστές εφαρμογές. Η πρώτη εφαρμογή αφορά την πρόβλεψη τόσο των αυριανών υψηλότερων και χαμηλότερων τιμών όσο και των αυριανών τιμών με τις οποίες ανοίγει και κλείνει το Bitcoin και το Litecoin. Ειδικότερα, σε αυτή την εφαρμογή, δοκιμάστηκαν μια ευρεία γκάμα από νευρωνικά δίκτυα και μεταξύ αυτών, εκείνο που έδωσε τα καλύτερα αποτελέσματα συνολικά, αποδείχθηκε ότι είναι η αρχιτεκτονική ανατροφοδοτούμενου νευρωνικού δικτύου γνωστή ως LSTM, της οποίας οι υπερ-παράμετροι ρυθμίστηκαν από τον αλγόριθμο Μπαεζιανής Βελτιστοποίησης. Η δεύτερη εφαρμογή προσπαθεί να εκτιμήσει το διάστημα διακύμανσης της αυριανής τιμής κλεισίματος του Bitcoin. Για την επίλυση της, δημιουργήθηκε ένα προσεγγιστικό Μπαεζιανό νευρωνικό δίκτυο με σκοπό την παροχή εκτίμησης της αβεβαιότητας για το Bitcoin. Η αβεβαιότητα σε αυτήν την περίπτωση προσδιορίζεται με τη χρήση διαστημάτων πρόβλεψης, εκτιμώντας το εύρος στο οποίο θα εντοπιστούν τα μελλοντικά αποτελέσματα. Η έρευνα της παρούσας πτυχιακής, παρόλο που επικεντρώνεται κυρίως για το Bitcoin και το Litecoin, θα μπορούσε εύκολα να επεκταθεί για οποιαδήποτε άλλο κρυπτονόμισμα ή μετοχή χρησιμοποιώντας τις μεθοδολογίες που παρουσιάζονται.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Μηχανική μάθηση

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Βαθιά μάθηση, Νευρωνικά Δίκτυα, Μπαεζιανά Νευρωνικά Δίκτυα, LSTM, Μπαεζιανή Βελτιστοποίηση, Κρυπτονομίσματα

To my beloved parents and grandmother for their invaluable support.

ACKNOWLEDGEMENTS

I would like to express my gratitude to my supervisor Professor Emiris I. not only for his guidance in selecting the subject for this thesis, but also for his immediate response whenever I needed further advice and feedback. I would also like to thank my second supervisor PhD candidate Christoforou E. for his continuous support in providing me with advice, suggestions and invaluable sources throughout the research project.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

This thesis is my final project as an undergraduate student in the Department of Informatics and Telecommunications of the National and Kapodistrian University of Athens. This dissertation has been my main focus during my final year. The thesis subject was a challenge for me because its content was beyond the scope of knowledge that I had attained during my degree. The entire process opened up a completely new realm of knowledge which has led me into the field of Deep Learning and time series analysis.

1. INTRODUCTION

In the wake of the 2008 financial crisis, trust in banks, financial institutions and governments has melted away amongst the population; this is especially true for the younger, more tech-savvy demographic. It is from this group of people that Bitcoin emerged. A central tenet of cryptocurrencies is to avoid using banks or established financial institutions to route money or accept payments. Hence, it eliminates the need for banks as third-party guarantors of transactions, and restricts the ability of governments to interfere or regulate payments. In 2010, one bitcoin was worth only $0.06, whereas 50 BTC would have been enough for a coffee. At its peak in 2017, those same 50 BTC would have been worth $850k. Was this rapid growth predictable?

Cryptocurrency and stock price forecasting in general, is one of the most important tasks of quantitative finance. It goes without saying that profits are the guiding force behind most investment choices. However, stock market prices and especially these newly-popped-up virtual currencies do not behave as simply as time series do, making it much harder to make robust and consistent predictions. The theory of cryptocurrency price prediction has not only been a major discussion topic in Finance recently, but has also fueled interest within the scientific community [4][5][6]. Cryptocurrencies have faced periodic rises and sudden plunges during specific time periods, and therefore the cryptocurrency trading community has a need for a standardized method to accurately predict the fluctuating price trends as much as possible. Traditional time series prediction methods include univariate Autoregress-ive (AR), Univariate Moving Average (MA), Simple Exponential Smoothing (SES), and Autoregressive Integrated Moving Average (ARIMA). Unfortunately, mainly due to the lack of seasonality and their high volatility in the cryptocurrencies market, these methods are under-performing [7]. On the contrary, Machine Learning methods seem promising for this task. Examples of studies within the scope of Machine Learning to predict Bitcoin prices include random forests [8], neural networks [9] and the Bayesian neural network [10]. Deep Learning models, particularly deep feedforward neural networks, which have recently sparked the interest of new investors on the financial market, have already found numerous applications in quantitative finance, such as volatility forecasting, financial investment problems and price prediction.

In a supervised learning scheme, in contrast to the traditional time series models such as ARIMA and its extensions, neural networks are a useful tool for price prediction since no strong assumption is needed for their application and consequently, they are able to catch patterns with an important generalization power. In addition, more complex deep learning methods such as RNNs and LSTM networks seem more appropriate for sequential data and should yield substantially higher prediction accuracy. Indeed, many studies in the field [9][11] have compared RNN's and LSTM models to forecast Bitcoin pricing with traditional multilayer perceptron (MLP) and other statistical models, and found significantly lower mean absolute error (MAE) in LSTM prediction, supporting the aforementioned statement. In the same manner, my supervisor has done similar research work, where he indicated that RNNs provide the most promising results in both cryptocurrency trend classification and regression setting [12].

In the scope of this thesis, we first provide an overview of the theory behind EMH in finance and in following, the theory behind Deep Learning, RNN, LSTM and BNN models by explaining the basic concept of neural networks, their components and architectures. Finally, we present our two applications and their results. The purpose of the first is to implement a series of Deep Learning models to predict the price of next day's LTC and BTC price in a regression setting; whereas, in the second we implement an approximate Bayesian Neural Network to provide confidence intervals for the Closing BTC price to accompany the predictions of the first application.

2. FINANCIAL AND CRYPTOCURRENCY BACKGROUND

This chapter provides an introduction to the financial aspects of the thesis. We aim to describe the efficient market hypothesis and its implications for predicting futures' returns. We continue by discussing the role of Blockchain technology at its most basic level, which is considered to be the backbone of the new virtual currencies.

2.1 Efficient Market Hypothesis

Since the purpose of this thesis can be summarized as attempting to predict tomorrow's asset movement using only assets' past prices and consequently, market movements as input, a discussion of the efficient market hypothesis (EMH) seems relevant. The EMH has been an important concept in the theory of financial markets since its formalization by Eugene Fama [13] and can be broken down as a set of assumptions about the predictability of future market prices. It derives its name from the definition of efficient markets; prices fully reflect all available information. The three forms of efficient market hypothesis are the following:

- **Weak Form:** The weak form of the EMH states that current prices fully reflect the information to be gained from historical prices. This implies that it is impossible to consistently outperform the market using investment strategies that rely on past prices, such as attempted in this thesis. The claim has been widely discussed and has gone from being generally accepted in earlier studies [13] to being generally dismissed and even disproven in certain settings [14][15][16].
- **Semi-strong Form:** The EMH in its semi-strong form loosens the restriction of relying on past prices and includes all publicly available information. This form of the hypothesis therefore infers that one can not consistently outperform the market using any investment strategy based on public information.
- **Strong Form:** In this strictest form of EMH, the available information is generalized to include all information including insider information not known to the general public. The semi-strong form as well as the strong form include the weak form and make even further assumptions.

Since even the weak form has been the target of some debate, the two stronger forms of the EMH are generally accepted as unfeasible. In support of that, with the appearance of Behavioral Finance, many financial economists believe that stock prices are at least partially predictable on the basis of historical stock price patterns, which reinvigorate Fundamental and particularly Technical analysis as tools for price prediction.

2.2 Blockchain Technology

Understanding how the blockchain works with bitcoin will allow us to see how the technology can be transferred to many other real-world use cases. Satoshi's innovation, that powers Bitcoin and all cryptocurrencies that came after it, is called Blockchain. A blockchain is basically a ledger of transactions, much like a bank maintains, but copies of that ledger are distributed among computers all over the world, automatically updating with every transaction. Always up-to-date and 100% verified. Maintaining this distributed ledger demands a lot of work, but no one is required to do this task. Instead, the system pays out cryptocurrency to those who volunteer to do it, known as miners. For the first time, we can have a distributed network come to consensus on what transactions took place on the network. This allows for a ”trust-based network”. A ledger where there isn't a single central authority maintaining it, but rather a decentralized system in which a network of people maintain it. Nakamoto didn't just solve the cash problem on the internet; he had a solution for the problem of trust on the internet, too, thus facilitating a new type of interaction. The true identity of Nakamoto is still unknown, but his/their vision was laid out in a 2009 whitepaper called ”Bitcoin: A Peer-to-Peer Electronic Cash System” [17]. Historically, Bitcoin was blockchain's first trial run. However, despite the great potential of blockchain, it faces numerous challenges which limit its wide usage. The most prominent drawback it faces is the lack of scalability. With the amount of transactions increasing day by day, the blockchain becomes bulky. An example that supports this statement is the fact that the Bitcoin blockchain can only process roughly 7 transactions per second [18]; whilst Visa's payment network is capable of performing 24.000 transactions per second. We still have a long way towards mainstream adoption, but many industries have been adopting blockchain systems as of late. Over the next few years we will likely see businesses and governments experimenting with new applications to find out where blockchain technology adds the most value.

3. DEEP LEARNING BACKGROUND

Deep learning is a subfield of Machine Learning based on artificial neural networks with representation learning. Learning can be supervised, semi-supervised or unsupervised. Even though deep learning is a fairly old field, it only gained significant attention in the early 2010s. In the few years since then, it has achieved remarkable results in various fields such as image recognition, Natural Language Processing and Reinforcement Learning, involving skills that seem natural and intuitive to humans but have long been elusive for machines. With the latest developments in optimization, learning algorithms and hardware, it's fair to say that neural networks will continue to gain momentum for the next years to come. It's a new approach to learning representations from data that puts an emphasis on learning successive layers of increasingly meaningful representations. In a neural network, we don't specify for the computer how to solve our problem. Instead, it learns from observational data, figuring out its own solution to the problem at hand.

The adjective "deep" in Deep Learning stands for this idea of successive layers of representations, or in other words, refers to the use of multiple layers stacked on top of one other in the neural network. The number of layers that contribute to the model of the data is called the *depth* of the neural network. The term neural networks is a reference to neurobiology, however, while some of the central concepts in deep learning were inspired from brain neuron functioning, there is no evidence that the brain works in the same manner and the current conclusions are far from completely explaining the complex functioning of the brain, while still offering very effective solutions to many problems. Thus, for our purpose, deep learning is a sophisticated mathematical framework for learning representations from the data.

A neural network is a parametric model that aims to approximate the mapping $f : \mathcal{X} \to \mathcal{Y}$, given a data set $\mathcal{D} = \{x_i, y_i\} \subset (X, Y)$. The set of the parameters of the neural network is called weights w, and the problem of finding the set of weights that best approximate the mapping f is solved via Maximum Likelihood Estimation (MLE).

$$w^{MLE} = \underset{w}{\operatorname{argmax}} \log p(\mathcal{D} \mid w)). \tag{3.1}$$

3.1 Feed-forward Neural Networks

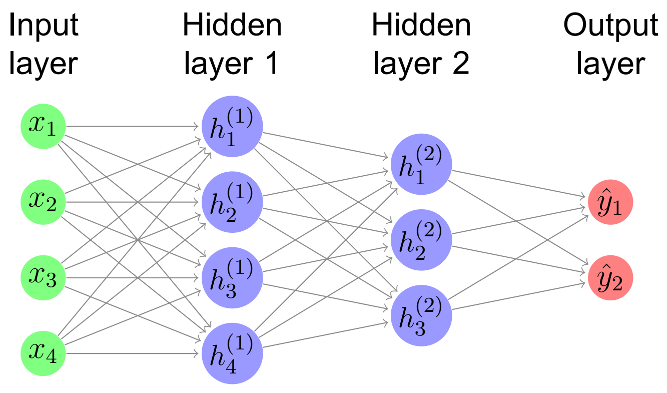


Figure 3.1: A Multilayer perceptron with 4 input units corresponding to the input vector $x = (x_0, x_1, x_2, x_3)$, 2 hidden layers with 4 and 3 units for each layer respectively and 2 output units corresponding to the output vector $\hat{y} = (\hat{y}_1, \hat{y}_2)$

The most basic type of neural net is the feed-forward network, which is a generalization of the linear perceptron, where layers of artificial neurons are stacked together, with non-linearities applied between each layer. The reason behind the various successful applications of neural networks is due to the universal approximation theorem [19]. It proves that feed-forward neural networks with non-linear activation functions and at least one hidden layer are universal function approximators. The universal approximation theorem has also been proved for a wider class of activation functions, such as *ReLU* [20].

Given an input vector $x \in \mathbb{R}^p$ and an output vector $y \in \mathcal{Y}$, a feed-forward neural network with m hidden layers and h_i nodes in each layer and weights w, bias b and non-linear activation function ϕ, produces a prediction by:

$$\begin{aligned} h^{(1)} &= \phi(w^{(1)}x + b^{(1)}), \\ h^{(2)} &= \phi(w^{(2)}h^{(1)} + b^{(2)}), \\ &\vdots \\ y^* &= g(w^{(m)}h^{(m-1)} + b^{(m)}) \end{aligned}$$

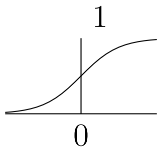
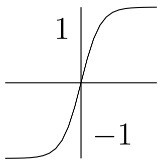
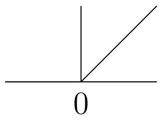
where $w^{(i)} \in \mathbb{R}^{h_i \times h_{i\text{-}1}}$ and $b^{(i)} \in \mathbb{R}^{h_i}$.The function g can be a soft-max in the case of classification or linear, which often is in the case for regression problems. However, a myriad other choices for g exist.

3.2 Activation function

As discussed in brief, to enable neural networks to approximate complex non-linear transformations, we must introduce some form of non-linearities to the model. On the contrary, if the activation functions were linear, no matter how many layers the neural network has, all would collapse into one. This occurs because a linear combination of linear functions is still a linear function. Thus, non-linear activation functions are essential to enable neural networks to approximate complex non-linear transformations from complex data, such as images, video, audio, and data sets which are non-linear or have high dimensionality. Almost any process imaginable can be represented as a functional computation in a neural network, provided that the activation function is non-linear. Traditionally, two widely used non-linear activation functions are the *sigmoid* and hyperbolic tangent activation functions. Of course, a plethora of other possible choices exist for the activation function, but for the scope of this thesis we will consider these three different functions, the two being the hyperbolic tangent and *sigmoid* or also known as logistic function. Hyperbolic tangent maps x to the interval (-1,1), whereas the output values in *sigmoid* function bound between 0 and 1.

A common drawback with both the *sigmoid* and *tanh* functions is that they tend to saturate. This means that large values snap to 1 and small values snap to -1 or 0 for *tanh* and *sigmoid* respectively. For very high or very low values of x, there is almost no change to the prediction, causing the well-known problem of *vanishing gradient* because it causes the gradients to converge to 0. Vanishing gradients make it difficult to know which direction the parameters should move to improve the cost function [2]. This leads the network to refuse to learn further, or to be too slow to reach an accurate prediction. To overcome these problems, the Rectified Linear Unit (*ReLU*) activation function was introduced. This has become a widely used approach in recent machine learning research, mostly due to its computation efficiency (allows the network to converge very quickly) and strong performance in many tasks. Although it looks like a linear function, *ReLU* has a derivative function and allows backpropagation.

Table 3.1: Non-linear activation functions.

Name	Function	Derivative	Figure
Sigmoid	$\sigma(x) = \frac{1}{1+e^{-x}}$	$\sigma'(x) = \sigma(x)(1-\sigma(x))$	
Tanh	$tanh(x) = \frac{e^x - e^{-x}}{e^z + e^{-z}}$	$tanh'(x) = 1 - tanh(x)^2$	
ReLU	$R(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0. \end{cases}$	$R'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0. \end{cases}$	

3.3 Loss function

To control something, first we need to be able to observe it. To control the output of a neural network, we need to be able to measure how far this output is from what we expected. This is the task of the loss function of the network, also known as the objective function. The loss function takes the predictions of the network $\hat{y}$ and the true target y and computes a distance score, capturing how well the network has done on this specific example. We are concerned with finding the set of weights W that is able to fit to the training data, while also providing generalization to new data.

Given the fact that we are modelling a continuous response variable (crypto's next day price), we are faced with a regression problem. As with activation functions, here as well a myriad of possible loss functions exist. For our task, a natural loss metric is the mean squared error (MSE). This measures the squared average distance between the real data and the predicted data.

$$MSE = \frac{1}{N}\sum_{i}^{N}(y_i - \hat{y}_i)^2 \tag{3.2}$$

where y_i is the actual outcome and $\hat{y}_i$ is the model's prediction. Due to squaring, predictions which are far away from actual values are penalized heavily in comparison to less deviated predictions. Using MSE as loss function is equivalent to maximum likelihood estimation using a Gaussian likelihood for the data given the model's parameters.
Other well-known loss functions for regression tasks are the Mean Absolute Error (MAE), Root Mean Square Error (RMSE) and Mean Absolute Percentage Error (MAPE).

3.4 Back-propagation algorithm

So far we have specified how a neural network can be universal function approximator, and we have also defined an objective function that determines how well our current model actually fits the data. The last question that remains to be answered is how we adjust the weights parameters in the neural network to minimize the loss function and hence, to best approximate the mapping $\mathcal{Y}$. The way this is generally done is by utilizing an algorithm called backpropagation with the help of gradient descent.

- **Forward pass:** In feed-forward networks, when we feed an input x, information propagates forward through the hidden layers in the network to exit through the output layer and produce $\hat{y}$. This forward propagation step is called forward pass. To be able to update the weights of the network correctly, we need to be able to propagate the information from the loss, backward in the network, during a second step called the backward pass.
- **Backward pass:** The back-propagation algorithm or backprop [21], allows this feedback of information that enables the computation of the gradient. It is the most popular method for supervised training of MLPs. Backprop is not a learning algorithm; it is only a computation method that allows the network to learn, utilizing an optimization algorithm such as gradient descent. Basically, it is an algorithm that expresses the error gradient with respect to quantities of a given neuron as a function of its outgoing neurons. This is possible thanks to the chain rule of calculus that computes

the derivatives of the composition of several functions. Therefore, we are now able to calculate how much the weights that resulted in those values should be changed and then alter them accordingly.

This entire cycle of forward and backward pass through the full training data set is known as an epoch. Since one epoch is too big to feed into the neural network at once, we divide it into several smaller batches. We can think of a for-loop over the number of epochs where each loop proceeds over the training data set. Within this for-loop is another nested for-loop that iterates over each batch of samples, where one batch has the specified ”batch size” number of samples.

3.5 Optimizer

The simplest algorithm is known as gradient descent where we repeatedly take small steps in direction of the negative error gradient of the parameter θ. For each iteration we get the following update:

$$\theta = \theta - \eta \nabla Q(\theta) \tag{3.3}$$

where η is the *learning rate* and $\nabla Q(\theta)$ is the gradient of the loss function w.r.t the parameter. Generally, a large learning rate allows the model to learn faster, at the cost of arriving at a sub-optimal final set of weights. A smaller learning rate may allow the model to learn a more optimal or even globally optimal set of weights but may take significantly longer to train.

There are three variants of gradient descent, which differ in how much data we acquire to compute the gradient of the objective function. Depending on the amount of data, we make a trade-off between the accuracy of the parameter update and the time it takes to perform an update. The three variants are the following:

- **Batch gradient descent** or aka **vanilla gradient descent** optimizer updates the parameters of the model after each epoch. So, if the dataset is too large it may take a lot of time to converge to the minima. Batch gradient descent can be very slow and is impractical for datasets that don’t fit in memory.
- **Stochastic gradient descent (SGD)** optimizer, in contrast, performs a parameter update for each training example (x_i, y_i). Counter-intuitively SGD usually performs much faster and can also be used to learn online. On the downside due to frequent updates the steps taken towards the minima are very noisy. Hence, it often leads the gradient descent to overshoot into other directions.
- **Mini-batch gradient descent** finally takes the best of both worlds and performs an update for every (x_i, y_i) mini-batch of n training examples where the size of the minibatches is determined by the batch size. This ensures that the following advantages of both stochastic and batch gradient descent are used thus, Mini-Batch Gradient Descent is most commonly used in practice. For a given iteration, we get the following update:

$$\theta = \theta - \eta \nabla \sum_{i}^{n} Q(\theta, x_i, y_i) \tag{3.4}$$

SGD is the basis for many other learning algorithms, such as AdaGrad or RMSProp, where the difference lies in the adaptive learning rate that the latter optimizers sustain. The performance of the model on the training dataset can be monitored by the learning algorithm and the learning rate can be adjusted in response, leading to a more robust strategy in avoiding local minima or flat regions. Another adaptive algorithm that nowadays is one of the most used optimization algorithms, is the Adaptive Moment Estimation (Adam) [22]. Adam can be looked at as a combination of RMSprop and Stochastic Gradient Descent with momentum. In all the proposed models In this dissertation, the Adam optimizer is used.

3.6 Recurrent neural networks

In the feedforward neural networks that we have discussed so far, we considered MLPs without any cyclic connection, which makes MLP a static model in the sense that the input-output pairs are mutually independent [23]. This seems inefficient if we want to model sequential data, where the input features are interdependent. If we form such cyclic connections, we obtain a recurrent neural network (RNN), which can model dynamic processes as such are the time series.

Recurrent networks are now used in various applications such as stock price forecasting, audio, speech or even language processing, where in order to predict the next word in a sentence, the previous words are required and there's a need for our neural network to remember them.

Each cycle makes it possible for a neuron to follow a path back to itself, allowing information feedback. These cycles, or recurrent edges, allow the network's hidden units to see their own previous output, so they give the network memory [24] and introduce the notion of time into the model. The recurrent neurons are sometimes referred to as context neurons or state neurons. The structure of a simple recurrent neural network is shown in Figure 3.2 with its folded and unfolded representation.

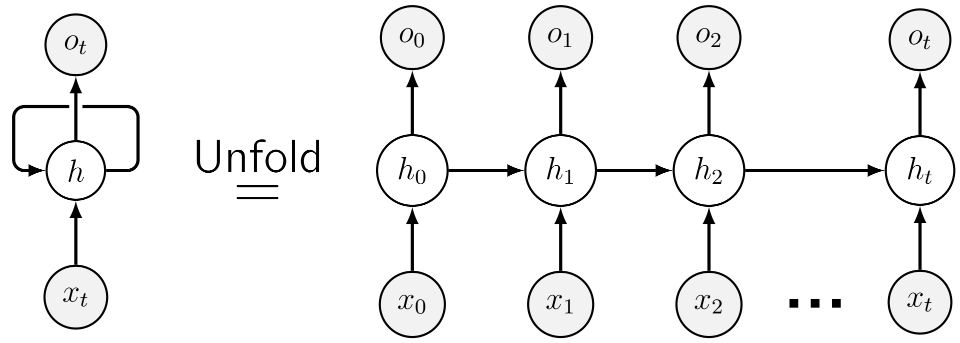


Figure 3.2: a simple RNN that maps an input sequence of x values to a corresponding sequence of output o values in its folded and unfolded version for t time steps

For the RNN to be effective on real problems, two major issues needed to be resolved for the network to be useful: **(a)** How to train the network with Back propagation and **(b)** How to stop gradients vanishing or exploding during training.
For the first issue unfortunately, the backpropagation that was discussed in the previous section breaks down in a recurrent neural network, because of the recurrent or loop connections. This was addressed with a modification of the backpropagation technique called *Backpropagation Through Time(BPTT)*. Instead of performing backpropagation on

the recurrent network as stated, the structure of the network is unrolled, where copies of the neurons that have recurrent connections are created. For example, a single neuron with a connection to itself (A $\rightarrow$ A) could be represented as two neurons with the same weight values (A $\rightarrow$ B). This allows the cyclic graph of a recurrent neural network to be converted into an acyclic graph like a classic feedforward neural network, and then Backpropagation can be applied. Consequently, unrolling recurrent neural networks can create a very deep neural network that could potentially, in the Backpropagation step, lead the gradients, which are calculated in order to update the weights, to become unstable.

These can be very large numbers or very small numbers, consequently causing the exploding or vanishing gradients problem as noted. These large numbers, in turn, are used to update the weights in the network, making training unstable and the network unreliable. To prevent this, a new type of architecture called the Long Short-Term Memory Networks, introduced gates to force the derivatives of the loss to lie within a pre-defined range.

3.7 Long Short-Term Memory network

Long short-term memory (LSTM) is an RNN architecture introduced in 1997 by Hochreiter and Schmidhuber [25] in order to provide a viable solution to the long term dependency problem and the vanishing gradient problem that traditional RNNs possessed. LSTM networks, with the Gated Recurrent Unit (GRU) are the most popular and effective neural network models for sequence learning. Both are gated recurrent neural networks. The idea behind GRU and LSTM units is to create connections through time with a constant error flow, thus the gradient neither explodes nor vanishes. They are explicitly designed to mitigate the long-term dependency problem. Remembering information for long periods of time is practically their default behaviour, making them suitable for sequence learning.

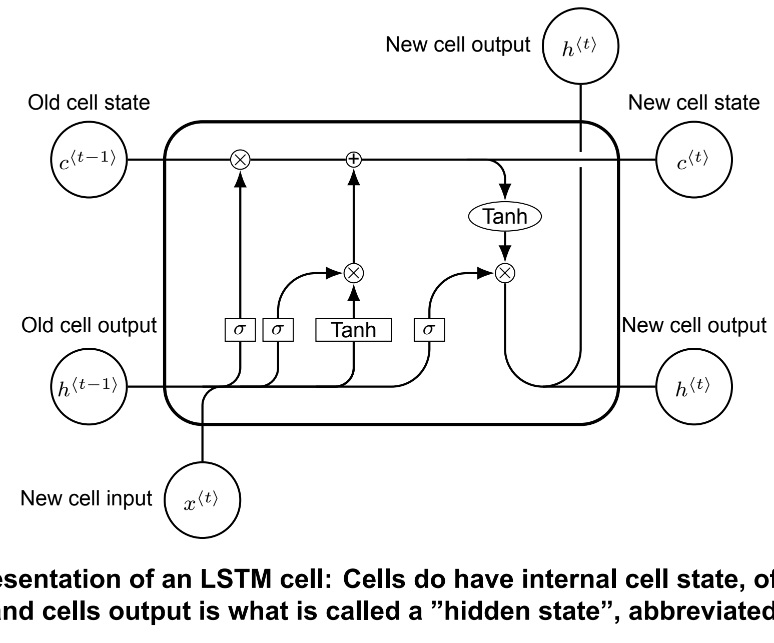


Figure 3.3: Representation of an LSTM cell: Cells do have internal cell state, often abbreviated as "c", and cells output is what is called a "hidden state", abbreviated as "h"

Instead of neurons, LSTM networks have memory blocks that are connected in layers. A block has components that make it more sophisticated than a classical neuron and a memory for checking recent sequences. A block also possesses gates that manage the block's state and output. A unit operates upon an input sequence and each gate within a unit uses the *sigmoid* activation function to control whether they are triggered or not, making the change of state and addition of information flowing through the unit in a conditional manner. Cell state is the central feature. It is referred to as a constant error carousel (CEC) in Hochreiter and Schmidhuber work [25]. The cell state produces only some minor linear transformations, achieving a constant error flow through the memory block and thus acting as the ”memory” of the network. As the gates are concerned, there are 3 types of them inside a memory unit:

- **Input Gate:** has the responsibility to update the cell state. First, it passes the previous hidden state and current input into a *sigmoid* function. That decides which values will be updated by transforming the values to be between 0 and 1. 0 means not important whereas 1 means important. It also passes the hidden state and current input into the *tanh* function to squish values between -1 and 1 in order to regulate the network. Then it multiplies the *tanh* output with the *sigmoid* output. The *sigmoid* output will determine which information is important to keep from the *tanh* output.

- **Output Gate:** The output gate figures out what the next hidden state should be. Note that the hidden state contains information on previous inputs. The hidden state is also used for predictions. First, we pass the previous hidden state and the current input into a *sigmoid* function. Then we pass the newly modified cell state to the *tanh* function. We multiply the *tanh* output with the *sigmoid* output to decide what information the hidden state should carry. The output is the hidden state. The new cell state and the new hidden is then carried over to the next time step.

- **Forget Gate:** This gate allows the memory block to reset itself, thanks to a *sigmoid* activation function. It decides what information should be thrown away or kept. Information from the previous hidden state and information from the current input is passed through the *sigmoid* function. Values come out between 0 and 1. The closer to 0 means to forget, and the closer to 1 means to keep. Historically, Forget Gate was added 3 years after the first publication of LSTMs to remedy the lack of resets in the internal states that could potentially cause the breakdown of the network [26].

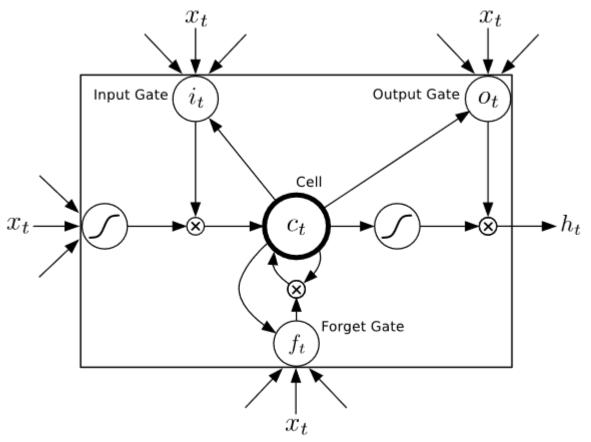


Figure 3.4: Long Short-term Memory Cell illustrating the Gates, source: Graves [1]

To recapitulate, the forget gate decides what is relevant to keep from prior steps. The input gate decides what information is relevant to add from the current step. The output gate determines what the next hidden state should be.

Let x_t be one observation at time t of the input vector, the LSTM cell is implemented [1] by the following equations:

$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i), \tag{3.5}$$
$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f), \tag{3.6}$$
$$c_t = f_t c_{t-1} + i_t tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c), \tag{3.7}$$
$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_t + b_o), \tag{3.8}$$
$$h_t = o_t tanh(c_t) \tag{3.9}$$

where σ is the *logistic sigmoid* activation function; i, f, o and c are respectively the *input gate*, *forget gate*, *output gate*, *cell* and *cell input* activation vectors, all of which are the same size as the hidden vector h. The weight matrix subscripts have the obvious meaning, for example W_{hi} is the hidden-input gate matrix, W_{xo} is the input-output matrix etc. The weight matrices from the cell to gate vectors (e.g. W_{ci}) are diagonal, so element m in each gate vector only receives input from element m of the cell vector. The bias terms (which are added to i, f, c and o) have been omitted for clarity.

3.8 Bayesian Neural Networks

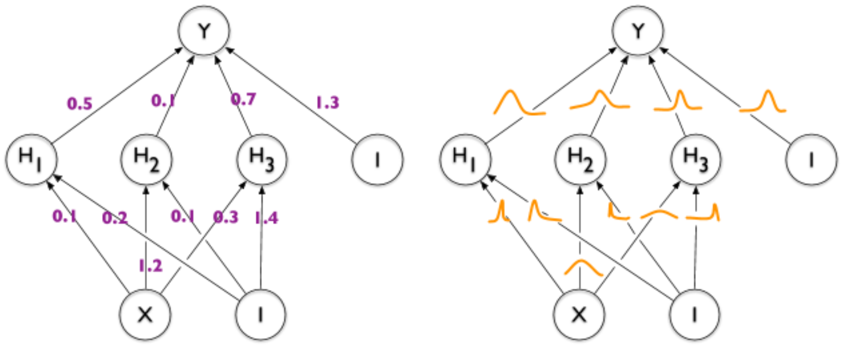


Figure 3.5: Left: deterministic Neural Networks have a fixed value of their parameters,as provided by classical backpropagation. Right: In Bayesian Neural Networks each weight is assigned a distribution.

Bayesian Neural Networks (BNN), in contrast to its frequentist counterparts, are networks whose weights or parameters are expressed as a distribution rather than a deterministic value and learned using Bayesian inference. The capability of simultaneously learning complex non-linear representations from the data and quantifying uncertainties make them very appealing in certain domains. In Neural networks we showed that finding the set of parameters w is solved via Maximum Likelihood Estimation (MLE). This is often not desirable because it does not guarantee that our model will perform well in unseen data, leading to overconfidence in predictions. Bayesian approach, on the other hand, provides a more principled path in creating inferences from the data.

More precisely, Bayesians impose a prior distribution on the network's weight which reflects original beliefs about the parameters and then attempt to compute the posterior posterior density of w given the data D using Bayes rule:

$$p(w|D) = \frac{p(D|w)p(w)}{p(D)} = \frac{p(D|w)p(w)}{\int p(D|w)p(w)dw} \tag{3.10}$$

where $p(w|D)$ is the posterior of w given data D, $p(D|w)$ is the likelihood of parameters w in our model, $p(w)$ is the prior and $\int p(D|w)p(w)dw$ is the evidence.

This operation returns a probability distribution over all possible configurations of the parameters, giving more weight on settings which probably have generated the data better. This distribution encodes our uncertainty about what values we should place in these parameters. Therefore, predictions for new datapoints $X^* = \{x_1^*..x_n^*\}$ are made through marginilization, which yields the posterior predictive distribution:

$$p(Y^*|X^*, D) = \int p(Y^*|X^*, w)p(w|D)dw \tag{3.11}$$

In this fashion, uncertainty in models parameters is translated into uncertainty in predictions, allowing to create reliable error bounds and avoid overfitting. Intuitively BNN's treat the prediction process expressed in Equation 3.11 as a weighted average of an ensemble of an infinite amount of networks trained in dataset D.

Unfortunately, the non-linearities in NN's make the computation of the integral in Equation 3.11 intractable. Analogously, the integral for the model evidence (denominator in Equation 3.10) is also intractable.

Various ways to approximate the posterior have been developed which yields us with a wide variety of BNNs today. Well-known methods among them are Laplace Approximation [27], Variational Inference [28] and Markov Chain Monte Carlo (MCMC).

Approximate Bayesian Inference by Monte Carlo Dropout

In this work, we make use of the relatively new approach called MC-Dropout [29][30]. The authors proved that using dropout during test time is identical to Variational Inference in Gaussian processes and as a result can lead to Bayesian Inference approximation, providing good uncertainty estimates. Monte Carlo Dropout offers a new and handy way to estimate uncertainty with minimal changes in the existing networks. We just need to keep our dropout on at test time, then forward pass the data for T times and store all the predictions. The posterior predictive distribution is now facilitated by averaging the predictions of those stochastic forward passes. Consecutively, the prediction intervals can now be estimated for every prediction via their sample variance.

3.9 Generalization methods

The central challenge in Machine Learning is that our algorithm must perform well on new, previously unseen data and not only on those which our model was trained on. The ability to perform well on previously unobserved inputs is called *generalization*. In this section, we present some problems that deep neural networks can face concerning generalization and some solutions.

3.9.1 Model selection

Model selection in the context of Machine Learning can have different meanings, corresponding to different levels of abstraction. Before diving into the details of different approaches to model selection, and when to use them, there is "one more thing" we need to discuss: model evaluation. Model evaluation aims at estimating the generalization error of the selected model, i.e., how well the selected model performs on unseen data. Obviously, a good machine learning model is a model that not only performs well on data seen during training, but also on unseen data. Otherwise the machine learning model could simply memorize the training data. Hence, before shipping a model out for production, we should be fairly certain that the model's performance will not degrade when it is confronted with new data.

Holdout selection procedure

The holdout method is arguably the simplest model evaluation technique. We take our labeled dataset and split it into three sets: A training set, a validation set and a test set. We then use the training set to fit different learning machines and we apply these models to the validation set. We select the model which has the best performance on the validation set and apply this model to the test set in order to measure the *generalization error*. For our regression task the evaluation metric will be the Mean Square Loss function as discussed in the previous chapter.

So far, what we have described is simply an optimization problem. What separates machine learning from optimization is that we want the *generalization error*, also called the *test error*, to be low as well. The generalization error evaluates the prediction power of a model on unknown data, here the test set, which was not used for the training of the model.

Underfitting vs Overfitting

The factors determining how well a machine learning algorithm will perform are its ability to **(a)** make the training error small and **(b)** make the gap between training and test error small. These two factors correspond to the two main challenges in machine learning: underfitting and overfitting. Note that in Statistics and Machine Learning, this challenge is known as the bias–variance dilemma or bias–variance problem, because it's trying to simultaneously minimize these two sources of error that prevent supervised learning algorithms from generalizing beyond their training set.

- **Underfitting** occurs when the model is not able to learn the underlying structure of the data and obtain a sufficiently low training error.
- **Overfitting** occurs when the gap between the training error and test error is too large. Hence, the model will have good performance on the training set, but not on the test set due to poor generalization.

We can control whether a model is more likely to overfit or underfit by altering its capacity (or complexity). Informally, a model's capacity is its ability to fit a wide variety of functions. Models with low capacity may struggle to fit the training set, whereas models with high capacity can overfit by memorizing properties of the training set that do not serve them well on the test set.

3.9.2 Regularization techniques

In neural networks, due to the high complexity, they have a tendency to overfit the given training data, which yields poor generalization for unseen data points. To overcome this problem, several regularization techniques have been proposed. Overall, regularization is

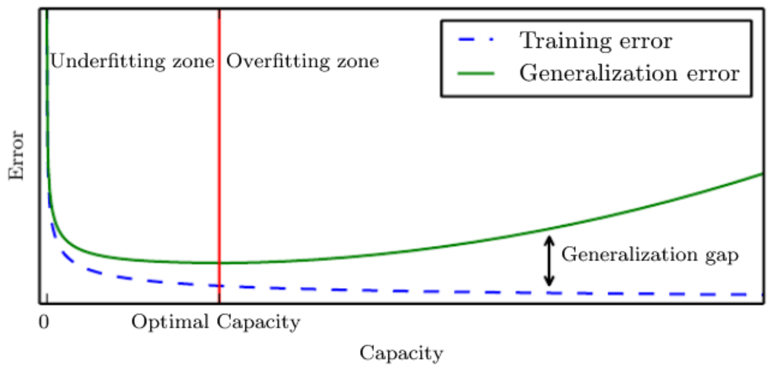


Figure 3.6: Typical relationship between capacity and error, source: Goodfellow et al.[2]

a technique to avoid the overfitting of models with a large learning capacity. The regularization techniques increase the bias and reduce the variance of the model. In this section, I will try in short to explain those that I utilized in my models.

3.9.2.1 Early stopping

An alternative to regularization as a way of controlling the effective complexity of a network is the procedure of early stopping (Optimal Capacity in Figure 3.6). Early stopping is a method that stops training when a monitored quantity has stopped improving. Indeed, if the performance of the model on the validation dataset starts to degrade (e.g. loss begins to increase, or accuracy begins to decrease), then the training process is stopped. The model at this stage has low variance and is known to generalize the data well. Training the model further would increase the variance of the model and lead to overfitting. This regularization technique is called “early stopping”. Early stopping is one of the most used regularization techniques in deep learning, because of its simplicity and its effectiveness in reducing training time.

3.9.2.2 Dropout

Dropout is also one of the most common techniques for addressing the problem of overfitting in deep neural networks. The key idea is to randomly drop units (along with their connections) from the neural network during training with a given probability p as a hyperparameter. This prevents units from co-adapting too much [31]. It basically creates an ensemble of sparse networks, which can significantly increase the model’s ability to generalize and prevent overfitting. At test time, it is easy to approximate the effect of averaging the predictions of all these thinned networks by simply using a single unthinned network that has smaller weights.

3.9.2.3 L2-regularization

L2-regularization, also known as Ridge regression, penalizes large weights by adding a penalty to the model’s loss function, multiplied by a regularization parameter $\lambda \in \mathbb{R}^+$.

$$\mathcal{L}(y^*|x^*, w) + \lambda ||W||^2 \tag{3.12}$$

where $\mathcal{L}(y^*|x^*, w)$ denotes the network's loss function of a new prediction, given data and its trained weights W. Large values of λ yield to heavy regularization, whilst small values result in light to no regularization. From a probabilistic perspective, L2 regularization equals introducing a Gaussian prior on the weights of the network.

3.10 Hyperparameter Optimization

There are plenty of parameters you can select when setting a Neural Network before training begins. These are often called hyperparameters and in comparison with parameters, which are learned by the model during training, hyperparameters are set by the user before the learning process. Hyperparameters, for example, could be the number of layers, the number of neurons in each layer, the batch size, the type of activation function and optimization to use, etc. The optimization method also has one or more hyperparameters you can select, such as the learning rate and learning decay. One way of searching for good hyperparameters is by hand, where you try one set of parameters and see how they perform, then you try another set of parameters and check if they improve performance and so on. You try to build an intuition for what works well and guide your parameter-search accordingly. Not only is this extremely time-consuming, but the optimal parameters are often counter-intuitive to conceive of.

Another way of searching for good hyperparameters is the Grid Search technique. Grid Search can be thought of as an exhaustive search for selecting a model. It basically divides each parameter's valid range into evenly spaced values, and then simply have the computer try all combinations of parameter values. Although the flowchart becomes automated since it is run entirely by the computer, it quickly becomes obvious that it suffers from the Curse of Dimensionality problem. Adding more hyperparameters increases exponentially the number of parameters-combinations, making it extremely time-consuming. If you have, for example, 5 hyper-parameters to tune and each of them can take 10 different possible values, then there are a total of 10^5 different combinations of parameters. In the event of adding just one more hyperparameter, then there are 10^6 different parameter combinations, resulting in exponential space increase.

In contrast to Grid Search, Random Search is a technique where random combinations of the hyperparameters are used to find the best solution for the built model. It is similar to Grid search, and yet it has proven, given the same resources, to yield better results comparatively [32].

However, Grid and Random Search are relatively inefficient because they do not choose the next hyperparameters to evaluate based on previous results. Both search techniques are completely uninformed by past evaluations, and as a result, often spend a significant amount of time evaluating ”bad” hyperparameters.

Bayesian Search, on the other hand, solves the above problem. It's an another well-known method for hyperparameter optimization. This technique is particularly suited for optimization of high-cost functions, situations where the balance between exploration and exploitation is important. It is based upon Bayes Rule and considers previously known knowledge to help narrow down the search space of good hyperparameter combinations. It works by building a probabilistic model of the objective function, called the surrogate function. Then to sample efficiently, Bayesian optimization uses an acquisition function to determine the next candidate samples (location), before evaluating them on the real objective function [33][3]. A common choice for acquisition function is the Expected Improvement (EI).

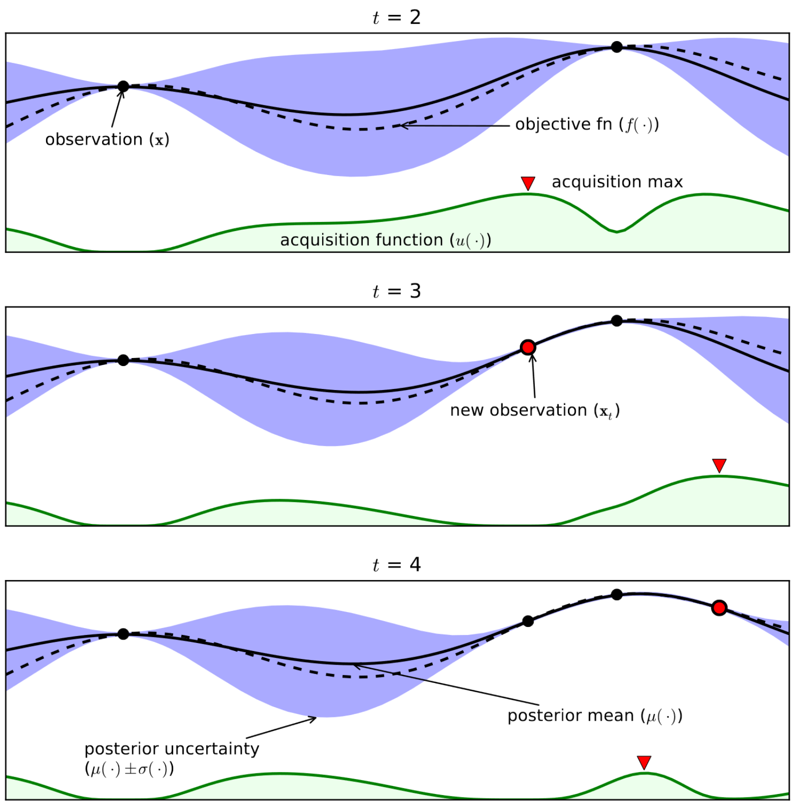


Figure 3.7: Bayesian optimization, source: Brochu et al.[3]

The Figure 3.7 shows an example of using Bayesian optimization over a hyperparameter. The Gaussian process (GP) approximation of the objective function over four iterations of sampled values of the objective function is depicted in purple.

The Figure also shows the acquisition function in the lower green shaded plots. We can easily observe that the acquisition is high **(a)** where the Gaussian Process predicts a high objective (high accuracy in classification or low loss in regression setting problems) following the exploitation discipline and **(b)** where the prediction uncertainty is high following the exploration discipline. Obviously, locations with both characteristics are sampled first. Another interesting observation is that the area on the left remains unsampled with a low score in acquisition function, as while it has high uncertainty, correctly predicted that it will not offer any improvement over the highest observation. Lastly, as the number of observations grows, the posterior distribution (or the surrogate function) improves, and the algorithm becomes more certain of which regions in parameter space are worth exploring and which are not. Therefore, Bayesian optimization can bring down the time spent to get to the optimal set of parameters and bring better generalization performance on the test set making it an excellent choice for hyperparameter tuning.

4. APPLICATION A: CRYPTOCURRENCY PRICE PREDICTION FOR BTC, LTC

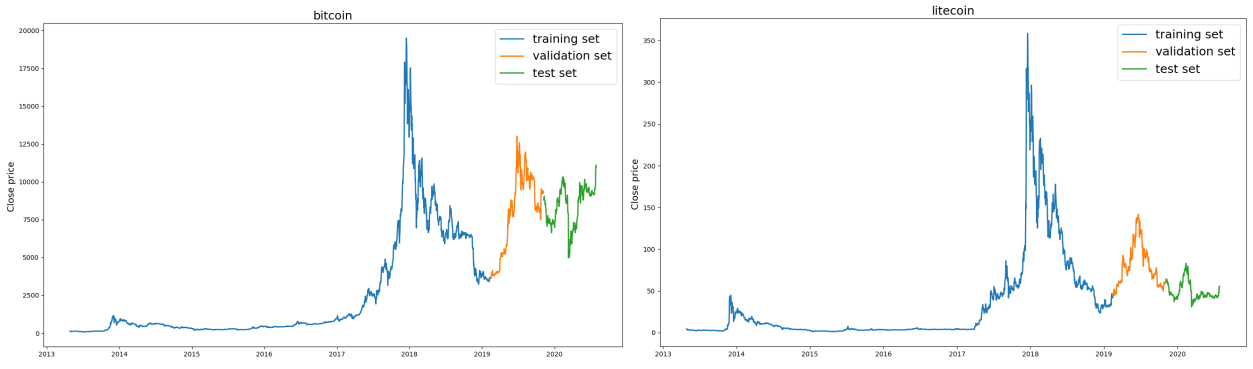
4.1 Objective

For this application, our goal is quite straightforward and is to predict the next day's OHLC [1] cryptocurrency price in USD for BTC and LTC. Hence, for the purpose of this task a plethora of models were trained in a range of different datasets. In the context of this thesis, I chose to mention solely the ones that offered the best results in accordance with their simplicity under Occam's razor law.

The selection of Litecoin [2] wasn't chosen randomly. It provided quite good results when Bitcoin and Litecoin were forecasted combined. This probably occurs due to its **(a)** high correlation with bitcoin in accordance with the Pearson correlation coefficient method, where the fluctuation of the one coin affects the other **(b)** and the fact that for both assets we have enough data, since they were the first that were released (2009 and 2011 respectively).

4.2 Dataset

This section introduces the data that was used to train and test all models. This includes preprocessing of data such as scaling and normalization. We used a dataset consisting of daily opening and closing prices, as well as daily high and low prices. In addition we have included the Market Capitalization and Volume indicators for both BTC and LTC. The data contains 2650 samples in total and are ranging from 2013-04-28 to 2020-07-29.



(a) BTC Close price (b) LTC Close price

Figure 4.1: Training, validation and test sets for BTC and LTC prices in line graph form

We should note that many different datasets for the task were utilized such as Blockchain features (captured from CoinMetrics.io), other correlated cryptocurrencies, dimensionality reduction technique (PCA) and technical indicators but many of them didn't surpass the results of the presented dataset or the results showed a minimal improvement that couldn't justify the additive complexity.

[1]OHLC stands for Open-High-Low-Close

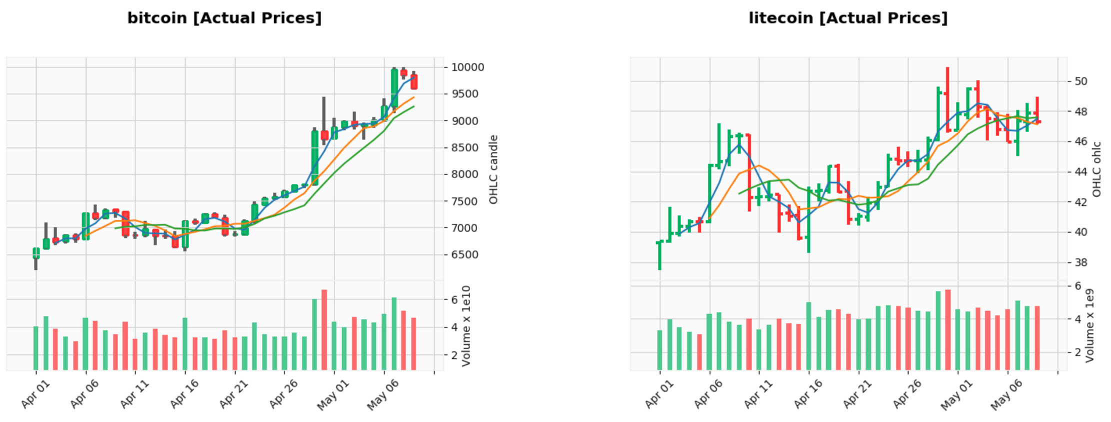
[2]Founder's intention behind Litecoin was to create a "lite version of Bitcoin," and its developers have always stated that Litecoin can be seen as the "silver" to Bitcoin's "gold"

Table 4.1: Training, validation and test sets for application A

	Time Period
Training	from 2013-04-30 to 2019-02-17
Validation	from 2019-02-18 to 2019-11-08
Test	from 2019-11-09 to 2020-07-29

The data is split into a training set, a validation set and a final test set following the holdout procedure as discussed in the previous chapter. We train the models on the training set and test their performance on the validation set, where we select the best hyperparameter configuration that fits the data and check the generalization capability in the test set. To train, validate and test models for time series prediction, we must maintain the temporal structure of the data to avoid any forward-looking bias. The training set must therefore be chronologically before the test set. The data is split as depicted in Table 4.1.

It is important to note here that we apply a sliding window with stride 1 and length K = 3 (is also called lag) in the dataset, where K is the number of previous time steps to use as input variables to predict the next time period, which in our case are the daily crypto prices.



(a) Candlestick chart for BTC

(b) OHLC chart for LTC price

Figure 4.2: Data Visualization charts with volume and moving average(3,6,9)

Data normalization

Choosing the type of method for normalizing a time series, especially financial ones is never easy. In most cases, the data take relatively large values, or are heterogeneous (referring to time-series that have different scales). If we feed these kinds of data into our NN, they can potentially trigger large gradient updates that will prevent the network from converging. To make learning easier for the network and speed up the training process, data should **(a)** take small values **(b)** be homogeneous (all features should take values at roughly within the same range). Therefore, normalization in our data should be applied. The most common normalization methods used during data transformation include:

- Min-Max Scaling, where the data inputs are mapped on a number from 0 to 1:

$$x_{new} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

- Mean Normalization, which makes data have values between -1 and 1 with a mean of 0:

$$x_{new} = \frac{x-\mu}{x_{max}-x_{min}}$$

- Z-Score (Standardization), where the features are redistributed with their mean of 0 and standard deviation of 1:

$$x_{new} = \frac{x-\mu}{\sigma}$$

, where x_{min}, x_{max}, μ, σ stand for minimum and maximum data points, the sample mean and standard deviation respectively.
For our problem, we use Min-Max Scaling and adjust features on a scale from 0 to 1 given that most of our time-series have a peak, therefore we could argue we know the maximum of the series, in which case Min-Max Scaling does a good job.

4.3 Models

As with the datasets, the same policy was followed for the model architecture selection with the many different models that were tested. Therefore, following the experiments, I decided to include only those with the best performance and those that showed no overfitting issues. Within the neural networks, I decided to exclude the LSTM Autoencoder and Composite LSTM Autoencoder [34] as well. Since I had opted for a low-number feature dataset, there wasn't any need for the dimensionality reduction that they provide. Interestingly, the LSTM Autoencoder, due to the 2 decoders that consist of (reconstructing the input and predicting the future) had better performance than a plain Autoencoder or an LSTM with PCA in high dimensional datasets.

4.3.1 LSTM

A plain LSTM was used for our baseline model of the list, where the hyperparameter configuration was chosen by hand after some custom trial and error and tuning. The general architecture of the model consists of an input layer of dimension (time steps × number of features = 3 × 8), the inner structure and an output layer of dimension (time steps × number of prices to predict = 1 × 8). Indeed, we apply at each future time step a ReLU layer with eight neurons, one OHLC vector for each of the two crypto. We change the inner structure of the model to test different architectures of neural networks. The inner structure in our base model consists of 3 layers with 50 neurons in each one. To improve the generalization capability of the model, after each layer a Dropout layer is immediately followed with a rate of 20% to combat potential overfitting issues during training. We also utilized the early stopping generalization technique with 20 epochs patience, meaning that if the validation loss for the 20 epochs in a row is not improved, then the model training process halts. Last but not least, the batch size was set to 32 and the Adam optimizer was chosen with a 0.001 learning rate. The architecture of the model in plot format is shown in Appendix Figure A.1.

```
 ...
early_stopping_monitor = EarlyStopping(monitor='val_loss',
                   mode='min',patience=20, restore_best_weights=True)
# design network
model = Sequential()
model.add(LSTM(units=50, input_shape=(train_X.shape[1], train_X.shape[2]),
    return_sequences=True))
model.add(Dropout(0.2))
# Adding a second LSTM layer and some Dropout regularization
model.add(LSTM(units=50, input_shape=(train_X.shape[1], train_X.shape[2]),
    return_sequences=True))
model.add(Dropout(0.2))
# Adding a third LSTM layer and some Dropout regularization
model.add(LSTM(units=50))
model.add(Dropout(0.2))
model.add(Dense(units=num_features_to_predict))

model.compile(loss='mse', optimizer='adam')

# fit network
history = model.fit(train_X, train_y, epochs=500, batch_size=32,
                        validation_data=(val_X, val_y),shuffle=False, callbacks=[
    tensorboard, early_stopping_monitor, model_checkpoint_callback])
```

Listing 4.1: snippet code of LSTM implemented in Keras

4.3.2 Bayesian optimized LSTM

As discussed in theory a great alternative for finding a good hyperparameter combination is the Bayesian Optimization technique. Before we begin with the actual search for hyperparameters, we first need to define the valid search ranges for each of these parameters. The hyperparameters with their selection thresholds that were set for tuning are the following:

- learning rate: $[10^{-6}, 10^{-2}]$
- layers: $[2, 5]$
- number of units in each layer: $[5, 512]$
- dropout rate: $[0.1, 0.9]$
- batch size: $[5, 100]$

The flowchart for finding the best configuration is shown in Figure 4.3.

The process was run for 50 iterations and it took around 2 hours in Google Colab to finish. The Table B.1 in Appendix shows all the hyperparameters tried by the Bayesian optimizer and their associated validation MSE loss sorted so the best models are shown first.

Since we can not infer much information from a plain table, showing the Partial Dependence plots in Figure 4.4 seem a much better way of visualizing high-dimensional spaces and indicate a better intuition.

The plot on the left shows the last surrogate model built by the Bayesian optimizer where yellow regions are better and blue regions are worse. The black dots show where the

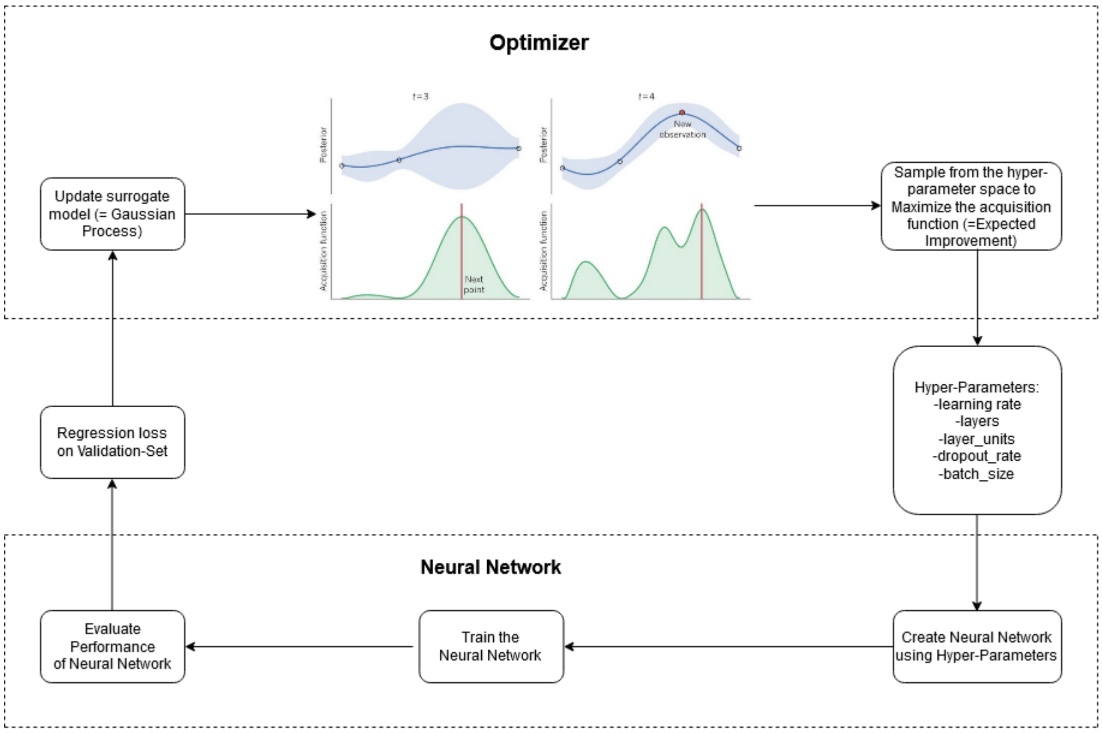

Optimizer
Update surrogate model (= Gaussian Process)
Sample from the hyper-parameter space to Maximize the acquisition function (=Expected Improvement)
Hyper-Parameters: -learning rate -layers -layer_units -dropout_rate -batch_size
Regression loss on Validation-Set
Neural Network
Evaluate Performance of Neural Network
Train the Neural Network
Create Neural Network using Hyper-Parameters
Posterior
Acquisition function
Next point
New observation

Figure 4.3: Flowchart of the Bayesian optimization algorithm at a superficial level

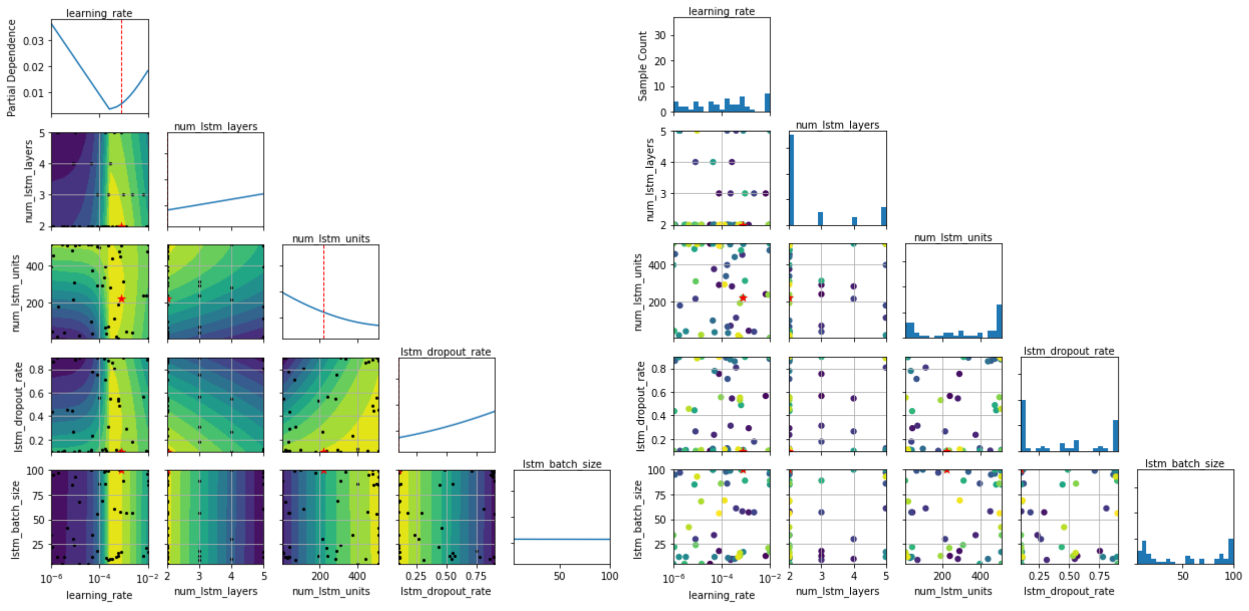


Figure 4.4: Partial Dependence plots for the 50 hyperparameter configurations tried by Bayesian optimizer

optimizer has sampled the search space and the red star shows the best parameters found.

On the right is another type of matrix plot. Here the diagonal shows histograms of the sample distributions for each of the hyperparameters during Bayesian optimization. The plots below the diagonal show the location of samples in the search space and the colour-coding shows the order in which the samples were taken. Even though we have only 50 samples, we can likely see that the samples eventually become concentrated in a certain region of the search space, where it chooses to exploit rather than explore.

From the first partial Dependence plot we can clearly observe that regardless of the value of the other hyperparameters, the learning rate region that produces the best results is between 10^{-4} and 10^{-3}. Another interesting inference we can derive from both Table B.1 and the second matrix-plot, is that the Bayesian framework shows a preference for 2 LSTM layers instead of more, as opposed to the base hand-tuned model, which has 3 layers. This makes sense, given that the best 16 out of the 50 models have 2 layers.

After the 50 iterations the best model was shown to be the one with the following hyperparameters:

- learning rate: 0.00076
- layers: 2
- number of units in each layer: 222
- dropout rate: 0.1
- batch size: 100

The architecture of the model is also shown in Figure A.1.

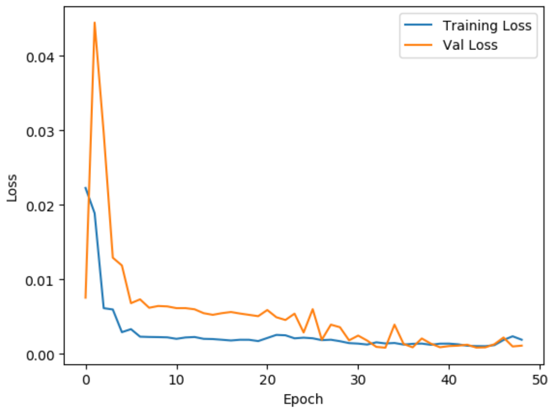
4.3.3 LSTM-FCN

A promising model that seems to learn the representation of the data efficiently in various complex classification tasks is the LSTM-FCN[3] and its variations with attention mechanism LSTM-FCN [35]. The proposed models have been tested on all 85 UCR time series datasets and have shown to outperform most of the well-known models providing state-of-the-art results, while requiring minimum preprocessing, time and memory consumption during training time. The aforementioned benefits, combined with the wide ranging applicability on several sequence modelling tasks, make it appealing for our price forecasting problem. For the scope of this thesis we will use the LSTM-FCN variation. LSTM-FCN model consists of 2 parts: **(a)** a LSTM block with a high rate of dropout to combat overfitting and **(b)** a FCN part with 3 temporal convolution layers, followed by batch normalization. The output of the global pooling layer and the LSTM block is concatenated and passed onto a Dense layer output layer of dimension (timesteps × number of prices to predict = 1 × 8). To convert the model from a classification task to a regression one, the last activation function was set to ReLU and the cross entropy loss function was changed to the MSE. Other than that, the early stopping generalization method was added with patience=20 to prevent overfitting. Lastly, a trivial change from the proposed model took place, which is the reduction of the dropout rate by 10% (from 80% to 70%). All the other hyperparameters remained the same as was proposed in the research work. The architecture of the model is shown in Figure A.2 in the Appendix.

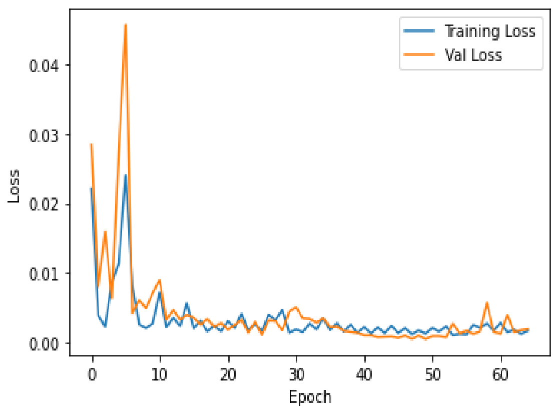
[3]FCN stands for Fully convolutional neural networks

4.4 Results

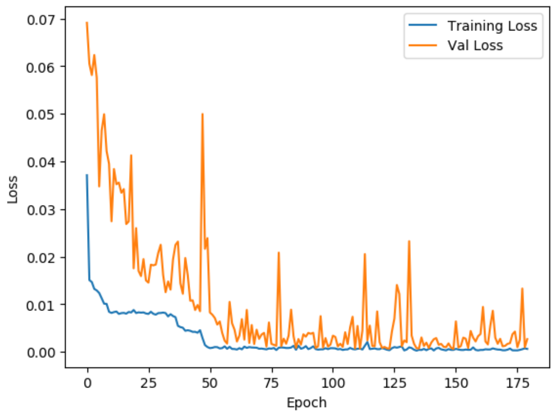
4.4.1 Loss Plots



(a) LSTM



(b) Bayesian optimized LSTM



(c) LSTM-FCN

Figure 4.5: Loss plots in each NN

We can observe that in all of our models both training and validation loss are steadily decreasing, as it is the gap between them. This is a clear indication that the models are not overfitting the dataset. An interesting finding is the fluctuation of the validation loss in the LSTM-FCN model. A guess concerning this phenomenon is the high dropout rate in the LSTM block. However, even with the high frequency peaks and lows, the model is still able to learn and produce quite a good representation of the data. To get the best model weights out of the training process, ModelCheckpoint callback from the Keras library was applied to all the 3 models. Rather than saving the weights of the last model in our training process, we monitor the model after each epoch and we only keep the one that has achieved the ”best performance”. For our research purposes, the best model is the one that achieves the minimum Mean Square Error in the validation set.

4.4.2 Comparing ARIMA

To support the claim that the Deep Learning models provide better results than the classical statistical models, a non-seasonal Autoregressive integrated moving average (ARIMA) model was also created for forecasting the Closing price of the Bitcoin. ARIMA models are generally denoted ARIMA(p,d,q) where parameters p, d, and q are non-negative integers, **(a)** p stands for the number of time lags of the autoregressive model, **(b)** d is the degree of differencing (the number of times the data have had past values subtracted), and **(c)** q is the order of the moving-average model. The best parameters were set by the Grid Search tuning technique and the combination (0,2,1) gave the best performance. Not surprisingly, the MSE for the Closing price in ARIMA was lower than the other DL models. Specifically, the MSE error in real data in ARIMA was 110364.988 whereas the baseline hand-tuned LSTM achieved much better results with a score of 108227.23. Hence, given the better results that DL models provide, for our evaluation, we are focusing solely on the networks that were presented in section Models.

4.4.3 Evaluation metrics

In this section we measure the performance of the models on the test set with different evaluation metrics. It's very important to check the quality of the models with the use of multiple evaluation metrics. This is because a model may perform well using one measurement from one evaluation metric, but may show poor performance using another measurement from another evaluation metric. Therefore, inspecting different metrics is critical in ensuring that our models are operating correctly and optimally. In regression setting problems, the most common metrics that also seem appropriate for the evaluation are the MSE, which is also the loss function of our models, RMSE, MAE, MAPE and MPE.

We consolidate the evaluations in three metric evaluation tables starting from top to bottom. Table 4.2 presents the averaged losses of the OHLC daily predictions including both Bitcoin and Litecoin cryptocurrencies. Table 4.3, on the other hand, separates the cryptocurrencies and evaluates them individually. Table 4.4, which is the 3rd and last one, breaks down the previous table and measures the MSE loss for every OHLC entity.

Table 4.2: Average performance metrics in both BTC, LTC on the test set

Network	MSE	RMSE	MAE	MAPE (%)	MPE (%)
LSTM	0.00083	0.02895	0.02308	14.4	-12.57
Baeysian opt. LSTM	**0.00028**	**0.01676**	**0.0116**	**5.32**	**1.31**
LSTM-FCN	0.00056	0.02366	0.01857	7.58	4.48

Table 4.3: Average performance metrics in OHLC prices for each BTC, LTC on the test set

Cryptocurrency	Network	MSE	RMSE	MAE	MAPE (%)	MPE (%)
BTC	LSTM	**0.00042**	**0.02054**	**0.01381**	**3.43**	0.21
	Baeysian opt. LSTM	**0.00042**	0.02056	0.01383	**3.43**	**−0.12**
	LSTM-FCN	0.00092	0.03041	0.02563	6.15	5.78
LTC	LSTM	0.00125	0.03542	0.03235	25.37	-25.36
	Baeysian opt. LSTM	**0.00013**	**0.01179**	**0.00937**	**7.21**	**2.73**
	LSTM-FCN	0.00019	0.01398	0.01151	9.01	3.18

Several interesting conclusions can be extracted. From the first table, it is obvious that the Bayesian optimized LSTM is overall the best model among the three. In the second

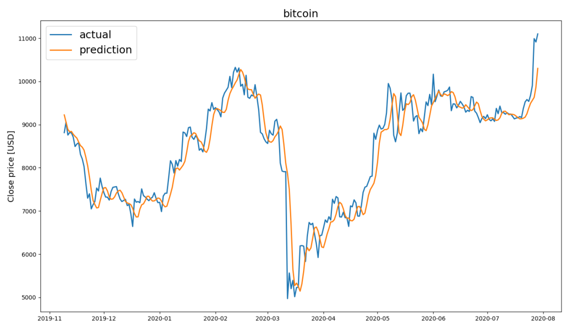
Table 4.4: MSE loss for all the OHLC of the models on the test set for BTC and LTC

	BTC				LTC			
Price / NN	Open	High	Low	Close	Open	High	Low	Close
LSTM	0.00028	0.00036	**0.00053**	**0.00051**	0.00058	0.0011	0.00184	0.00149
Baeysian opt. LSTM	0.00028	**0.00033**	0.00056	**0.00051**	0.00022	**0.0001**	**0.00012**	**0.00011**
LSTM-FCN	**0.00009**	0.00118	0.00095	0.00147	**0.00012**	0.00015	0.0002	0.00031

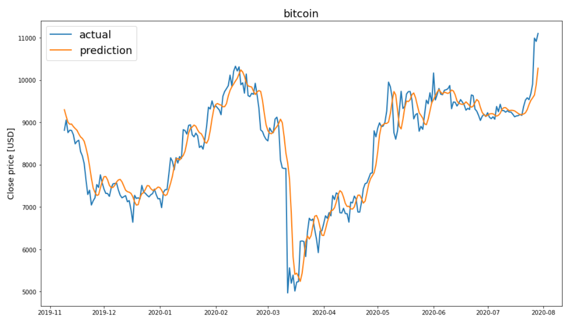
table, we observe that, overall, the LSTM is the model with the worst performance as Table 4.2 reveals, basically due to the systematic poor performance in predicting Litecoin's daily price. In contrast, for the plain LSTM, the performance in Bitcoin along with the Bayesian optimized network is excellent. In the third and last table, the LSTM-FCN model appears to display exceptionally good predictions for the daily Opening prices in both Bitcoin and Litecoin. We can easily deduce that each network has its own strong and weak spots beyond the Bayesian optimized LSTM, which seems to provide consistently good results.

4.4.4 Prediction plots

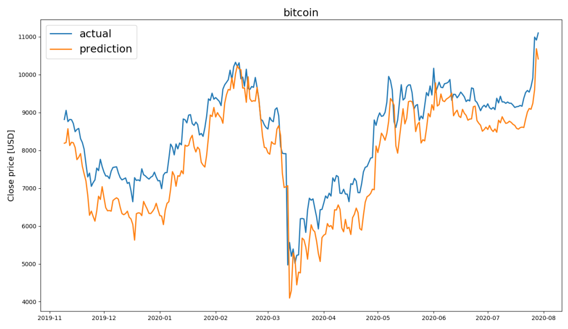
In the face of coronavirus pandemic, the Bitcoin and cryptocurrency markets have fallen sharply over the last few weeks, plummeting along with traditional markets. While LSTM-FCN didn't produce the best results for bitcoin according to the evaluation metrics, in the plots it appears that predicted surprisingly well the sudden plunge of bitcoin in the period March 12-14 2020 in comparison to the other LSTM models.



(a) LSTM

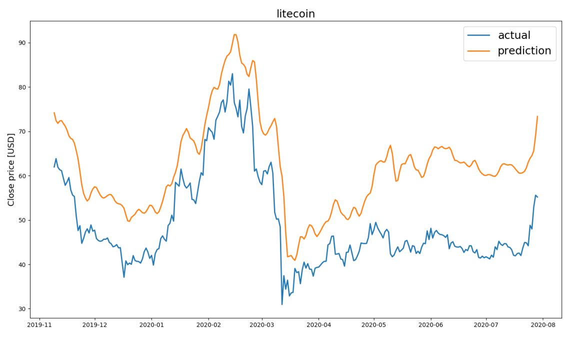


(b) Bayesian opt. LSTM

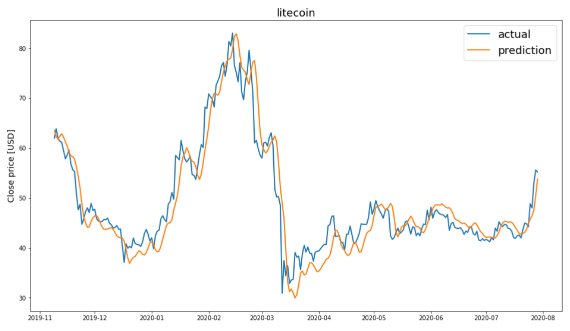


(c) LSTM-FCN

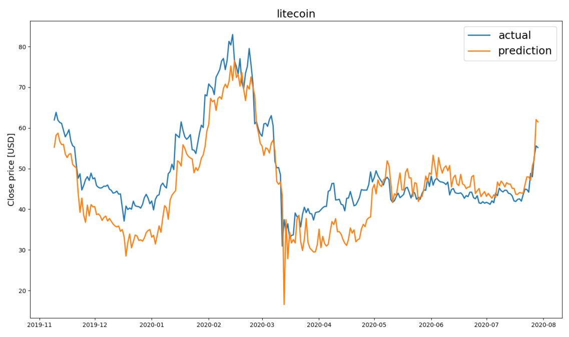
Figure 4.6: Closing Price Prediction for BTC in test set



(a) LSTM

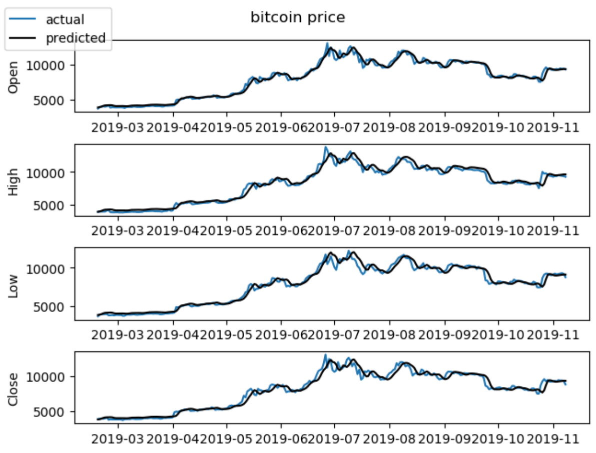


(b) Bayesian opt. LSTM

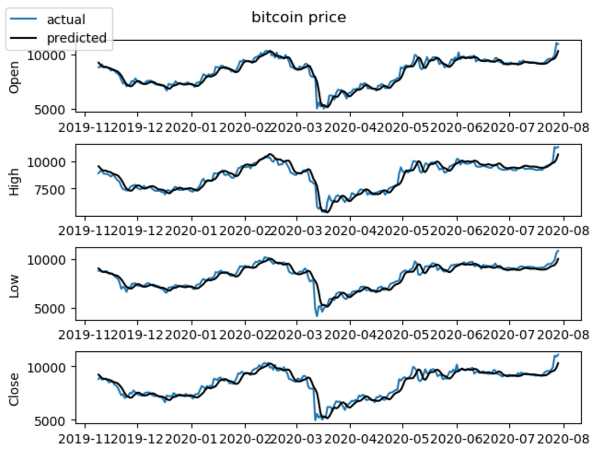


(c) LSTM-FCN

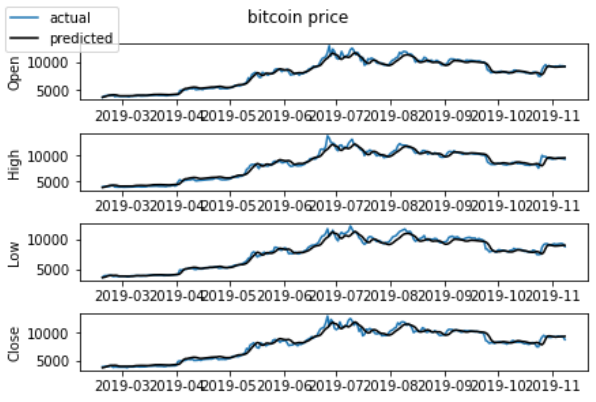
Figure 4.7: Closing Price Prediction for LTC in test set



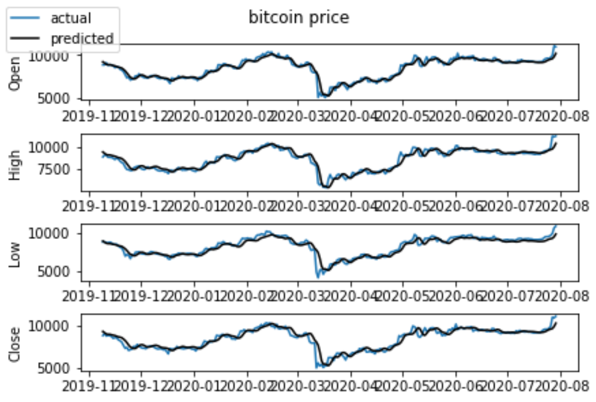
(a) LSTM in val. set



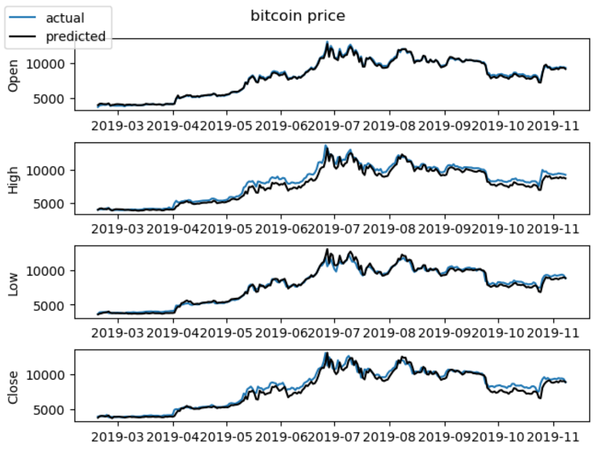
(b) LSTM in test set



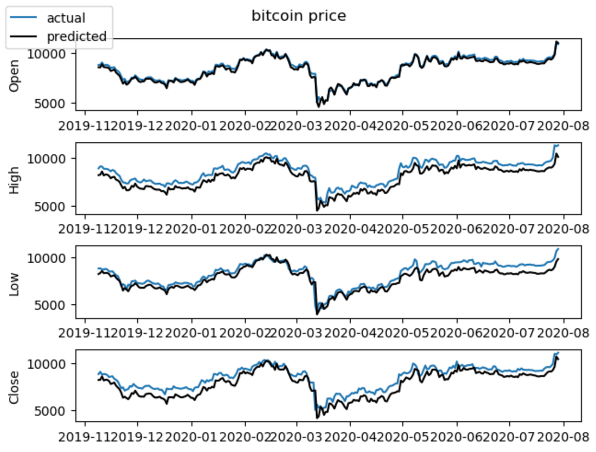
(c) Bayesian opt. LSTM in val. set



(d) Bayesian opt. LSTM in test set

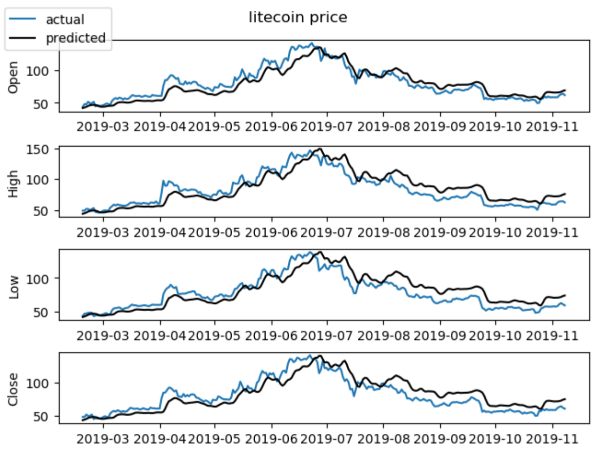


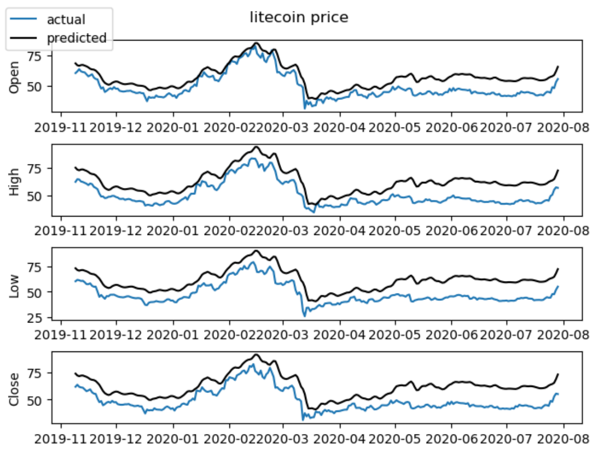
(e) LSTM-FCN in val. set

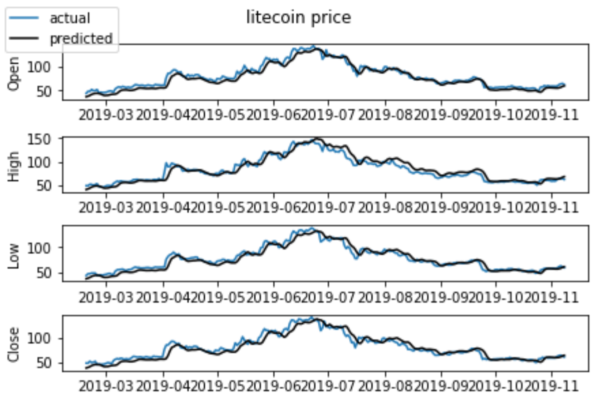


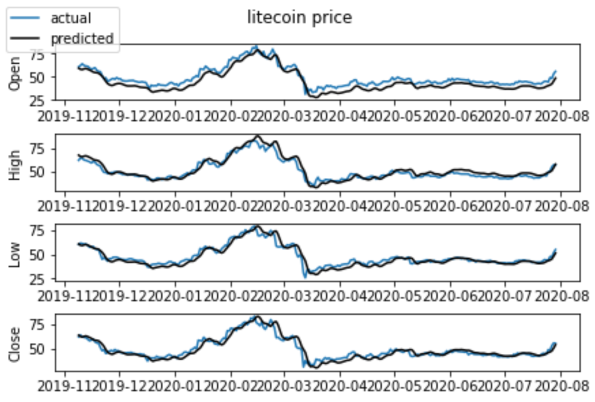
(f) LSTM-FCN in test set

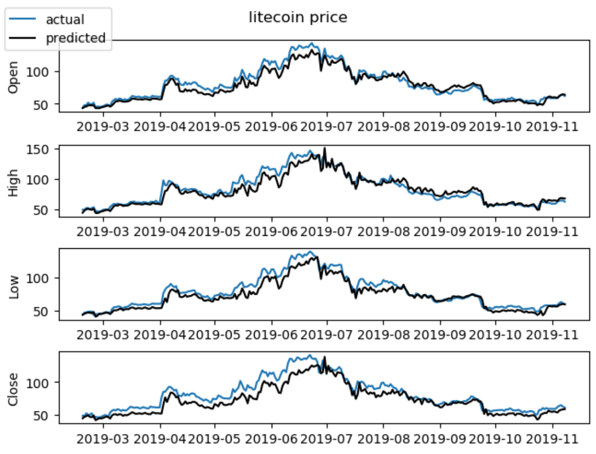
Figure 4.8: BTC OHLC predictions


(a) LSTM in val. set


(b) LSTM in test set


(c) Bayesian opt. LSTM in val. set


(d) Bayesian opt. LSTM in test set


(e) LSTM-FCN in val. set

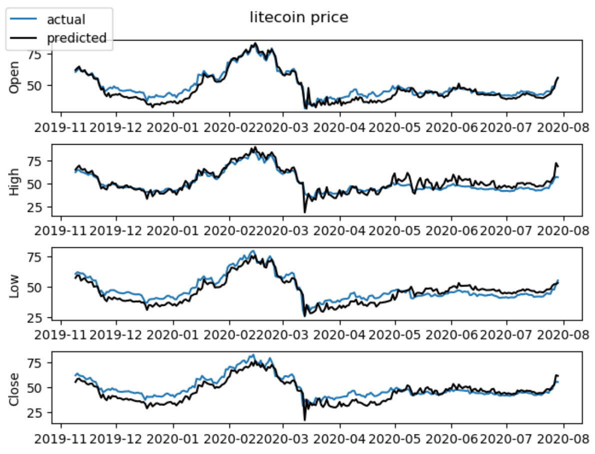

(f) LSTM-FCN in test set

Figure 4.9: LTC OHLC predictions

5. APPLICATION B: ESTIMATING ONE-STEP PREDICTION INTERVALS

5.1 Objective

This chapter introduces the methodology that was used to apply Bayesian neural networks to the problem of estimating prediction intervals for Bitcoin's daily Closing price. The goal here is not to find the best price predictions as it was in the first application, but to estimate sufficient upper and lower price range levels. For this purpose, we first introduce the data that was used, along with the various techniques for preprocessing that were necessary. After this, we introduce our BNN model that was tuned by the Bayesian optimization technique, with respect to the mean MSE loss for maximum results and lastly, we present the results of our model.

5.2 Dataset

For this application, the data differ from the previous one, since here we include also some well-known technical indicators for Bitcoin and a small portfolio of correlated cryptocurrencies as features. Even in this application, blockchain features didn't improve the model's performance so were excluded for simplicity. The data is split into a training set, a validation set and a final test set following the holdout procedure. The rolling window has size 3 and as far as normalization is concerned, a Min-Max Scaling approach was applied.

Table 5.1: Training, validation and test sets for application B

	Time Period
Training	from 2017-07-25 to 2019-12-24
Validation	from 2019-12-25 to 2020-04-11
Test	from 2020-04-12 to 2020-07-29

Our final dataset consists of 1100 samples in contrast to 2650 samples in the first application's dataset. A drawback in adding more cryptocurrencies with later release dates is the shrinking of the dataset since our policy is to eliminate the days where we don't have all the values available. This is a trade-off that we took into consideration while building the dataset.

5.2.1 Correlated assets

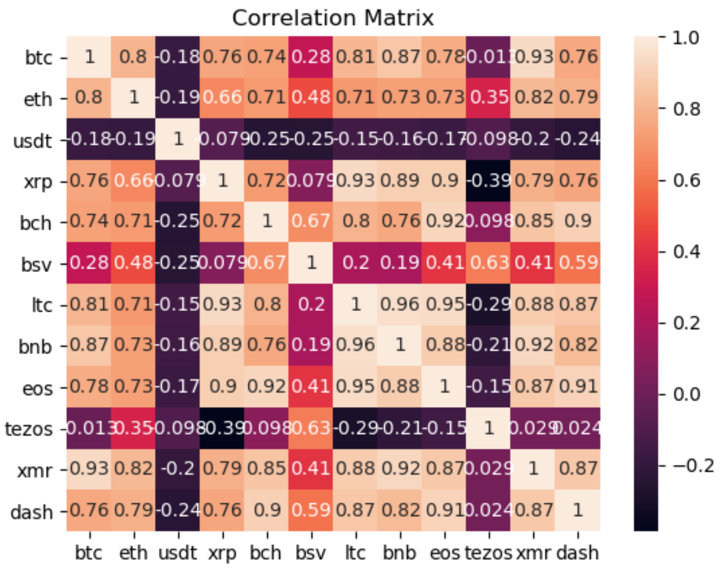
Unlike Modern portfolio theory (MPT), where the goal is to achieve diversification and reduce the correlation between the returns of the assets selected for the portfolio, in our dataset we desire to build a basket of correlated cryptocurrencies. To calculate that we will use from statistics the Pearson correlation coefficient. The Pearson Correlation Coefficient is a very useful measure of finding out the correlation between two random variables (in our case cryptocurrencies) by summarizing the strength of their linear relationship over a specific period of time.

The correlation coefficient (ρ) is determined by dividing the covariance by the product of the two variables' standard deviations. So, given a pair of random variables (X, Y), the formula for ρ is:

$$\rho_{X,Y} = \frac{cov(X,Y)}{\sigma_X \sigma_Y} \tag{5.1}$$

where $cov(X, Y)$ denotes the covariance of the variables X and Y and σ_X, σ_Y the respective standard deviations.

The value of the correlation coefficient ranges between -1 and +1, due to the normalization of the covariance. Values close to 1 indicate a strong positive linear correlation between X and Y, while values close to -1 indicate a strong negative correlation. In the same manner, values close to 0 show neutral and therefore no correlation between the random variables.



	btc	eth	usdt	xrp	bch	bsv	ltc	bnb	eos	tezos	xmr	dash
btc	1	0.8	-0.18	0.76	0.74	0.28	0.81	0.87	0.78	-0.013	0.93	0.76
eth	0.8	1	-0.19	0.66	0.71	0.48	0.71	0.73	0.73	0.35	0.82	0.79
usdt	-0.18	-0.19	1	-0.079	-0.25	-0.25	-0.15	-0.16	-0.17	-0.098	-0.2	-0.24
xrp	0.76	0.66	-0.079	1	0.72	0.079	0.93	0.89	0.9	-0.39	0.79	0.76
bch	0.74	0.71	-0.25	0.72	1	0.67	0.8	0.76	0.92	0.098	0.85	0.9
bsv	0.28	0.48	-0.25	0.079	0.67	1	0.2	0.19	0.41	0.63	0.41	0.59
ltc	0.81	0.71	-0.15	0.93	0.8	0.2	1	0.96	0.95	-0.29	0.88	0.87
bnb	0.87	0.73	-0.16	0.89	0.76	0.19	0.96	1	0.88	-0.21	0.92	0.82
eos	0.78	0.73	-0.17	0.9	0.92	0.41	0.95	0.88	1	-0.15	0.87	0.91
tezos	-0.013	0.35	-0.098	-0.39	0.098	0.63	-0.29	-0.21	-0.15	1	0.029	0.024
xmr	0.93	0.82	-0.2	0.79	0.85	0.41	0.88	0.92	0.87	0.029	1	0.87
dash	0.76	0.79	-0.24	0.76	0.9	0.59	0.87	0.82	0.91	0.024	0.87	1

Figure 5.1: Cryptocurrencies Correlation Matrix

To test the correlation between Bitcoin with other cryptocurrencies, a correlation matrix was built. A correlation matrix investigates the dependence between multiple variables at the same time. The result is a table containing the correlation coefficients between each cryptocurrency and is shown in Figure 5.1.

Only the coins that satisfied the 3 criteria were included in our dataset. Those that **(a)** were in the top 20 by Market Capitalization **(b)** were released not after 2018 **(c)** had a coefficient correlation with bitcoin above 0.75 or below -0.75.

Following that rule, aside from Bitcoin, the virtual coins correlated portfolio is composed of **ethereum (eth), litecoin (ltc), binance-coin (bnb), eos (eos), monero (xmr)**. For each of these coins, their Closing price, Volume and Market Capitalization were added as features in our dataset. As an exception only for Bitcoin, we also incorporated the Opening, High, Low daily prices.

5.2.2 Technical Indicators

Before feeding our dataset in our network, technical indicators were created for Bitcoin's Closing price. Technical indicators can be useful variables for time series modeling. In the forecast of cryptocurrencies prices, we decided to use some major technical indicators with a different days setting.

Let S_t be the Closing price of Bitcoin at time t. We define the simple moving average of order q as:

$$MA = \frac{1}{q}\sum_{t=1}^{q} S_t \tag{5.2}$$

and the rolling standard deviation of order q as:

$$\sigma = \sqrt{\frac{1}{q}\sum_{t=1}^{q}(S_t - MA)^2} \tag{5.3}$$

Now we are able to define Bollinger bands' volatility indicator. The three components of Bollinger bands correspond to MA, the central band, MA + $\delta\sigma$, the upper band and MA - $\delta\sigma$ the lower band, where δ is a fixed parameter typically equals 2.

Both Moving Average and Bollinger Bands features are shown in Figure 5.2.

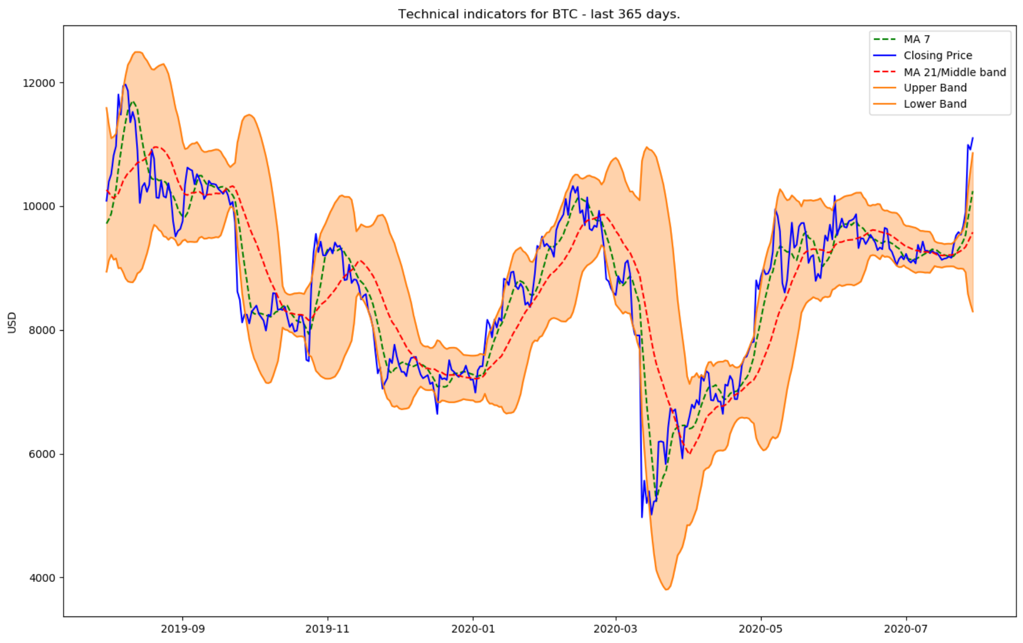


Figure 5.2: Dataset features: Bitcoin Closing prices (blue), its 7 and 21-days moving average (dashed green and red lines) and its lower and upper Bollinger bands (light orange)

Even though we often consider recent observations to be more relevant, the simple moving average weights all observations equally. Conversely, the exponential moving average decays the weights of older observations. This can be defined recursively as:

$$EMA = \alpha S_t + (1 - \alpha) EMA_{prev} \tag{5.4}$$

for any $\alpha \in [0, 1]$.

The 12-day and 26-day exponential moving averages (EMAs) are often the most quoted and analyzed short-term averages. They are also used to create indicators like the moving average convergence divergence (MACD). The MACD is calculated by subtracting the 26-period Exponential Moving Average (EMA) from the 12-period EMA. All these feautures were also included in our dataset. As a last feature, the percentage change between the current and its prior price was incorporated in our dataset.

To sum up, the newly created features in our dataset are: **(a)** a 7 and 21-day MA, **(b)** the Bollinger bands volatility indicator with 21-day MA and δ=2, **(c)** a MACD(26,12) trend indicator and **(d)** the percentage difference in a 1-day span. Besides these, as in the previous dataset this one also includes Bitcoin's Market Capitalization and Volume.

5.3 Bayesian Neural Network model

The Dropout NN model does not require any alteration from a standard frequentist neural network, thus all methodology presented in the previous section also applies here with minor changes. We only add L2 weight regularisation term which corresponds to choosing a prior. Modeling uncertainty with Monte Carlo dropout works simply by running multiple forward passes through the model with different dropout masks every time.

To tune the model, given its superiority in application A, I decided to opt for Bayesian optimization to obtain the model setting with the minimum MSE in the validation set. For each prediction 50 forward stochastic passes were made, leading to 50 different predictions. By computing the average and the variance of this sample we get an ensemble prediction. For the evaluation we compare the MSE between the real data and the ensemble averaging of the predicted data. The process was run in Google Colab for 100 iterations and took roughly 2.15' hours to finish. This time, the dropout rate was excluded from the process and the probability was set to p=0.1. In addition, for an appropriate setting, weight-decay λ and prior length-scale l were added as hyperparamaters in our Bayesian optimization framework. More information about them can be found in the paper [29]. It is a big relief that these are the only parameters to optimize when using the MC Dropout approach, considering the fact that sampling methods like MCMC and HMC usually require many parameters to optimize.

The Table C.1 in Appendix presents the top 50 tuned models out of the 100 sorted w.r.t their MSE loss. A better intuition could be inferred though, by looking at the partial Dependence plots that are shown in Figure 5.3.

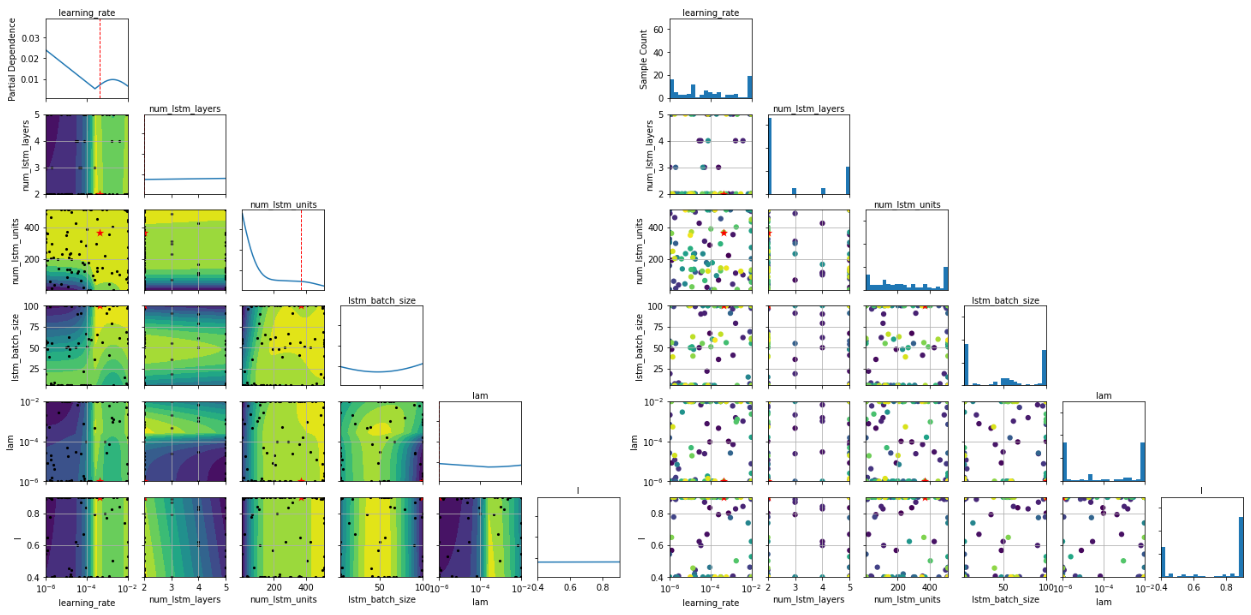


Figure 5.3: Partial Dependence plots in matrix format

The best hyperparameters configuration resulting from Bayesian optimization is the following:

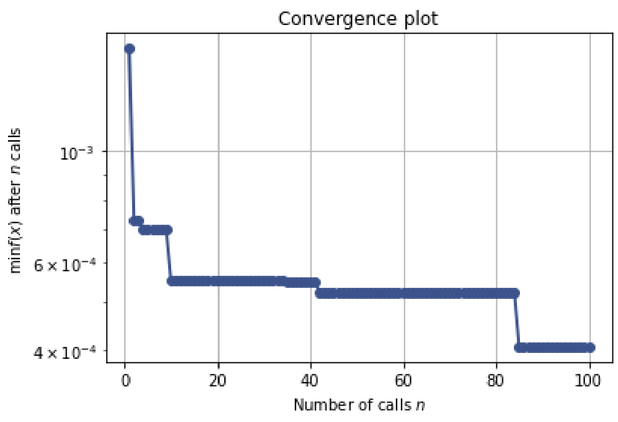
- learning rate: 0.0004489
- layers: 2
- number of units in each layer: 369
- batch size: 100
- λ: 0.000001
- l: 0.9

The architecture of the model is also depicted in Appendix Figure A.3 in a plot format.

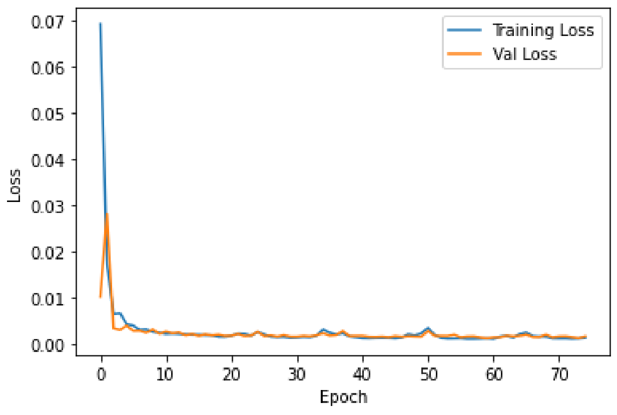
5.4 Results

5.4.1 Loss plot

In the left subfigure, we can observe the loss of the different models that Bayesian optimizer tested, whereas in the right we observe the loss of our best model that the optimizer chose. It started with a setting that had a validation error of an order of magnitude of over -3 and it ended up after 100 iterations, with a model with an order of magnitude of -4 and specifically a validation loss of 47×10^{-4}. Interestingly, after 80 iterations the optimizer seems to have found a good hyperparameter subspace with models similar to each other in performance.



(a) Convergence after 100 iteration in Bayesian opt. framework



(b) BNN's loss plot

Figure 5.4: Loss plots of Bayesian optimized BNN

5.4.2 Prediction plots

Presented below are our prediction intervals that were estimated from our BNN model with a standard deviation multiplied by factors of 1, 1.5 and 2 respectively. Unfortunately, our test dataset includes only 108 samples, making it difficult to infer safe assumptions of how well our model estimates the prediction intervals. However, we will make an effort to extract some interesting information from the results.

Clearly, in Figure 5.5 with the strict configuration of +/- the standard deviation from the $\hat{y}_{mean}$, we can observe that on days with small volatility, the predicted intervals correctly contained the price. Yet, the intervals fail to predict the price range in sharp, extreme movements. A solution is to multiply the predicted volatility by factors more than one (like the Bollinger Bands). Indeed, the multiplying factors of 1.5, 2 increase the correct predictions at the cost of loosened price prediction ranges. It's a trade-off that has to be made. Tighter prediction intervals means higher risk of failure, whereas broader ranges means looser price prediction thresholds. It's up to the user to choose which type of intervals suits him better.

One last point that I would like to outline is that prediction intervals could also be used for the detection of outliers. Anomaly detection and volatility estimation are key components in financial markets and are even more essential now in the cryptocurrency markets, given their highly volatile nature. A policy that could be followed, for example, is that every price that exceeds the predicted ranges in $1.5 \times \sigma$ configuration will be categorized as high movement, whilst every price that exceeds the predicted barriers in $2 \times \sigma$ setting will be categorized as extreme movement. Extreme movements often create bargains in the markets. High and extreme price movements, consequently, could trigger a trading signal to an agent to take action. A simple policy could be for example, the oft heard "Buy Low, Sell High" Strategy. Of course, this project was not made for investment and trading purposes. Nevertheless, predicting the market is very risky and a realistic investment system should be implemented by taking into account the active environment which is ever-evolving.

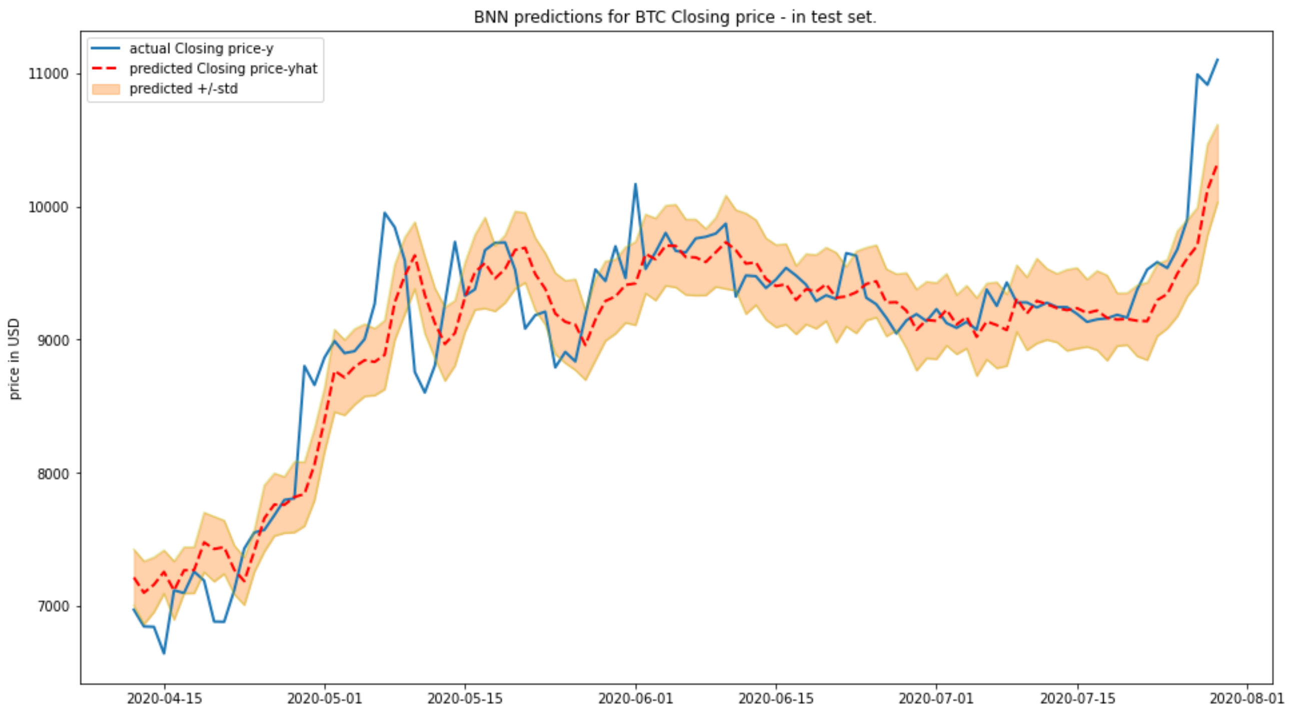


Figure 5.5: BTC Closing price prediction intervals with +/-σ

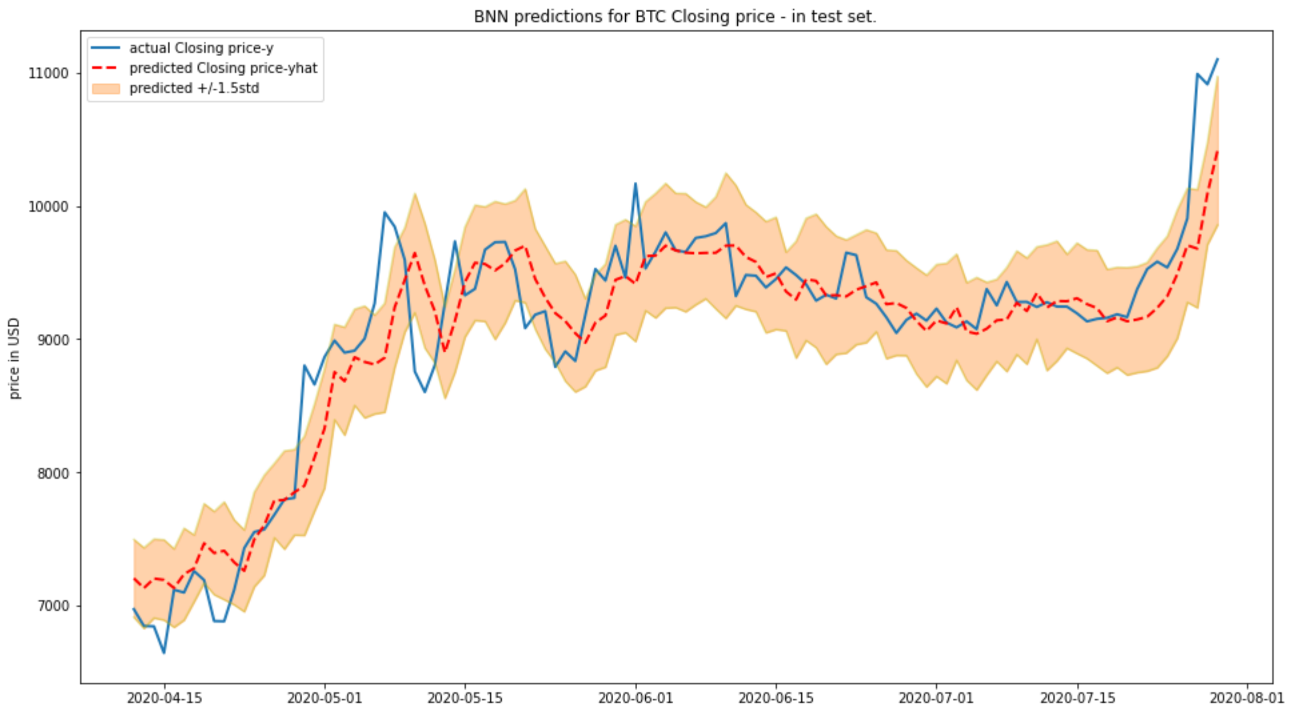


Figure 5.6: BTC closing price prediction intervals with +/-1.5$\times\sigma$

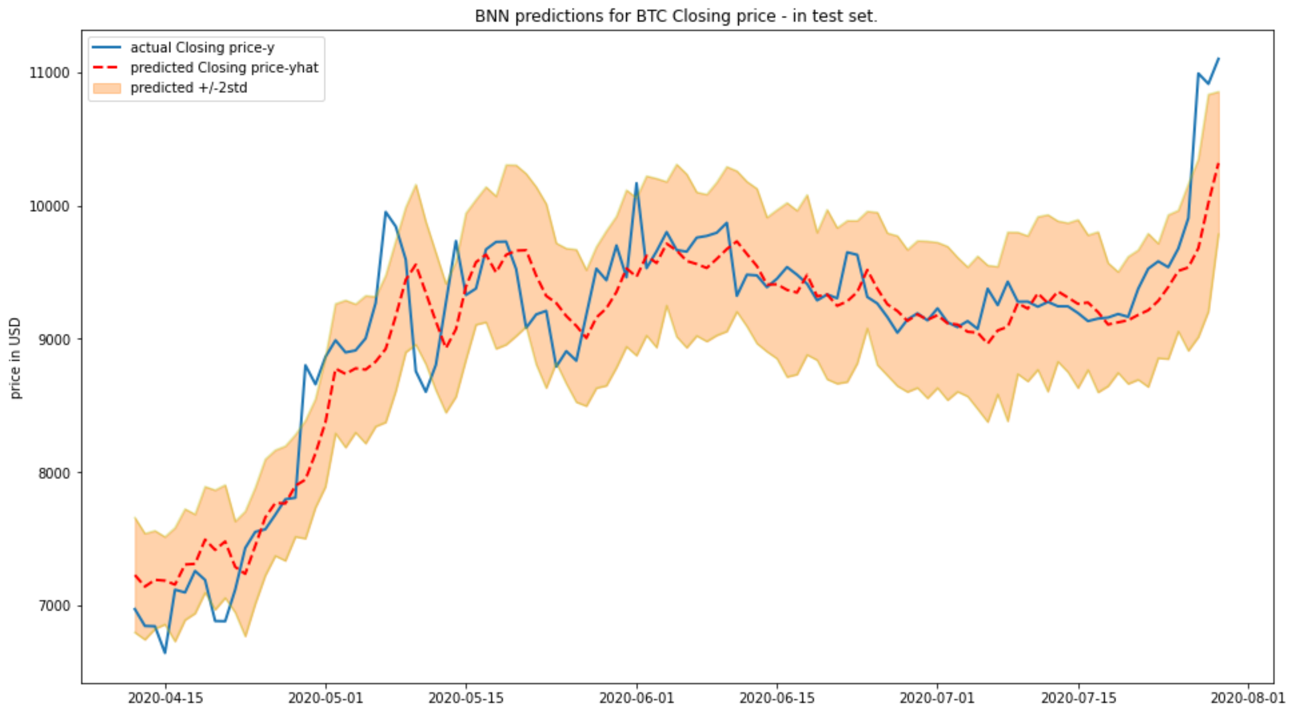


Figure 5.7: BTC Closing price prediction intervals with +/-2$\times\sigma$

6. DISCUSSION AND FUTURE WORK

6.1 Conclusion

In this thesis, we started by presenting a general introduction to the Efficient Market Hypothesis, while we introduced blockchain technology in short. We continued by showing all the components and methods derived from the domain of Machine Learning, in a comprehensive theoretical manner, which we then applied to a financial time series. Moving on to our first application, we showed how neural networks, especially LSTM and some of their variations, are useful tools in price forecasting. We compared our baseline hand-tuned LSTM model with the corresponding statistical one, and we showed its superiority over it. Beyond the hand-tuned LSTM, we applied an LSTM network, optimized by the Bayesian framework, and an LSTM-FCN model to our dataset, both resulting in highly intriguing outcomes. Afterwards, we introduced the aim of our second and last task, concerning the quantification of the uncertainty over Bitcoin's Closing price. The quantification of the uncertainty was handled via model averaging of multiple stochastic forward passes of an approximate Bayesian Neural Network, implemented following the Monte Carlo Dropout technique. This treatment allowed us to estimate the fluctuation boundaries for Bitcoin's next day Closing price. In contrast to the first application, this time a more sophisticated dataset was fed into our network, in which technical indicators as well a mini portfolio of correlated virtual coins were included. Lastly, as a final point, we illustrated how the results of the second algorithm could be interpreted in monitoring extreme price events in cryptocurrency markets over a 24-hour time period.

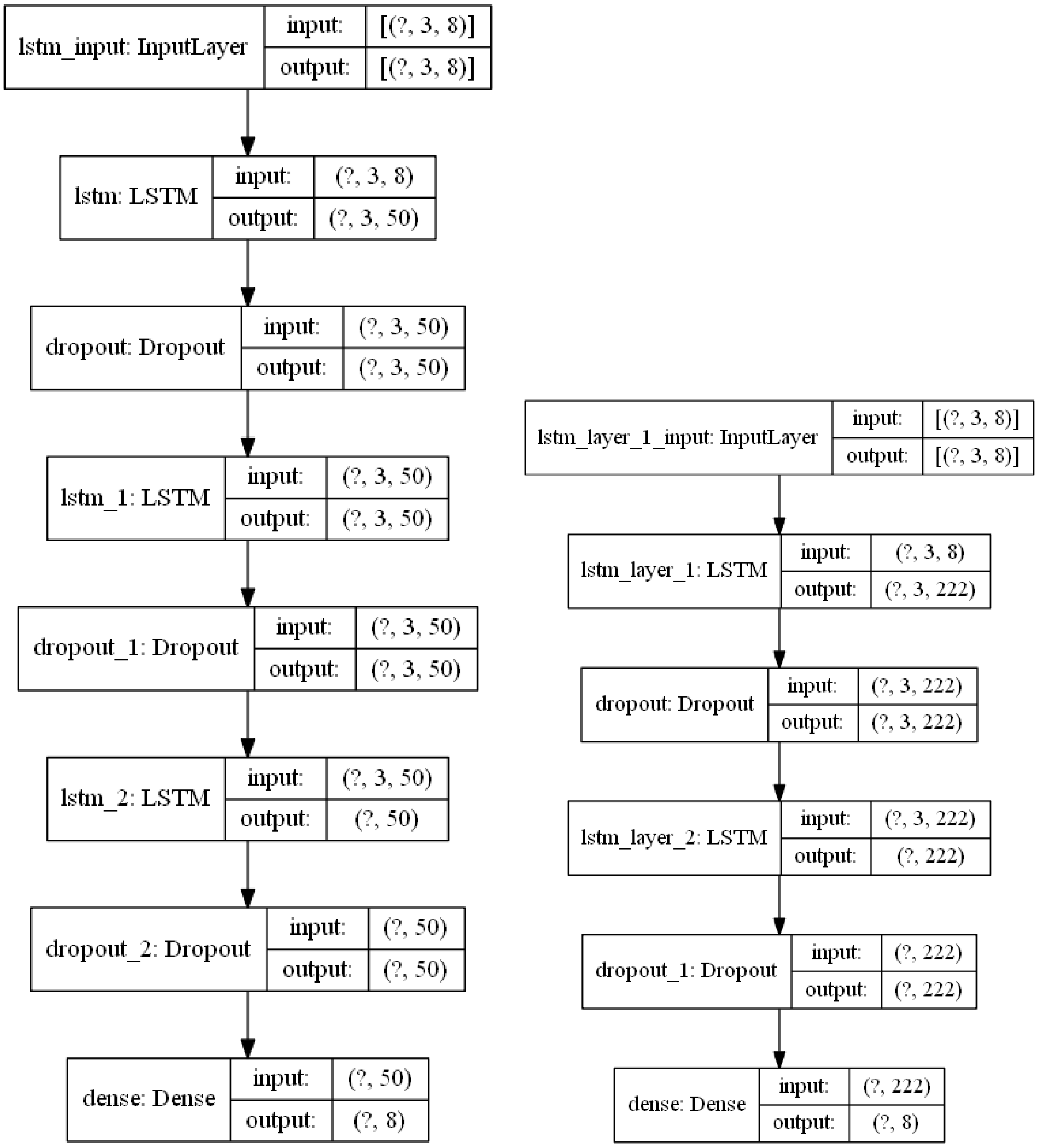
6.2 Future Work

This project can expand in the future in many different directions. In this section, I will propose 3 potential applications that can be implemented in the future. First of all, one extension could be the exploitation of the results of the proposed BNN model in the second application to perform trades based on the prediction intervals as noted in section 5.4.2. For instance, if the lower predicted threshold is still higher than the price at which we bought the asset, then the algorithm is triggered to sell and receive the residual gain. In the same manner, the trading amount could be determined based on the confidence of the algorithm. In simple linear fashion, when the algorithm is confident it could inform the investor to increase the bet, while if not, to decrease it. Of course, several different strategies could be followed as well. Moreover, the networks in both applications could be extended to perform multi-step predictions rather than only a single one. It would be interesting to attempt to predict the $t + k$ cryptocurrency movements, where k could be for example 5. Multi-step predictions, however, are very challenging, due to the error accumulation at each step. Effective ways have been introduced to partially alleviate this, such as with the Recursive Multi-step Forecast method, where the prediction at each step is used as input for the next one, meaning that the network is always attempting a one-step prediction but the final result is a multi-step forecasting. A final extension could be the forecasting of multiple different cryptocurrencies instead of only two. This can be implemented in various ways. One would be to create a unified predictor that predicts the prices of all the virtual coins simultaneously at each time step, whereas the 2nd approach would utilize the Transfer Learning technique in an already pre-trained model. The latter method should benefit the newly released coins that sustain small price historical datasets.

ABBREVIATIONS - ACRONYMS

ML	Machine Learning
DL	Deep Learning
MLP	Multilayer Perceptron
ANN	Artificial Neural Network
RNN	Recurrent Neural Network
LSTM	Long Short-Term Memory
GRU	Gated Recurrent Unit
OHLC	Open-High-Low-Close
FCN	Fully Convolutional Networks
MSE	Mean Squared Error
RMSE	Root Mean Squared Error
MAE	Mean Absolute Error
MAPE	Mean Absolute Percentage Error
MPE	Mean Percentage Error
MC	Monte Carlo

APPENDIX A. MODELS ARCHITECTURES



(a) LSTM

(b) Bayesian opt. LSTM

Figure A.1: LSTM and Bayesian optimized LSTM

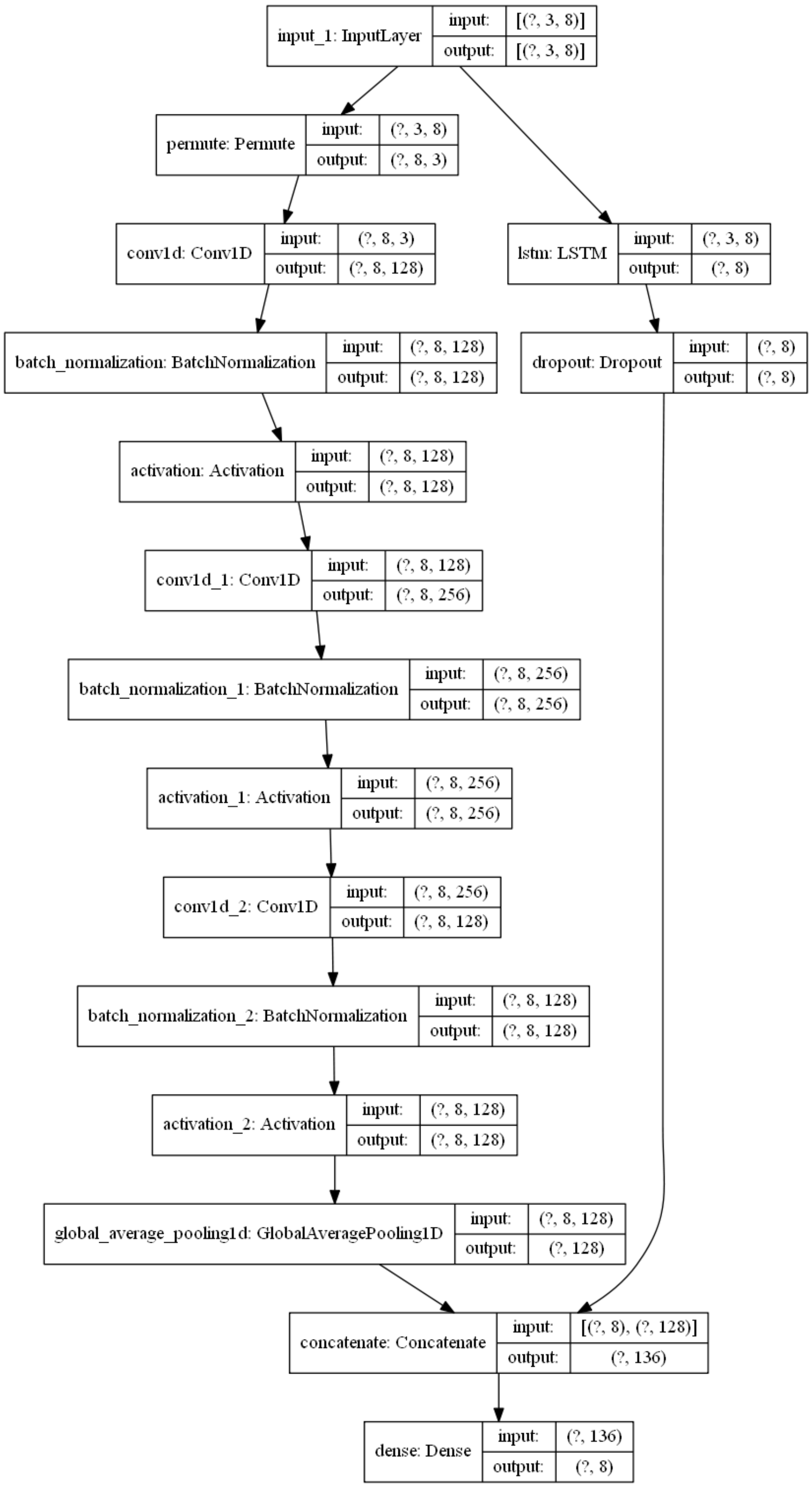

input_1: InputLayer | input: [(?, 3, 8)] | output: [(?, 3, 8)]
permute: Permute | input: (?, 3, 8) | output: (?, 8, 3)
conv1d: Conv1D | input: (?, 8, 3) | output: (?, 8, 128)
lstm: LSTM | input: (?, 3, 8) | output: (?, 8)
batch_normalization: BatchNormalization | input: (?, 8, 128) | output: (?, 8, 128)
dropout: Dropout | input: (?, 8) | output: (?, 8)
activation: Activation | input: (?, 8, 128) | output: (?, 8, 128)
conv1d_1: Conv1D | input: (?, 8, 128) | output: (?, 8, 256)
batch_normalization_1: BatchNormalization | input: (?, 8, 256) | output: (?, 8, 256)
activation_1: Activation | input: (?, 8, 256) | output: (?, 8, 256)
conv1d_2: Conv1D | input: (?, 8, 256) | output: (?, 8, 128)
batch_normalization_2: BatchNormalization | input: (?, 8, 128) | output: (?, 8, 128)
activation_2: Activation | input: (?, 8, 128) | output: (?, 8, 128)
global_average_pooling1d: GlobalAveragePooling1D | input: (?, 8, 128) | output: (?, 128)
concatenate: Concatenate | input: [(?, 8), (?, 128)] | output: (?, 136)
dense: Dense | input: (?, 136) | output: (?, 8)

Figure A.2: LSTM-FCN

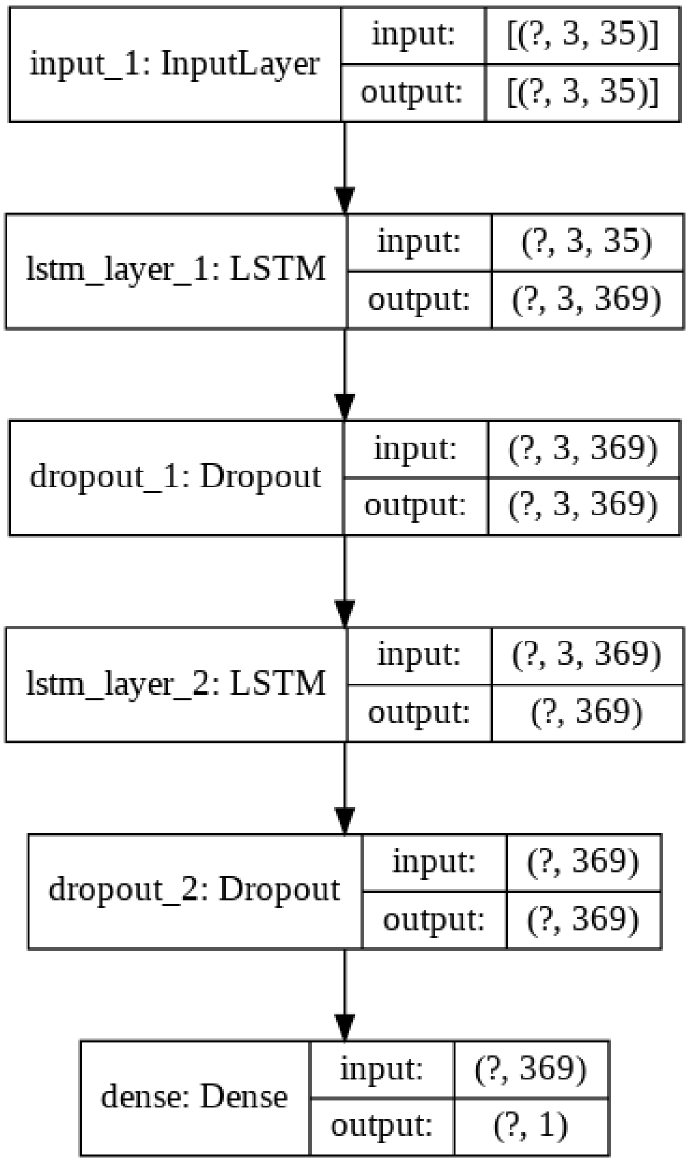


Figure A.3: MC-dropout BNN

APPENDIX B. APPLICATION A: HYPERPARAMETERS TRIED BY BAYESIAN OPTIMIZER

Table B.1: The 50 search results tried for the LSTM by Bayesian optimizer

MSE Loss	learning rate	layers	units	dropout rate	batch size
0.00047	0.00076	2	222	0.10000	100
0.00048	0.00003	2	512	0.10000	5
0.00052	0.00009	2	500	0.55562	86
0.00052	0.00131	2	43	0.29225	57
0.00060	0.00005	2	375	0.23611	34
0.00062	0.00000	2	512	0.10000	5
0.00067	0.00072	2	452	0.10213	58
0.00071	0.00013	2	292	0.11264	69
0.00077	0.00048	2	18	0.11201	12
0.00078	0.00001	2	512	0.90000	5
0.00078	0.00018	2	31	0.10871	11
0.00078	0.00001	2	483	0.44433	85
0.00083	0.00008	2	501	0.89991	56
0.00095	0.00023	2	508	0.87676	99
0.00096	0.00015	2	337	0.89176	98
0.00097	0.00064	2	476	0.48830	100
0.00111	0.00023	3	70	0.31345	9
0.00121	0.00017	2	65	0.71091	99
0.00237	0.00027	4	282	0.54530	10
0.00256	0.00057	2	192	0.53066	13
0.00336	0.00035	2	440	0.86009	10
0.00473	0.00093	3	312	0.89769	89
0.00496	0.00000	2	512	0.10000	34
0.00506	0.00017	5	397	0.12212	86
0.00508	0.00001	2	309	0.10000	24
0.00527	0.01000	2	5	0.10000	17
0.00529	0.00008	3	290	0.75380	18
0.00543	0.00220	3	38	0.17832	57
0.00560	0.00000	2	397	0.10000	5
0.00570	0.00000	2	40	0.10641	12
0.00598	0.00000	2	509	0.45698	42
0.00605	0.00004	2	98	0.11765	100
0.00624	0.00057	2	8	0.55249	16
0.00641	0.00005	2	35	0.77890	12
0.00643	0.00004	4	512	0.10000	100
0.00644	0.00001	4	215	0.26522	30
0.00664	0.00039	5	113	0.80741	61
0.00669	0.00001	2	21	0.10690	68
0.00836	0.00001	5	510	0.88634	93
0.00905	0.01000	2	512	0.10000	85
0.01205	0.00000	2	456	0.86499	95
0.01323	0.00664	3	240	0.56620	13
0.01339	0.00859	2	237	0.10165	21
0.01486	0.00963	2	476	0.88146	96
0.01585	0.00078	5	510	0.10768	17
0.02257	0.00000	5	179	0.10028	97
0.02420	0.01000	5	512	0.90000	100
0.03334	0.01000	2	5	0.90000	89
0.05631	0.00000	2	42	0.43731	22
0.08367	0.00000	5	19	0.89576	12

APPENDIX C. APPLICATION B: TOP 50 HYPERPARAMETERS TRIED BY BAYESIAN OPTIMIZER

Table C.1: Top 50 out of 100 search results tried for the BNN by Bayesian optimizer

Loss	learning rate	layers	units	batch size	λ	l
0.0004070	0.0004489	2	369	100	0.0000010	0.9000000
0.0005231	0.0000180	2	200	5	0.0000010	0.4000000
0.0005481	0.0000167	2	235	5	0.0000010	0.4191892
0.0005510	0.0000516	3	313	91	0.0000239	0.8686223
0.0005553	0.0000716	2	512	100	0.0000018	0.9000000
0.0005706	0.0003104	2	125	100	0.0100000	0.9000000
0.0006238	0.0000530	2	63	5	0.0000010	0.9000000
0.0006577	0.0000426	3	233	100	0.0100000	0.4000000
0.0006655	0.0001651	2	512	59	0.0000010	0.4000000
0.0006761	0.0003080	2	296	100	0.0100000	0.9000000
0.0006990	0.0002468	3	66	36	0.0004539	0.7909129
0.0007309	0.0000927	5	288	67	0.0000931	0.8338927
0.0007386	0.0000061	2	234	5	0.0100000	0.9000000
0.0008954	0.0085223	2	508	69	0.0050253	0.4012996
0.0008978	0.0030425	2	122	100	0.0100000	0.9000000
0.0009239	0.0000396	2	123	55	0.0100000	0.9000000
0.0009968	0.0005845	2	512	56	0.0100000	0.9000000
0.0010383	0.0000038	2	201	5	0.0000010	0.9000000
0.0010900	0.0000117	2	372	5	0.0100000	0.9000000
0.0010953	0.0018066	2	403	100	0.0009417	0.5013540
0.0011104	0.0000281	4	427	50	0.0000292	0.6221922
0.0011246	0.0100000	2	379	100	0.0001606	0.9000000
0.0011400	0.0000020	3	299	55	0.0000010	0.4000000
0.0011606	0.0097939	2	503	97	0.0000075	0.8879425
0.0011752	0.0000156	2	135	5	0.0000010	0.9000000
0.0012275	0.0100000	2	202	100	0.0000010	0.4000000
0.0012850	0.0001962	5	94	100	0.0100000	0.9000000
0.0013481	0.0003184	2	456	100	0.0100000	0.4000000
0.0013486	0.0012116	2	40	96	0.0059321	0.4028518
0.0014282	0.0001043	2	391	51	0.0100000	0.9000000
0.0014473	0.0000150	5	231	48	0.0100000	0.4000000
0.0014515	0.0003156	2	143	66	0.0100000	0.4000000
0.0014685	0.0000028	2	35	8	0.0000042	0.4060572
0.0014980	0.0001228	2	176	60	0.0000010	0.9000000
0.0015208	0.0000139	2	512	40	0.0000010	0.4559018
0.0015304	0.0038690	4	99	62	0.0010281	0.8420395
0.0015850	0.0000114	5	133	5	0.0000479	0.9000000
0.0015952	0.0005956	5	142	100	0.0100000	0.4000000
0.0016065	0.0007609	2	222	100	0.0001000	0.8000000
0.0016711	0.0000056	2	42	5	0.0000010	0.9000000
0.0016804	0.0000638	2	177	5	0.0100000	0.9000000
0.0017834	0.0000014	2	249	39	0.0005329	0.9000000
0.0018581	0.0000340	4	114	90	0.0067404	0.5976742
0.0018907	0.0000103	2	351	100	0.0100000	0.4000000
0.0019330	0.0000042	2	137	5	0.0029115	0.9000000
0.0019399	0.0000125	2	512	51	0.0100000	0.9000000
0.0020256	0.0100000	2	5	100	0.0000010	0.9000000
0.0021096	0.0100000	2	53	100	0.0023237	0.9000000
0.0021156	0.0000010	2	307	5	0.0100000	0.4000000
0.0021652	0.0000104	2	266	50	0.0000010	0.9000000

BIBLIOGRAPHY

[1] Alex Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[3] Eric Brochu, Vlad M Cora, and Nando De Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*, 2010.

[4] Laura Alessandretti, Abeer ElBahrawy, Luca Maria Aiello, and Andrea Baronchelli. Anticipating cryptocurrency prices using machine learning. *Complexity*, 2018, 2018.

[5] Gerald P Dwyer. The economics of bitcoin and similar private digital currencies. *Journal of Financial Stability*, 17:81–91, 2015.

[6] Rainer Böhme, Nicolas Christin, Benjamin Edelman, and Tyler Moore. Bitcoin: Economics, technology, and governance. *Journal of economic Perspectives*, 29(2):213–38, 2015.

[7] Sima Siami-Namini and Akbar Siami Namin. Forecasting economics and financial time series: Arima vs. lstm. *arXiv preprint arXiv:1803.06386*, 2018.

[8] Isaac Madan, Shaurya Saluja, and Aojia Zhao. Automated bitcoin trading via machine learning algorithms. *URL: http://cs229. stanford. edu/proj2014/Isaac% 20Madan*, 20, 2015.

[9] S. McNally, J. Roche, and S. Caton. Predicting the price of bitcoin using machine learning. In *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 339–343, 2018.

[10] Huisu Jang and Jaewook Lee. An empirical study on modeling and prediction of bitcoin prices with bayesian neural networks based on blockchain information. *Ieee Access*, 6:5427–5437, 2017.

[11] Lukáš Pichl and Taisei Kaizoji. Volatility analysis of bitcoin price time series. *Quantitative Finance and Economics*, 1:474–485, 12 2017.

[12] Emmanouil Christoforou, Ioannis Z Emiris, and Apostolos Florakis. Neural networks for cryptocurrency evaluation and price fluctuation forecasting. In *Mathematical Research for Blockchain Economy*, pages 133–149. Springer, 2020.

[13] Eugene F Fama. Efficient capital markets: A review of theory and empirical work. *The journal of Finance*, 25(2):383–417, 1970.

[14] Werner FM De Bondt and Richard Thaler. Does the stock market overreact? *The Journal of finance*, 40(3):793–805, 1985.

[15] Narasimhan Jegadeesh and Sheridan Titman. Returns to buying winners and selling losers: Implications for stock market efficiency. *The Journal of finance*, 48(1):65–91, 1993.

[16] Miljan Lekovic. Evidence for and against the validity of efficient market hypothesis. *Economic Themes*, 56:369–387, 11 2018.

[17] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, Manubot, 2019.

[18] Zibin Zheng, Shaoan Xie, Hongning Dai, Xiangping Chen, and Huaimin Wang. An overview of blockchain technology: Architecture, consensus, and future trends. In *2017 IEEE international congress on big data (BigData congress)*, pages 557–564. IEEE, 2017.

[19] A. R. Barron. Universal approximation bounds for superpositions of a sigmoidal function. *IEEE Transactions on Information Theory*, 39(3):930–945, 1993.

[20] Moshe Leshno, Vladimir Ya Lin, Allan Pinkus, and Shimon Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural networks*, 6(6):861–867, 1993.

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

[22] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[23] Andrew D Back and Ah Chung Tsoi. Fir and iir synapses, a new neural network architecture for time series modeling. *Neural computation*, 3(3):375–385, 1991.

[24] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

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

[26] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. 1999.

[27] David JC MacKay. The evidence framework applied to classification networks. *Neural computation*, 4(5):720–736, 1992.

[28] Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. Weight uncertainty in neural networks. *arXiv preprint arXiv:1505.05424*, 2015.

[29] Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*, pages 1050–1059, 2016.

[30] Yarin Gal and Zoubin Ghahramani. A theoretically grounded application of dropout in recurrent neural networks. In *Advances in neural information processing systems*, pages 1019–1027, 2016.

[31] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[32] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *The Journal of Machine Learning Research*, 13(1):281–305, 2012.

[33] Peter I Frazier. A tutorial on bayesian optimization. *arXiv preprint arXiv:1807.02811*, 2018.

[34] Nitish Srivastava, Elman Mansimov, and Ruslan Salakhudinov. Unsupervised learning of video representations using lstms. In *International conference on machine learning*, pages 843–852, 2015.

[35] F. Karim, S. Majumdar, H. Darabi, and S. Chen. Lstm fully convolutional networks for time series classification. *IEEE Access*, 6:1662–1669, 2018.