BENEMÉRITA UNIVERSIDAD AUTÓNOMA DE PUEBLA

FACULTAD DE CIENCIAS FÍSICO MATEMÁTICAS
LICENCIATURA EN ACTUARÍA

Deep learning applied to cryptocurrencies prices one-step forecast.

Degree in Actuarial Science.

Andrés Pérez Pérez

February 2019

SUPERVISOR:

Mtr. Radosav Sekulic
Mtr. José Asunción Hernández

Summary

Over the last few years, neural networks have become extremely popular, and their usage is increasing rapidly, it has become a powerful tool that is being applied to finance, business decisions and unprecedented interest in the search for new applications has been generated. This thesis has investigated the use of neural networks for one-step time series forecasting on cryptocurrencies prices. *Multi Layer Perceptron* (MLP), *Long Short Term Memory* (LSTM) and *Convolutional Neural Networks* (CNN) models are put to test to see if binary classification accuracy above 50% can be given using time series data, in this case, prices of four of the most capitalized cryptocurrencies, which are extremely volatile *(Litecoin [57], Bitcoin [55], Ethereum [56] and Bitcoin Cash [58]).* In this project, fundamental properties of cryptocurrencies are not considered, only their technical aspects, namely price movement and volume are observed. The assignment focuses on designing a small-embedded neural network with greater accuracy than 50%.

The popularity of cryptocurrencies skyrocketed in 2017 due to several consecutive months of superexponential growth of their market capitalization, which peaked at more than $800 billions in Jan. 2018 [62]. Today, there are more than actively traded cryptocurrencies. According to a recent survey, between 2.9 and 5.8 million of private as well as institutional investors are in the different transaction networks and access to the market has become easier over time [63]. Major cryptocurrencies can be bought using $fiat$ currency in many online exchanges *(e.g., Binance [64], Kraken [65], etc.)* and then be used in their turn to buy less popular cryptocurrencies. More recently, deep neural networks have attracted the attention of researchers in the financial field to make predictions on financial markets.

The different neural network architectures are built using a deep learning library in $Python$ [66], called $Keras$. This is a high-level software framework, built on top of either *Tensorflow* [1] or *Theano*[2], for fast and easy prototyping of neural networks. The conclusion of the thesis is that, the $LSTM$ and MLP satisfied the requirements of the assignment during the work of this thesis. The $LSTM$ showed the most promising results, above 58 % accuracy, being able to extract information about the training set that increased the classification accuracy of the test. This leads the way for further development and an eventual hardware implementation of the inference phase reducing the run-time latency.

[1] $TensorFlow$ is a free software library that is used to perform numerical calculations using data flow diagrams.
[2] $Theano$ is a Python library that allows you to define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays efficiently.

"We are made of star stuff. We are a way for the cosmos to know itself."

– Carl Sagan

Acknowledgements

I would like to thank the universe that gave us life, if there is life we have everything.

Note.

I have been working on this project with my supervisor Radosav Sekulic and with Jose Asuncion Hernandez's help during last year to accomplish this thesis.

The code is written in Python [66]. The neural networks are implemented using Keras [1], which is a free deep learning library. This uses Tensorflow [2] as backend. The code is mostly based on numpy and panda's libraries, using Keras for the neural network implementations. In addition to the Keras documentation, it is possible to find a long list of examples, available on their GitHub repository [3].

Deep learning applied to cryptocurrencies prices one-step forecast.

Content.

Summary 2

Acknowledgements 3

Note. 3

Abbreviations. 7

Part I. Background 8

1. Introduction 10

1.1 Motivation 10

1.2 Objectives 11

1.3 Outcomes 12

1.4 Thesis structure. 12

Part II. Literature Review 13

2. Deep learning 14

2.1 Origin and Statistical Learning Framework. 14

2.1.1 Origin 14

2.1.2 A Formal Model – The Statistical Learning Framework. 15

2.2 Learning Algorithms 19

2.2.1 The Task, T 19

2.2.2 The Performance Measured, P 20

2.2.3 The Experience, E 20

2.3 Classification. 20

2.4 Regression. 21

2.5 Unsupervised learning algorithms 21

2.6 Supervised learning algorithms 21

2.7 Linear Regression. 22

2.8 Gradient-Based Optimization 24

2.9 Stochastic Gradient Descent 26

2.10 Biological Neuron 27

2.11 Artificial Neuron....27
2.11.1 Capacity, overfitting and underfitting....28
2.11.2 Activation Functions....30
2.12 Artificial Neural Networks (ANN)....34
2.12.1 Forward Propagation....34
2.12.2 Weights & Biases....35
2.12.3 Optimization....36
2.12.4 Backpropagation....37
2.12.5 Batch size, Iterations, and Epochs....37
2.12.6 Training_Phase and Inference....37
2.12.7 Dropout....38
2.12.8 Data Augmentation....38
2.12.9 Batch Normalization....38
2.13 Multilayer Perceptron (MLP)....39
2.14 Convolutional Neural Network (CNN)....40
2.14.1 Input layer....41
2.14.2 Convolutional layer....41
2.15 Recurrent Neural Network (RNN)....45
2.15.1 Unfolding Computational Graphs....46
2.15.2 The long short-term memory....50
Part III Methodology....54
3. General Structure....55
3.1 Method....55
3.2 Specification....56
3.3 Baseline criteria....57
4. Pre-processing data and tools....57
4.1 Keras - Software Framework....57
4.1.1. Model Creation....58
4.1.2 Layers....58
4.1.3 Compilation....58
4.1.4 Training....59
4.1.5 Evaluation....59
4.2 Pre- Processing....59
4.2.1 Shape of Input Frame....59
4.4.2 Scaling....60
4.4.3 Labeling....60

4.4.4 Feature Selection 61
4.4.5 Selecting the Number of Samples 61
5. Design 62
5.1 Neural Network Topologies 62
5.1.1 Multi-Layer Perceptron (MLP) 63
5.1.2 Recurrent Neural Network (RNN) 63
5.1.3 Long Short-Term Memory (LSTM) 64
5.1.4 Convolutional Neural Network (CNN) 64
5.2 Choosing Optimizer 64
5.3 Choosing Look Back window. 64
6 Implementation 64
6.1 Multi-Layer Perceptron (MLP) 65
6.2 Convolutional Neural Network (CNN) 66
6.3 Long Short-Term Memory (LSTM) 67
Part IV Results and Discussion 68
7. Results and Discussion 69
7.1 Multi-Layer Perceptron (MLP) 69
7.2 Convolutional Neural Network (CNN) 72
7.3 Long Short-Term Memory (LSTM) 74
8. Results and Discussion 76
9. Conclusion and Future work 9.1 Overview 77
9.2 Future Work 78
Code: Pre-processing data 79
Code: Multilayer Perceptron 81
Code: Convolutional Neural Network 83
Code: Long-Short Term Memory 85
Bibliography 87

Abbreviations.

SYMBOL = DEFINITION

$AI = Artificial\ Intelligence$

$ANN = Artificial\ Neural\ Network$

$CNN = Convolutional\ Neural\ Network$

$CPU = Central\ Processing\ Unit$

$DNN = Deep\ Neural\ Network$

$GPU = Graphics\ Processing\ Unit$

$LSTM = Long\ Short-Term\ Memory$

$ML = Machine\ Learning$

$MLP = Multilayer\ Perceptron$

$NLP = Natural\ Language\ Processing$

$NN = Neural\ Network$

$ReLU = Rectified\ Lunear\ Unit$

$RNN = Recurrent\ Neural\ Network$

Part I.

Background.

1. Introduction.

This chapter gives a short motivation, describes the objectives, and presents the outcome from this thesis.

1.1 Motivation

Lately, deep neural networks have attracted the attention of researchers in the financial field to make predictions on financial markets, machine learning algorithms evolved from analyzing samples of data instead of accurately modelling all parts of a system from known models and equations. Indeep learning, or so called end-to-end learning, all parameters of the network are trained from input data, eliminating the need for prior knowledge about the system's dynamics to build a model. Deep learning has proven to be very effective. $Neural\ networks\ (NN)$ have become impressively accurate, even being as good as humans in tasks like image classification. However, it is still being seen that humans perform better with degraded or distorted images, as discussed in [4] (Dodge, S. 2017) and [5] (Geirhos, R. 2017). Moreover, $Google's\ AlphaZero\ AI$ won on all matches against the world champion chess program, Stockfish, in a 100-game match up, according to the Guardian [6].

Much of the interest about deep neural networks $(DNNs)$ has been the increasing accuracy, but this project mainly focuses on an embedded neural network, a smaller NN with focus on size and accuracy. It is meant to forecast one-step cryptocurrencies prices that contain highly random data, containing very little deterministic structures or patterns. Different NN architectures are put to test to see if any patterns can be extracted about this type of data, and thus increase the accuracy of a binary classification to over 50%.

1.2 Objectives

The goal of this assignment is to evaluate the ability of different neural network topologies to forecast cryptocurrencies prices. A one-step forecast is implemented, predicting the next single value of one-time series based on the historical data. In this project, the prices of four of the most capitalized cryptocurrencies [60], [61] (Litecoin [57], Bitcoin [55], Ethereum [56] and Bitcoin Cash [58]) are used. This is repeated, doing multiple one-step predictions without retraining the model. The data is highly random and will be predicted using no domain knowledge, meaning that the forecast is solely based on the time series itself.

Different architectures will be tested using a deep learning software framework. Since the data is highly random, containing little or maybe no structures and patterns, the goal is to test the different architectures to see if any of them can give binary classification accuracy slightly above 50%.

The objectives for this project are:

1. To write a literature review and acquire necessary background knowledge: the background theory consists of, among other things, the basic understanding of deep learning, recurrent neural networks, and convolutional neural networks.

2. To design a theoretical suggestion of different neural network architectures based on the literature review.

3. Acquiring and pre-processing of input data. Test and compare the results and evaluate the work.

1.3 Outcomes

The outcomes from this thesis are:

- A literature review of neural network basics and neural networks used on time series.
- System for pre-processing and combining feature data used in this thesis.
- Training ANN models for binary classification of the dataset in question.
- Discussion about the use case and further development.

1.4 Thesis structure.

The document is divided into five different parts: Background, Literature Review, Method, Results and Conclusion.

The Background gives a short motivation and introduces the problem and its objectives. The Literature Review consists of three chapters: Background Theory of Deep Learning, Software Tools. The theory chapter starts from the basics of deep learning. This reflects the prior knowledge of the writer, as an actuarial student, about the subject. It goes through the necessary theory about neural networks, including topics like basic properties of neurons, layers, multi-layer perceptron, activation functions, forward and backward propagation, etc.

Different network topologies can be used for cryptocurrencies prices prediction, and for that reason the chapter includes the two commonly used networks: convolutional neural networks and recurrent neural networks. GPUs[3] are the most commonly used platform for NNs today. This chapter also covers the most famous network architectures and the techniques they introduced. The Method part includes the following chapters: Data Preparation, Design, and Implementation.

[3] A graphics processing unit (GPU) is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display device.

Part II.

Literature Review

2. Deep learning.

This chapter introduces the basic terminology used in deep learning. It goes through the building blocks of the neural networks, and introduces the commonly used network topologies: *Multilayer Perceptron*, *recurrent neural networks*, and *convolutional neural networks* [76].

2.1 Origin and Statistical Learning Framework.

2.1.1 Origin

Deep learning (DL) is a sub field of machine learning that is a sub field of the much broader fieldof $Artificial\ Intelligence$ (AI). AI is deeply covered by other texts such as Artificial intelligence: a modern approach by Stuart Russell and Peter Norvig [8] and will not be covered in this thesis apart from pointing out that deep learning originates from AI.

The idea of deep learning is inspired by the biological behavior of the brain. Each neuron, the main component or building block of the brain, transmits information to other neurons forming a very large and complex network. Each node, or neuron, is stimulated by inputs and passes information, or some of the information, on to other neurons.

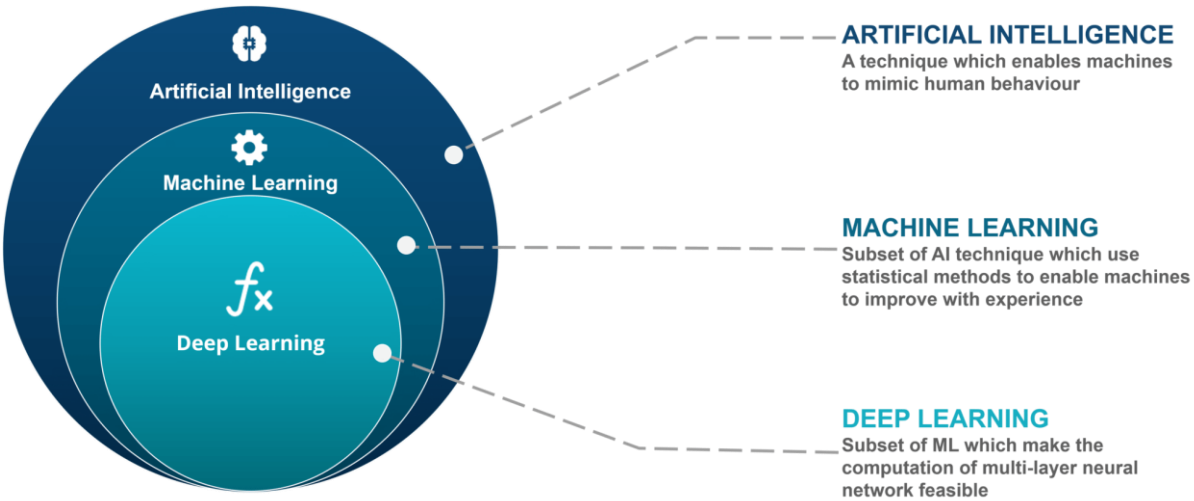


Figure 2.1 Comparison between artificial intelligence, machine learning and deep learning. [67].

2.1.2 A Formal Model – The Statistical Learning Framework.

This section (2.1.2) is based on Vladimir Vapkin work [80]. The setting of learning. Only basic explanation of the fundamentals concepts and elements to understand deep learning are covered in this section since Vapkin works has explained it with more detail and strictness.

- The system's input: In the basic statistical learning setting, the system has access to the following:
    - Domain set: An arbitrary set, X . This is the set of objects that should be labeled. For example. Usually, these domain points will be represented by a vector of features.
    - Label set: For this thesis, the label set is restricted to be a two-element set, usually $\{0, 1\}$ or $\{-1, +1\}$. Let Y denote our set of possible labels.
    - Training data: $S = ((x_1, y_1) \ldots (x_m, y_m))$ defined as a sequence of pairs in $X \times Y$: it is, a sequence of labeled domain points. This is the input that the system has access to. Be S a training set[4].
- The system's output: The system is requested to come up with prediction rule, $h : X \to Y$. This rule could be called also like a predictor, a hypothesis, or a classifier. The predictor is used to predict the label of new domain samples. The notation $A(S)$ is used to denote the hypothesis that a learning algorithm, A, returns upon receiving the training sequence S.
- A simple data-generation model: First, it is assumed that the instances are generated by some probability distribution. D is the probability distribution over X. It is important to assume that the system knows nothing about this distribution. This could be any arbitrary probability distribution. As to the labels, in the current discussion is assumed that there is some "correct" labeling function, $f : X \to Y$, and that $y_i = f(x_i)$ for all i. The labeling function is unknown to the system. In fact, this is just what the system is trying to figure out. In summary, each pair in the training data S is generated by first sampling a point x_i according to D and then labeling it by f.
- Measures of success: The error of a classifier is the probability of predicting the wrong label on a random data point generated by the underlying distribution. That is, the error of h is the probability to draw a random instance x, according to the distribution D, such that $h(x)$ does not equal $f(x)$.

[4] Despite the "set" notation, S is a sequence. In particular, the same example may appear twice in S and some algorithms can consider the order of examples in S.

Formally, given a domain subset[5], $A \subset X$, the probability distribution, D, assigns a number, $D(A)$, which determines how likely it is to observe a point $x \in A$. In many cases, A is an event and express it using a function $\pi : X \to \{0,1\}$, namely, $A = \{x \in X : \pi(x) = 1\}$. In that case, the notation is $Px \sim D[\pi(x)]$ to express $D(A)$.

The error of a prediction rule is , $h : X \to Y$, to be $L_{D,f}(h) = P_{x\sim D}[h(x) \neq f(x)] = D(\{x : h(x) \neq f(x)\})$. That is, the error of such h is the probability of randomly choosing an example x for which $h(x) \neq f(x)$. The subscript (D, f) indicates that the error is measured with respect to the probability distribution D and the correct labeling function f.

- A note about the information available to the system: The system is blind to the underlying distribution D over the world and to the labeling function f. The only way the system can interact with the environment is through observing the training set.

2.1.2.1 Empirical Risk Minimization.

As mentioned earlier, a learning algorithm receives as input a training set S, sampled from an unknown distribution D and labeled by some target function f, and should output a predictor $h_S : X \to Y$ (the subscript S emphasizes the fact that the output predictor depends on S). The goal of the algorithm is to find h_S that minimizes the error with respect to the unknown D and f. Since the system does not know what D and f are, the true error is not directly available to the system. A useful notion of error that can be calculated by the system is the training error – the error the classifier incurs over the training sample:

$$L_S(h) = \frac{|\{i \in [m] : h(x_i) \neq y_i\}|}{m}$$

where $[m] = \{1, \ldots, m\}$.

The terms $empirical\ error$ and $empirical\ risk$ are often used interchangeably for this error. Since the training sample is the snapshot of the world that is available to the system, it makes sense to search for a solution that works well on that data. This learning paradigm – coming up with a predictor h that minimizes $L_S(h)$ – is called $Empirical\ Risk\ Minimization$ or ERM for short.

Definition: ($The\ Realizability\ Assumption$) There exists $h^* \in H\ s.t.\ L_{(D,f)}(h^*) = 0$. Note that this assumption implies that with probability 1 over random samples, S, where the instances of S are sampled according to D and are labeled by f, then $L_S(h^*) = 0$.
The realizability assumption implies that for every ERM hypothesis $L_S(h_S) = 0$.

[5] Strictly speaking, A is a member of some σ-algebra of subsets of X, over which D is defined.

However, Is the more interest the true risk of h_S, $L_{(D,f)}(h_S)$, rather than its empirical risk. Clearly, any guarantee on the error with respect to the underlying distribution, D, for an algorithm that has access only to a sample S should depend on the relationship between D and S. The common assumption in statistical machine learning is that the training sample S is generated by sampling points from the distribution D independently of each other. Formally

- The $i.i.d.$ assumption: The examples in the training set are independently and identically distributed ($i.i.d.$) according to the distribution D. That is, every x_i in S is freshly sampled according to D and then labeled according to the labeling function, f. This assumption is denoted by $S \sim D^m$ where m is the size of S, and D^m denotes the probability over m-tuples induced by applying D to pick each element of the tuple independently of the other members of the tuple. Intuitively, the training set S is a window through which the system gets partial information about the distribution D over the world and the labeling function, f. The larger the sample gets, the more likely it is to reflect more accurately the distribution and labeling used to generate it.

Since , $L_{(D,f)}(h_S)$ depends on the training set, S, and that training set is picked by a random process, there is randomness in the choice of the predictor h_S and, consequently, in the risk $L_{(D,f)}(h_S)$. Formally, it is a random variable. It is not realistic to expect that with full certainty S will suffice to direct the system toward a good classifier (from the point of view of D), as there is always some probability that the sampled training data happens to be very no representative of the underlying D. The is important to address the probability to sample a training set for which $L_{(D,f)}(h_S)$ is not too large. Usually, the probability of getting a no representative sample denoted by δ, and call $(1-\delta)$ the confidence parameter of our prediction. On top of that, since is not possible to guarantee perfect label prediction, introduce another parameter should be introduced for the quality of prediction, the $accuracy\ parameter$ commonly denoted by ε. The event $L_{(D,f)}(h_S) > \varepsilon$ is a failure of the system, while if $L_{(D,f)}(h_S) \leq \varepsilon$ The output of the algorithm is as an approximately correct predictor. Therefore (fixing some labeling function $f: X \to Y$), is the main interest to upper bounding the probability to sample m-tuple of instances that will lead to failure of the system.

2.1.2.2 Probably Approximately Correct (PAC) learning.

Definition of PAC Learnability: A hypothesis class H is PAC learnable if there exist a function $m_H: (0,1)^2 \to N$ and a learning algorithm with the following property: For every $\varepsilon, \delta \in (0,1)$, for every distribution D over X, and for every labeling function $f: X \to \{0,1\}$, if the realizable assumption holds with respect to H, D, f, then when running the learning algorithm on $m \geq m_H(\varepsilon, \delta)$ $i.i.d.$ examples generated by D and

labeled by f, the algorithm returns a hypothesis h such that, with probability of at least $1 - \delta$ (over the choice of the examples), $L_{(D,f)}(h) \leq \varepsilon$.

The definition of Probably Approximately Correct learnability contains two approximation parameters. The accuracy parameter ε determines how far the output classifier can be from the optimal one (this corresponds to the "approximately correct"), and a confidence parameter δ indicating how likely the classifier is to meet that accuracy requirement (corresponds to the "probably" part of "PAC"). Under the data access model that is investigated, these approximations are inevitable. Since the training set is randomly generated, there may always be a small chance that it will happen to be noninformative (for example, there is always some chance that the training set will contain only one domain point, sampled repeatedly). Furthermore, even when the training sample does faithfully represent D, because it is just a finite sample, there may always be some fine details of D that it fails to reflect. The accuracy parameter, ε, allows "forgiving" the system's classifier for making minor errors.

$Sample\ Complexity$. The function $m_H : (0,1)^2 \to N$ determines the sample complexity of learning H: that is, how many examples are required to guarantee a probably approximately correct solution. The sample complexity is a function of the accuracy ε and confidence (δ) parameters. It also depends on properties of the hypothesis class H.

Note that if H is PAC learnable, there are many functions m_H that satisfy the requirements given in the definition of PAC learnability. Therefore, to be precise, the sample complexity of learning H should be the "*minimal function*," in the sense that for any $\varepsilon, \delta, m_H(\varepsilon, \delta)$ is the minimal integer that satisfies the requirements of PAC learning with accuracy ε and confidence δ.

It can be rephrased as stating:

$COROLLARY$ 3.2. Every finite hypothesis class is PAC learnable with sample complexity.

$$m_H(\varepsilon, \delta) = \frac{\log(|\mathrm{H}|/\delta)}{\varepsilon}$$

2.1.2.3 Generalized Loss Functions

$Generalized\ Loss\ Functions$: Given any set H (that plays the role of our hypotheses, or models) and some domain Z let l be any function from $H \times Z$ to the set of nonnegative real numbers, $l : H \times Z \to R_+$.

We call such functions loss functions.

Note that for prediction problems $Z = X \times Y$. However, the loss function is generalized beyond prediction tasks, and therefore it allows Z to be any domain of example.

The risk function to be the expected loss of a classifier, $h \in H$, with respect to a probability distribution D over Z, namely,

$$L_D(h) = E_{z\sim D}\ l(h,z).$$

That is, the expectation of the loss of h over objects z picked randomly according to D. Similarly, the empirical risk should be the expected loss over a given sample $S = (Z_i,\ldots,Z_m) \in Z^m$, namely,

$$L_S(h) = \frac{1}{m}\sum_{i=1}^{m} l(h,z).$$

$DEFINITION\ 3.4.(Agnostic\ PAC\ Learnability\ for\ General\ Loss\ Functions)$ A hypothesis class H is agnostic PAC learnable with respect to a set Z and a loss function $l : H \times Z \to R_+$, if there exist a function $m_H : (0,1)^2 \to N$ and a learning algorithm with the following property: For every $\varepsilon,\delta \in (0,1)$ and for every distribution D over Z, when running the learning algorithm on $m \geq m_H\ (\varepsilon,\delta)\ i.i.d.$ examples generated by D, the algorithm returns $h \in H$ such that, with probability of at least $1 - \delta$ (over the choice of the m training examples).

$$L_D(h) \leq min\ L_D(h') + \varepsilon$$

Where $L_D(h) = E_{z\sim D}\ l(h,z)$.

2.2 Learning Algorithms.

A machine-learning algorithm is an algorithm that can learn from data, but what is meant by learning? Mitchell in 2013 [13], provides a succinct definition:

"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E". One can imagine a wide variety of experiences E, tasks T, and performance measures P.

2.2.1 The Task, T

Machine learning facilitates the tacklement of tasks that are too difficult to solve with fixed programs written and designed by human beings. From a scientist and philosophical point of view, machine learning is interesting because developing the understanding of it entails developing the understanding of the principles that underlie intelligence.

In this relativity formal definition of the word "task", the process of learning itself is not the task. The means to attain the ability to perform a task is by learning. For example, if it is desired that a robot could be able to walk, then walking is the task. The robot could be

programed to learn how to walk, or a written program that specifies how to walk manually could be directly attempted.

Machine learning tasks are usually described in terms of how the machine learning system should process an example. An example is a collection of features that have been quantitatively measured from some object or event and that must be processed by the machine learning system. Typically, an example is represented as a vector $x \in \mathrm{R}^{n}$ where each entry x_i of the vector is another feature. For example, the features of an image are usually the values of the pixels in the image.

Many kinds of tasks can be solved with machine learning, for example: classification, regression, transcription, machine translation, structured output, anomaly detection, synthesis and sampling, imputation of missing values, denoising, density estimation… but for the purposes of the current thesis, only classification and regression will be explained.

2.2.2 The Performance Measured, P.

To evaluate the abilities of a machine-learning algorithm, a quantitative measure of its performance must be designed. Usually this performance measure P is specific to the task T being carried out of the system.

For tasks such as classification, classification with missing inputs, and transcriptions, the accuracy of the model is measured. Accuracy is just the proportion of examples for which the model produces the correct output. Equivalent information can also be obtained by measuring the error rate, the proportion of examples for which the model produces an incorrect output. Error is often referred to as the expected 0-1 loss. The 0-1 $loss$ on an example is 0 if it is correctly classified, and 1 if it is not.

2.2.3 The Experience, E

Machine learning algorithms can be broadly categorized as unsupervised or supervised by the kind of experience they can have during the learning process.

2.3 Classification.

In this type of task, the computer is asked to specify which of k categories some input belongs. To solve this task, the learning algorithm is usually asked to produce a function

$$f: R^n \rightarrow \{1, \ldots, k\}.$$

When $y = f(x)$, the model assigns an input described by vector $\boldsymbol{x}$ to a category identified by numeric code y.

2.4 Regression.

In this type of task, the computer program is asked to predict a numerical value given some input. To solve this task, the learning algorithm is asked to output a function $f : R^n \rightarrow R$. This type of task is like classification, except that the format of output is different. An example of a regression task is the prediction of the expected claim amount that an insured person will make (used to set insurance premiums), or the prediction of future prices of securities. These kinds of predictions are also used for algorithmic trading.

2.5 Unsupervised learning algorithms

It is first needed to experience a dataset containing many features to then learn useful properties of the structure of this dataset. In the context of deep learning, the aim is to learn the entire probability distribution that generate a dataset, whether explicitly, as in density estimation, or implicitly, for the tasks like synthesis or denoising. Some other unsupervised learning algorithms perform other roles like clustering, which consists of dividing the dataset into clusters of similar examples.

2.6 Supervised learning algorithms

It is necessary to experience a dataset containing features, but each example also associated with a label or target. For example, the Iris dataset is annotated with the species of each iris plant. A supervised learning algorithm can study the Iris dataset and learns to classify iris plants into three different species based on their measurements.

Overall, unsupervised learning involves observing several examples of random vector x and attempting to implicitly or explicitly learn the probability distribution $p(x)$, or some interesting properties of that distribution. While supervised learning involves observing several examples of a random vector x and an associated value or vector y, then learning to predict y from x usually by estimating $p(y|x)$.

The term-supervised learning originates from the view of the target $\boldsymbol{y}$ being provided by an instructor or teacher who shows the machine learning system what to do. In unsupervised learning there is no instructor or teacher, and the algorithm must learn to make sense of the data without this guide.

2.7 Linear Regression.

This review will not go in depth on linear regression topics, since they are broadly covered in other texts like Rentier, A. [72] and Sober, G. [73]. However, it will be explained to understand the idea of machine learning.

To make this more concrete, an example of a simple machine learning algorithm is presented below. As the name implies, linear regression solves a regression problem. In other words, the goal is to build a system that can take a vector $x \in R^n$ as input and predict the value of a scalar $y \in R$ as its input. The output of linear regression is a linear function of the input. Let $\hat{y} \in R$ be the value that the model predicts y should take on. The output is defined to be

$$\hat{y} = w^T x \qquad 2.7.1$$

where $w \in R^n$ is a vector of parameters, $w = (w_1, \dots, w_n)$ and $x = (x_1, \dots, x_n)$.

Parameters are values that control the behavior of the system. In this case, w_i is the coefficient that is multiplied by feature x_i before summing up the contributions from all the features. w could be understood as a set of weights that determine how each feature affects the prediction. If a feature x_i receives a positive weight w_i, then increasing the value of that feature increases the value of the prediction $\hat{y}$. If a feature receives a negative weight, then increasing the value of that feature decreases the value of the prediction. If feature's weight is large in magnitude, then it has a large effect on the prediction. If a feature's weight is zero, it has no effect on the prediction. Then, the definition of the task T is to predict y from x by outputting $\hat{y} = w^T x$.

Next, a definition of the performance measure P is needed.

Suppose there is a design matrix of m example inputs that will not be used for training, only for evaluating how well the model performs. There is also a vector of regression targets providing the correct value of y for ach of these examples. Because this dataset will only be used for evaluation, it is called the test set. The design matrix of inputs are referred to as $X^{(test)}$ and the vector of regression targets as $y^{(test)}$, being $X^{(test)} = (X, \dots, X_m)$ and $y^{(test)} = (y_i, \dots, y_m)$ with $m \leq n$.

One way of measuring the performance of the model is to compute the mean squared error of the model on the test set. If $y^{(test)}$ gives the predictions of the model on the test set, then the mean squared error is given by

$$MSE_{test} = \frac{1}{m} \sum_{i}^{m} (\hat{y}^{(test)} - y^{(test)})_i^2 \qquad i = 1, \dots, m. \qquad 2.7.2.$$

Intuitively, it is observed that this error measure decreases to 0 when $\hat{y}^{(test)} = y^{(test)}$.

Then.

$$MSE_{test} = \frac{1}{m}\sum_{i}^{m} || \hat{y}^{(test)} - y^{(test)} ||_2^2 \qquad 2.7.3.$$

So, the error increases whenever the Euclidian distance between the predictions and the targets increases.

To make a machine learning algorithm, an algorithm that will improve the weights $\boldsymbol{w}$ in a way that reduces MSE_{test} when the algorithm can gain experience by observing a training set ($X^{(train)}$, $y^{(train)}$) needs to be created. One intuitive way of doing this is just to minimize the mean squared error on the training set MSE_{train}.

To minimize MSE_{train} it can simply be solved for where its gradient is equal to 0.

$$\nabla_w MSE_{train} = 0 \qquad 2.7.4$$

$$\Rightarrow \frac{1}{m}\nabla_w || \hat{y}^{(train)} - y^{(train)} ||_2^2 = \qquad 2.7.5$$

$$\Rightarrow \frac{1}{m}\nabla_w || X^{(train)} w - y^{(train)} ||_2^2 = \qquad 2.7.6$$

$$\Rightarrow \frac{1}{m}\nabla_w \left(X^{(train)} w - y^{(train)} \right)^T \left(X^{(train)} w - y^{(train)} \right) = 0 \qquad 2.7.7$$

$$\Rightarrow \frac{1}{m}\nabla_w \left(w^T {X^{(train)}}^T X^{(train)} w - 2\, w^T {X^{(train)}}^T y^{(train)} + {y^{(train)}}^T y^{(train)} \right) = 0 \qquad 2.7.8$$

$$\Rightarrow \left(2{X^{(train)}}^T X^{(train)} w - 2{X^{(train)}}^T y^{(train)} = 0 \right. \qquad 2.7.9$$

$$\Rightarrow w = \left({X^{(train)}}^T X^{(train)} \right)^{-1} {X^{(train)}}^T y^{(train)}. \qquad 2.7.10$$

The system of equations whose solution is given by the last formula is known as the normal equation. Evaluating this equation constitutes a simple learning algorithm.

It is worth noting that the term linear regression often used to refer to a slightly more sophisticated model with one additional parameter an intercept term b. In this model

$$\hat{y} = w^T x + b.$$

So, the mapping from parameters to predictions is still a linear function, but the mapping from features to predictions is now an affine function. This extension to affine functions means that the plot of the model's predictions still looks like a line, but it needs to not pass through the origin. Instead of adding the bias parameter b, it is possible to use the model with only weights, but augment x with an extra entry that is always set to 1. The weight corresponding to the extra 1 entry plays the role of the bias parameter.

The intercept term b is often called the bias parameter of the affine transformation. This terminology derives from the point of view that the output of the transformation is biased toward being b in the absence of any input. This term is different from the idea of a statical bias, in which a statistical estimation algorithm's expected estimate of a quantity is not equal to the true quantity.

Linear regression is of course an extremely simple and limited learning algorithm, but it provides an example of how a learning algorithm can work.

2.8 Gradient-Based Optimization

Most deep learning algorithms involve optimization of some sort. Optimization refers to the task of either minimizing or maximizing some function $f(x)$ by altering x. Most optimization problems are phrased in terms of minimizing $f(x)$. Maximization may be accomplished via minimization algorithm by minimizing $-f(x)$ [68].

The function to be minimized or maximized is called the objective function, or criterion. When it is minimized, it may be also called it the cost function, loss function, or error function.

The value that minimizes or maximizes a function would be denoted with a superscript *. For example, $x* = \arg min f(x)$.

Suppose there is a function $y = f(x)$, where both x and y are real numbers.

The derivative of this function is denoted as $f'(x)$ or $\frac{dy}{dx}$. The derivative $f'(x)$ gives the slope of $f(x)$ at the point of x. In other words, it specifies how to scale a small change in the input to obtain the corresponding change in the output: $f(x + \varepsilon) \approx f(x) + \varepsilon f'(x)$.

The derivative is therefore useful for minimizing a function because it tells how to change x to make a small improvement in y. For example, it is known that $f(x - \varepsilon\, sign(f'(x)))$ is less than $f(x)$ for small enough ε. It can be reduced $f(x)$ by moving x in small steps with the opposite sign of the derivative. This technique is called gradient descent [16].

When $f'(x) = 0$, the derivative provides no information about which direction to move. Points where $f'(x) = 0$ are known as critical points. A local minimum is a point where $f(x)$ is lower than at all neighboring points, so it is no longer possible to decrease $f(x)$ by making infinitesimal steps. A local maximum is a point where $f(x)$ is higher than at all neighboring points, so it is not possible to increase $f(x)$ by making infinitesimal steps. Some critical points are neither maxima nor minima. These are known as saddle points. See figure 2.2.

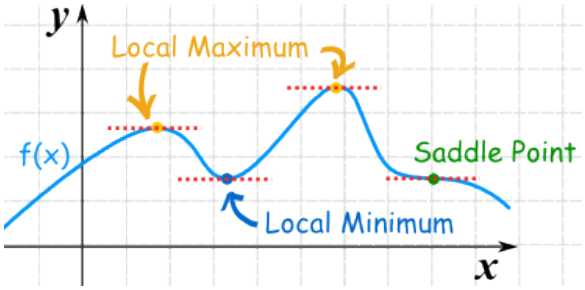


Figure 2.2. Type of critical points. [71].

A point that obtains the absolute lowest value of $f(x)$ is a global minimum. There can be only one global minimum or multiple global minima of the function. It is also possible for there to be local minima that are not globally optimal. In the context of deep learning, functions that may have many local minima that are not optimal, and many saddle points surrounded by very flat regions are optimized. All of this makes optimization difficult, especially when the input to the function is multidimensional. Therefore, usually it is settled for finding a value of f that is very low but not necessarily minimal in any formal sense. See figure 2.3 for an example.

Functions that have multiple inputs are often minimized: $f: \mathbb{R}^2 \rightarrow \mathbb{R}^2$. For the concept of minimization to make sense, there must still be only one (scalar) output.

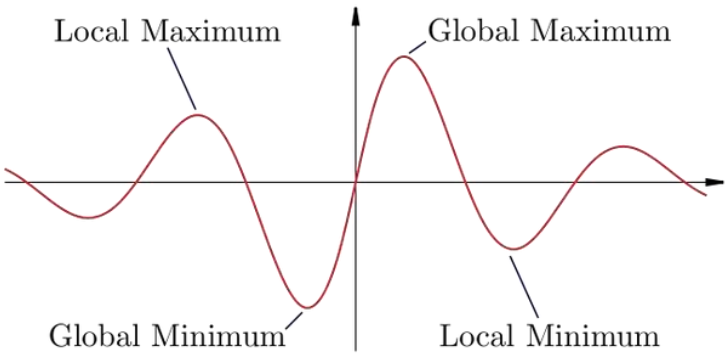


Figure 2.3: Global and local values. [70].

For functions with multiple inputs, the concept of partial derivatives must be used. The partial derivatives $\frac{\delta}{\delta x_i} f(x)$ measures how f changes as only the variables x_i increases at point x.

The gradient generalized the notion of the derivative to the case where the derivative is with respect to a vector: the gradient of f is the vector containing all the partial derivatives, denoted $\nabla_x f(x)$. Element i of the gradient containing all the partial derivative of f with respect of x_i. In multiple dimensions, where ε is the learning rate, a positive scalar determining the size of the step. Several different ways can be chosen. A popular approach is to set ε to a small constant. Sometimes, it is possible to solve for the step size that makes the directional derivative vanishes. Another approach is to evaluate $f(x - \varepsilon\nabla_x f(x))$ for severall values ε and choose the one that results in the smallest objective function value. This last strategy is called a line search.

Steepest descent converges when every element of the gradient is zero (or, in practice, very close to cero). In some cases, sometimes running this iterative algorithm could be avoided and just jump directly to the critical point by solving the $\nabla_x f(x)$ equation for x.

Although gradient descent is limited to optimization in continuous spaces, the general concept of repeatedly making a small move (that is approximately the best small move) toward better configurations can be generalized to discrete spaces. Ascending an objective function of discrete parameters is called hill climbing [17]. See figure 2.4 as an example of gradient descent algorithm.

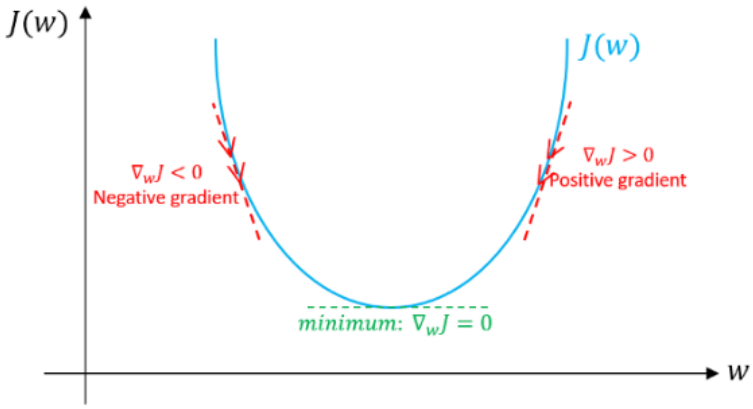


Figure 2.4: Gradient descent algorithm. [69].

2.9 Stochastic Gradient Descent

Nearly all deep learning is powered by one very important algorithm: Stochastic Gradient Descent (SGD). Stochastic gradient descent is an extension of the gradient descent algorithm.

In stochastic gradient descent we do not require the update direction to be based exactly on the gradient. Instead, we allow the direction to be a random vector and only require that its expected value at each iteration will equal the gradient direction. Or, more

generally, we require that the expected value of the random vector will be a sub gradient of the function at the current vector.

$$Stochastic\ Gradient\ Descent\ algorithm\ (SGD)\ for\ minimizing$$

$$f(w)$$

parameters: $Scalar\ \eta > 0, integer\ T > 0$
initialize: $w^1 = 0$
for $t = 1, 2, \ldots, T$
choose v_t at random from a distribution such that $E[v_t \mid w^{(t)}] \in \partial f(w^{(t)})$
update $w^{(t+1)} = w^{(t)} - v_t$

output $\overline{w} = \frac{1}{T}\sum_{t=1}^{T} w^{(t)}$

2.10 Biological Neuron

A simplified biological neuron is shown in Figure 2.5. In the most basic sense, the biological neuron consists of a cell body with one axon and many dendrites. The connections between neurons are called synapses, and this is the connection between the axon of one neuron and the dendrite of another. A single neuron has many dendrites and one axon, making every neuron a multiple input single output building block. Connecting many neurons together forms the network of the brain.

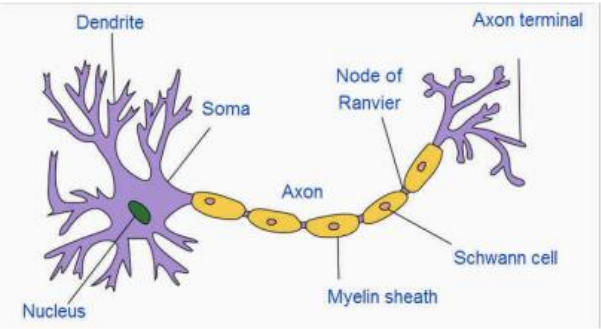


Figure 2.5: Simplified biological neuron. [18].

2.11 Artificial Neuron

The artificial neuron is based on the biological version. It consists of inputs, weights and a bias, a summation, an activation function, and the output as shown in Figure 2.6. The output of a neuron can be called the activation of a neuron. The summation basically does a linear transformation on the inputs by its weights and bias as in equation 2.11.1.

The non-linearity is introduced by the activation function which decides how much of the information from this sum to pass through to the output. There are diverse types of activation functions, also linear activation. Neural networks using only linear activations are essentially linear regression models.

$$\sum_{i=1}^{n} w_i * x_i + b\,. \qquad 2.11.1$$

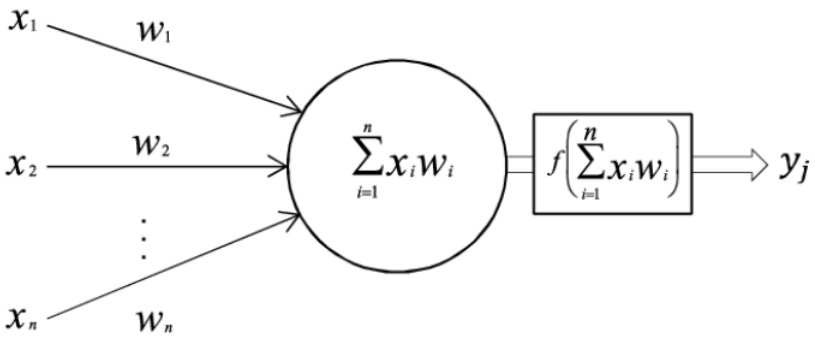


Figure 2.6: Artificial neuron. [74].

2.11.1 Capacity, overfitting and underfitting

The central challenge in machine learning is that the algorithm must perform well on $new, previously\ unseen$ inputs -- not just those on which the model was trained. The ability to perform well on previously unobserved inputs is called generalization [77].

Typically, when training a machine learning model, the training data set is accessible and is possible to compute some error measure on the training set, called training error, and the main purpose is to reduce this training error. So far, what is described is simply an optimization problem. What separates machine learning from optimization is that in machine learning, the generalization error, also called test error, should be as low as possible. Generalization error is defined as the expected value of the error on a new input. Here the expectation is taken across different possible inputs.

Typically, the generalization error of machine learning model is estimated by measuring its performance on a test set of examples that were collected separately from the training set.

In the linear regression example, the model is trained by minimizing the training error.

$$\frac{1}{m^{(train)}} \left|\left| X^{(train)} w - y^{(train)} \right|\right|_2^2. \qquad 2.11.2$$

but it is more important to measure the test error.

$$\frac{1}{m^{(test)}} \left|\left| X^{(test)} w - y^{(test)} \right|\right|_2^2. \qquad 2.11.3$$

How is the performance affected on the test set when only the training set is observed? The field of statistical learning theory provides some answers. If the training and the test set are collected arbitrarily, there is indeed little that can be done. If it can make some assumptions about how the training and test set are collected, then is possible to make some progress.

These two factors correspond to the two central challenges in machine-learning underfitting and overfitting. Underfitting occurs when the model is not able to obtain a sufficiently low error value on the training set. Overfitting occurs when the gap between the training error and test error is too large [78].

Altering its capacity of whether it is more likely to overfit or underfit can control a model. 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. Models with high capacity can overfit by memorizing properties of the training set that do not serve them well on the test set.

On way to control the capacity of a learning algorithm is by choosing its hypothesis space, the set of functions that the learning algorithm can select as being the solution. For example, the linear regression algorithm as the set of all linear functions of its input as its hypothesis space. Linear regression can be generalized to include polynomials, rather than just linear functions, in its hypothesis space. By doing so, it increases the model's capacity.

A polynomial of degree 1 gives the linear regression model with the already familiar prediction.

$$\hat{y} = b + w.$$

By introducing x^2 as another feature provided to the linear regression model, it enables to learn a model that is quadratic as a function of x.

$$\hat{y} = b + w_1 x + w_2 x^2.$$

Though this model implements a quadratic function of its $input$, the output is still a linear function of the $parameters$, so it can still use the normal equations to train the model in closed form. More powers of x can be added as additional feature, for example, to obtain a polynomial of degree 9.

$$\hat{y} = b + \sum_{i=1}^{9} w_i w_1 x^i.$$

Machine learning algorithms will generally perform best when their capacity is appropriate for the true complexity of the task they need to perform and the amount of training data they are provided with. Models with insufficient capacity are unable to solve

complex tasks. Model with high capacity can solve complex tasks, but when their capacity is higher than needed to solve the present task, they may overfit.

2.11.2 Activation Functions

Different activation functions are used for different problems. The following sub sections give a brief overview of the most common ones. The information about each function is taken from Patterson, J. [19, p. 65]. All plots of the following activation functions are made in Python.

2.11.2.1 Linear

Figure 2.7 shows the linear activation function $f(x) = x$. When using this activation function, the output is simply proportional to the input. It basically lets the signal through.

```
import matplotlib.pyplot as plt
plt.style.use('seaborn-whitegrid')
import numpy as np
fig = plt.figure()
ax = plt.axes()
ax = plt.axes()
x = np.linspace(0, 10, 10)
ax.plot(x, (1 * (x) + 0))
plt.show()
```

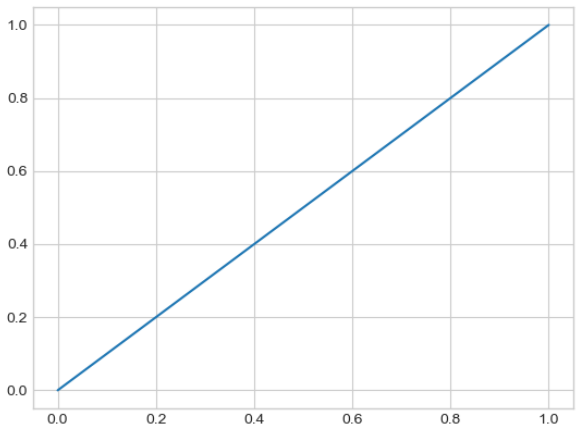


Figure 2.7: Linear Activation Function.

2.11.2.2 Rectified Linear

The $Rectified\ Linear\ Unit\ (ReLU)$ function shown in the equation below, illustrated in Figure 2.8, is the most common activation function due to its simplicity and good results. A subset of neurons fire at the same time, and this makes the network sparser, improving efficiency. With

a uniform initialization of the weights, around *50%* of the hidden neurons will fire according to *Glorot Xavier (2011) [21].* Sparsity is discussed in more depth later in the literature review.

$$f(x) = max(0, x).$$

```
import matplotlib.pyplot as plt
plt.style.use('seaborn-whitegrid')
import numpy as np
fig = plt.figure()
ax = plt.axes()
ax = plt.axes()
x = np.linspace(-10, 10, 100)
ax.plot(x, np.maximum(.01 * (x), 0))
plt.show()
```

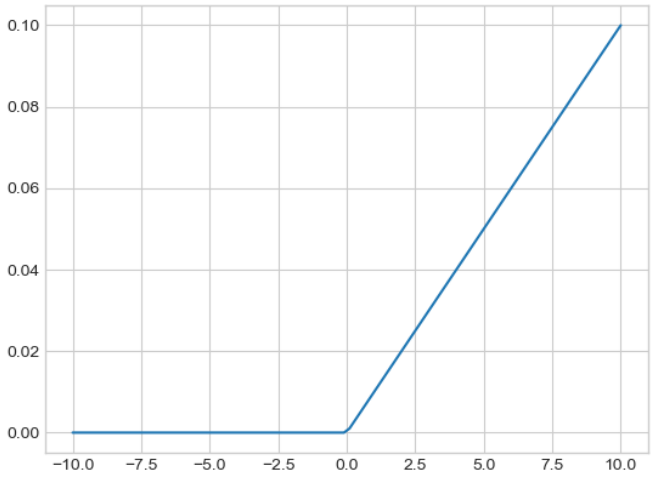


Figure 2.8: Rectified Linear Activation Function.

There is also a $Leaky\ ReLU$. It is similar to the $ReLU$ function except that when x is less than 0 the function has a small negative slope. Dying ReLU [19, p. 70] can be a problem with standard $ReLU$. $Leaky\ ReLU$ prevents neurons from being totally inactive, or to have dead neurons, which means that the neurons are inactive for all the input samples. Solving dead neurons and other issues are discussed in solving internal covariate shift in deep learning with linked neurons by Riera C. [22].

$$f(x) = \begin{cases} x & x > 0, \\ 0.01x & x \leq 0. \end{cases} \quad 2.3$$

There is also a $Parameterised\ ReLU$ function, shown in Equation 2.4. The a, in the equation, decides the slope for negative values of x. The network trains the added parameter. This activation function can be used when the Leaky ReLU does not solve the problem of $dead\ neurons$.

$$f(x) = \begin{cases} x & x > 0, \\ ax & x \leq 0. \end{cases} \quad 2.4$$

2.11.2.3 Softplus

Figure 2.9 shows the Softplus activation function. It is also a version of the ReLU function. The standard ReLU function is graphed with a red dotted line. In contrast to the ReLU this function is continuously differentiable.

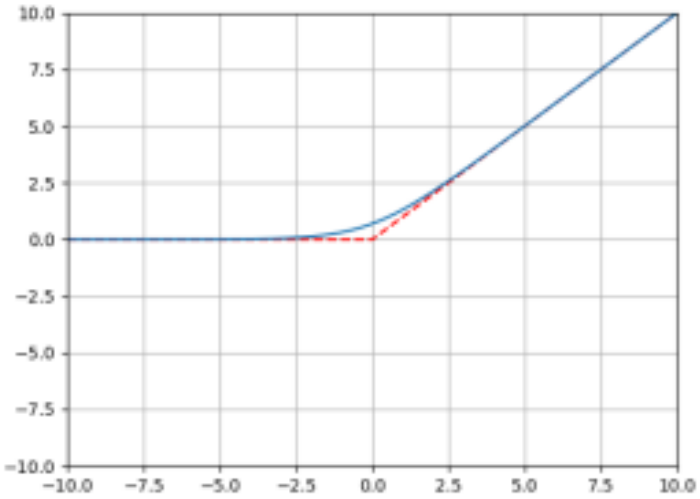


Figure 2.9: Softplus activation function.

2.11.2.4 Sigmoid

The sigmoid function, in equation below, is also a very popular activation function. It squeezes the output between 0 and 1. It is continuously differentiable. The gradient of this function is highest around 0 and flattens out for higher or lower input values. Meaning that when the network falls into that region of the graph it learns slower and slower, i.e. the vanishing gradient problem. The sigmoid function is shown in Figure 2.10.

$$f(x) = \frac{1}{1 + e^{-x}}.$$

```
import matplotlib.pyplot as plt
plt.style.use('seaborn-whitegrid')
import numpy as np
fig = plt.figure()
ax = plt.axes()
x = np.linspace(-10, 10, 200)
ax.plot(x, 1/(1+np.exp(-x)))
plt.show()
```

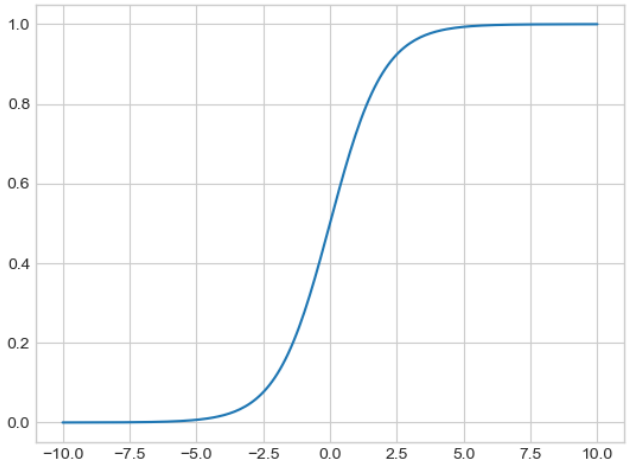


Figure 2.10: Sigmoid activation function.

2.11.2.5 Softmax

The softmax function, shown in the equation below, is also like the sigmoid function. It outputs continuous values from 0 to 1 and is often used at the output layer as a classifier, because it outputs the probabilities distributed over the number of classes. I.e. summing up all the probabilities add up to 1 or 100%.

Any time it is desired to represent a probability distribution over a discrete variable with n possible values, the softmax function could be used. This can be seen as the generalization of the sigmoid function, which was used to represent a probability distribution over a binary variable.

Softmax functions are most often used as the output of a classifier, to represent the probability distribution over n different classes. More rarely, softmax functions can be used inside the model itself.

$$f(x_i) = \frac{e^{x_i}}{\sum_{i=1}^{n} e^{x_i}} \; for \; i = 0,1,2, \dots, n.$$

2.11.2.6 Binary Step Function

The binary step function, shown in the following equation, is basically just a threshold that states if the neuron should be active or not. It is shown in Figure 2.11.

$$f(x) = \; 1, for \; x \geq 0.$$

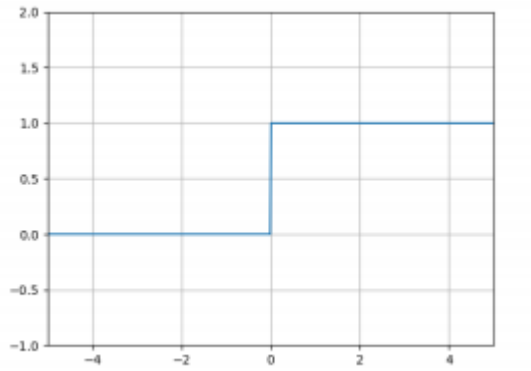


Figure 2.11: Binary step function.

2.12 Artificial Neural Networks (ANN)

Basic artificial neurons are the building block of an artificial neural network. An ANN consists of three different types of layers: the input layer, the hidden layers, and the output layer [23]. There may be many hidden layers in the network. Figure 2.10 shows a basic ANN with one hidden layer with four neurons, three inputs, and three outputs. A network where each layer has multiple neurons and all the neurons in one layer are connected to the neurons in the next layer is called a fully connected network or multi-layer perceptron (MLP).

A deep neural network, a NN with more than two layers, is to a considerable extent based on statistics and linear algebra. This review will not go in depth on these topics since they are broadly covered in other texts like Patterson, J. [19], [24]

2.12.1 Forward Propagation

The neural network feeds (forward) information from the inputs through the hidden layers to the outputs. This movement of information through the network is called forward propagation.

2.12.2 Weights & Biases

Between layers in the NN, the output of neurons in one layer, or the activations of these neurons, are connected to the input of neurons in the next layer, each connection associated with a weight. These weights are the tuning knobs of the network. It could be said that the weight is the strength of the connection between two neurons, or how much of the activation from one neuron that is carried through to the next. This can be illustrated using different thickness of the connections like in Figure 2.12. The weights, and the bias that basically offsets the activation of the neuron, are the adjustable parameters of a NN.

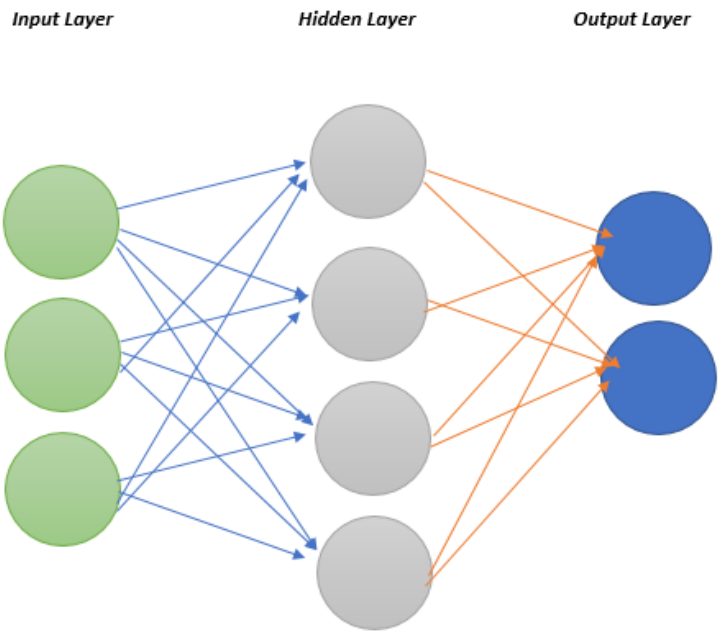


Figure 2.12: Basic Artificial Neural Network.

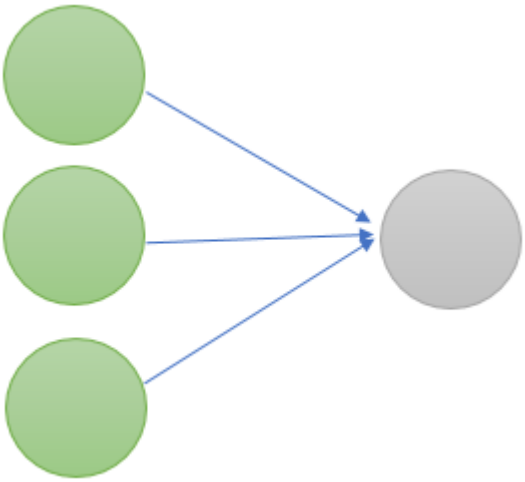


Figure 2.13: Weights between neuron i a NN.

2.12.3 Optimization

The parameters, weights and biases, in a neural network are updated using a training data set. Initially, the parameters of the network can be assigned randomly. With more training data, the model will more accurately resemble the real system. Machine learning (*ML*) finds a way to represent data based on the training set. It does not try to match the data to a mathematical model, i.e. it is not told what patterns to look for, but updates the parameters of the model based on a cost function which represents the differences between the desired values, i.e. the labels of the training data, and the actual output provided by the network

The weights and biases are updated such that the average costs of the entire training example are minimized the most. As seen in Figure 2.14, using a simple linear function can be an under fit of the data as it in many cases does not represent the data very well. There is also a problem with overfitting in machine learning. Overfitting the model will give a very low error in the training data but does not provide a generalized solution to the problem. This can result in a significant decrease in accuracy on the test set, i.e. on new unseen inputs after training, as it also will account for noise and outliers in the training set.

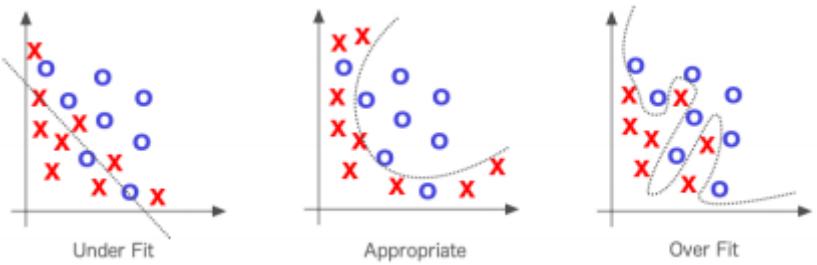


Figure 2.14: Underfitting & overfitting. [19, p. 27]

The process of updating the parameters of the model is called parameter optimization and is basically adjusting the weights based on the cost function. The weights are adjusted such that the cost function decreases most efficiently. A popular method is the first order optimization using gradient descent, as it is easy to use and less time consuming and computationally heavy than for example using the hessian for second order optimization.

2.12.4 Backpropagation

Each step takes the average cost of all the training samples. Each activation is a weighted sum of all the activations of the previous layer and a bias. I.e. the error is dependent of these weights, the bias and the activations from the last layer. Since the activations are dependent of the previous layer and cannot be directly altered, it could be back propagated though the network, adjusting the weights. Using every training sample for every gradient descent step takes a long time to compute. Stochastic gradient descent (SGD) is used to make this process faster. It basically randomizes the order of the input data and splits it up into mini batches. A step is computed according to the mini batch. This does not give exactly the correct direction in the high dimensional space to move in as it does not accord for the whole training, but using a subset gives a good approximation.

2.12.5 Batch size, Iterations, and Epochs

These terms are easily explained using an example: If there are 10000 training samples divided into 10 batches. The batch size is 1000, and there are 10 iterations for each epoch. The number of epochs represents the number of times the model has trained on all the training samples in the data set.

2.12.6 Training Phase and Inference.

Normally the dataset is split into a training set, a validation set, and a test set. During the training phase, the training data is used to update the parameters of the network. The validation data is used during training to monitor the training process and to detect e.g. overfitting. Inference is when the trained model is tested with new unseen data. E.g. when a trained model is deployed and used in a live application.

This sub section is mostly based on the paper from Xavier Glorot, Antoine Bordes, and Yoshua Bengio: $Deep\ Sparse\ Rectifier\ Neural\ Networks$ [25]. Using activation functions like $ReLU$ which outputs 0 for negative input values naturally makes the network sparse. This can have some advantages over a non-sparse network. Pruning is another technique to achieve sparsity. It identifies non-important neurons and sets them to zero.

"We argue here that if one is going to have fixed-size representations, then sparse representations are more efficient (than non-sparse ones) in an information-theoretic sense, allowing for varying the effective number of bits per example" [26]. Sparse representations allow the network to vary the effective dimension and required precision of a given input. Using $ReLU$ the output is a linear representation of the subset of active neurons.

Using a sparse NN results in a less entanglement network making it easier to identify the factors explaining the variations in the data. Sparse NN gives a computational advantage in comparison to a dense network and it can contribute to reducing the problem of overfitting. Sparse networks are becoming more popular, as the accuracy of the NNs do not decrease significantly when introducing a sparser network. "Maximum sparsity is obtained by exploiting both inter-channel and intra-channel redundancy, with a fine-tuning step that minimize the recognition loss caused by maximizing sparsity. This procedure zeros out more than 90% of parameters, with a drop of accuracy that is less than 1% on the ILSVRC2012 dataset" [27].

2.12.7 Dropout

Dropout is a technique used under training to avoid overfitting. As the name suggests, it drops out random neurons in the hidden layers. This means that the neurons are temporarily removed from the network. An illustration of dropout neurons is shown in Figure 2.15 taken from Dropout: A Simple Way to Prevent Neural Networks from Overfitting [28]. For each presentation of each training case a different reduced network is used. During the inference phase all neurons are active.

2.12.8 Data Augmentation

Having too few training samples is a frequent problem using neural networks, as they often need many samples to create a good generalization of the problem. Data augmentation is creating new input data from already given inputs increasing the number of samples. This is useful in application, which has a restricted a number of training samples available. Examples of data augmentation on images are mirroring, rotations, random cropping and color shifting.

2.12.9 Batch Normalization

It is common to normalize the data before inputting it to the NN. Batch normalization normalizes the mean of the layer's output activation close to 0 and its standard deviation close to 1. This method is commonly used to accelerate the training of CNNs.

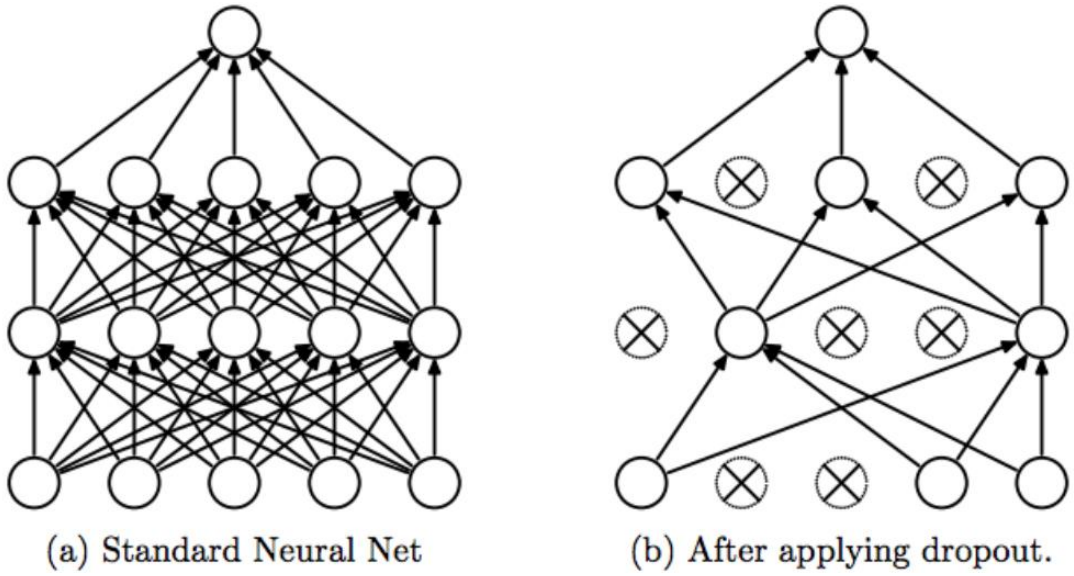


Figure 2.15: Illustration of dropout in a NN. [29].

"The training is complicated by the fact that the inputs to each layer are affected by the parameters of all preceding layers so that small changes to the network parameters amplify as the network becomes deeper" [39]. As the title says: $Batch\ Normalization$: Accelerating Deep Network Training by Reducing Internal Covariate Shift, and the paper defines the internal covariate shift as: "The change in the distribution of network activations due to the change in network parameters during training". As the problem becomes much more severe as the networks get deeper, batch normalization layers are more needed in these cases. The network also trains the two extra parameters introduced by batch normalization.

2.13 Multilayer Perceptron (MLP)

The goal of the feedforward network is to approximate some function f^* . For example, a classifier, $y = f^*(x)$ maps an input x to a category y. A feedforward network defines a mapping $y = f(x; \theta)$ and learns the value of the parameters θ the result in the best function approximation.

The models are called feedforward because information flows through the function being evaluated from x, through the intermediate computations used to define f, and finally to the output y. There are no feedback connections in which outputs of the model are fed back into itself. When feedforward neural networks are extended to include feedback connections. They are called recurrent neural networks, as presented later in this chapter.

Feedforward neural networks are called networks because typically they are represented by composing many different functions together. The model is associated with a directed acyclic graph describing how the functions are composed together. For

example, three functions $f^{(1)}, f^{(2)}\ and\ f^{(3)}$ are connected into a chain, to form $f(x) = f^{(3)}\left(f^{(2)}\left(f^{(1)}(x)\right)\right)$. These chain structures are the most commonly used structures of neural networks. In this case, $f^{(1)}$ is called the first layer of the network, $f^{(2)}$ is called the second layer, and so on.

The overall length of the chain gives the depth of the model. The name "deep learning" arose from this terminology. The final layer of the feedforward network is called the output layer. During neural network training, $f(x)$ is driven to match $f^*(x)$. The training data provides noisy, approximate examples of $f^*(x)$ evaluated at different training points. Each example x is accompanied by a label $y \approx f^*(x)$.

The training examples specify directly what the output layer must do at each point x; it must produce a value that is close to y. The behavior of the other payers is not directly specified by the training data. The learning algorithm must decide how to use those layers to produce a value that is close to y. The behavior of the other layers is not directly specified by the training data.

The learning algorithm must decide how to use those layers to produce the desired output, but the training data does not say what each individual layer should do. Instead, the learning algorithm must decide how to use these layers to best implement an approximation of f^*. Because the training data does not show the desired output for each of these layers. They are called hidden layers.

Finally, these networks are called $neural$ because they are loosely inspired by neuroscience. Each hidden layer of the networks is typically a vector valued. Each element of the vector may be interpreted as playing a role analogous to a neuron. Rather than thinking of the layer as representing single vector-to-vector function, it could be explained as a layer of many units that act in parallel, each representing a vector-to-scalar function as well.

2.14 Convolutional Neural Network (CNN)

Convolutional neural networks are a type of neural network that has gained a lot of momentum lately partly due to its great ability to classify objects in images. It learns to recognize features through convolution. It utilizes that pixels closer together in an image are more related to each other than pixels far apart. For classifying images, MLPs does not scale very well. It takes the input as a one-dimensional vector and passes the data through the fully connected hidden layers. This is fine for small images. 10 pixels by 10 pixels image and 3 RGB channels will give 300 weights per neuron in the first hidden layer. A 640x480 pixels image and 3 RGB channels will give on the other hand 921600 weights per neuron in the first hidden layer.

The CNN basically consists of several types of layers stacked on top of each other.

There is no given way to stack the different layers, it is up to the designer. Using object classification is a very intuitive example going through the basics of CNNs, but they can be used on other types of data like text or sound, they are even being used to make computers learn to play video games [48].

In the following sub sections different types of layers, input layer, convolutional layer, pooling layer, fully connected layer, and batch normalization will be described.

2.14.1 Input layer

The input layer stores the raw input data. It is a three-dimensional input consisting of the width and height of the image, and the color channels, typically three for RGB, represent the depth.

2.14.2 Convolutional layer

2.14.2.1 The convolution Operation.

In its most general form, convolution is an operation on two functions of a real valued argument. To motivate the definition of convolution, see examples of two functions.

Suppose the location of a spaceship is tracked with a laser sensor. This laser sensor provides a single output $x(t)$, the position of the spaceship at time t. Both x and t are real valued, that is, a different reading from the laser sensor can be given at any instant in time.

Now, if supposedly the laser sensor is somewhat noisy, to obtain a less noisy estimate of the spaceship's position, several measurements should be averaged. Of course, more recent measurements are more relevant, so a weighted average that gives more weight to recent measurement is wanted. It can be done this with a weighting function $w(a)$, where a is the age of a measurement. If such weighted average operation was applied at every moment, a new function s is obtained, providing a smoothed estimate of the position of the spaceship.

$$s(t) = \int x(a)w(t-a)da. \qquad 2.14.1$$

This operation is called convolution. The convolution operations are typically denoted with an asterisk:

$$s(t) = (x * w)(t). \qquad 2.14.2$$

In this example, w needs to be a valid probability density function, or the output will not be a weighted average. Also, w needs to be 0 for all negative arguments or it will consider the future, which is presumably beyond any capabilities. These limitations are particular to this example. In general, convolution is defined for any functions for which the above integral is defined and may be used for other purposes besides taking weighted averages.

In convolutional network terminology, the first argument (in this example, the function x) to the convolution if often referred to as the input, and the second argument (in this example the function w) as the Kernel. The output is sometimes referred to as the feature map.

In the example above, the idea of a laser sensor that can provide measurements at every instant is not realistic. Usually, when data is running on a computer, time will be discretized, and our sensor will provide data at regular intervals. In this example, it might be more realistic to assume that the laser provides a measurement per second. The time index t can then take on only integer values. If x and w are defined only on integer t, the discrete convolution can be defined:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a)\,. \qquad 2.14.3$$

In machine learning applications, the input is usually a multidimensional array of data, and the kernel is usually a multidimensional array of parameters that are adapted by the learning algorithm. It could be referred to these multidimensional arrays as tensors. Because each element of the input and kernel must be explicitly stored separately, it is usually assumed that these functions are zero everywhere but in the finite set of points for which the values are stored. This means that in practice, the infinite summation can be implemented as a summation over a finite number of array elements.

Finally, often convolution is used over more than one axis at a time. For example, if a two-dimensional image I is used as the input, the best option is a two-simensional kernel K:

$$S(i,j) = (I * K)(i,j) = \sum_{m}\sum_{n} I(m,n)\, K(i-m, j-n). \qquad 2.14.4.$$

Convolution is commutative, meaning it can equivalently be written.

$$S(i,j) = (K * I)\,(i,j) = \sum_{m}\sum_{n} I\ \ (i-m, j-n)K(m,n). \qquad 2.14.5.$$

Usually the latter formula is more straightforward to implement in a machine-learning library, because there is less variation in the range of valid values of m and n.

The commutative property of convolution arises because the kernel relative to the input has been flipped, in the sense that as m increases, the index into the input increses, but the index to the kernel decreases. The only reason to flip the kernel is to obtain the commutative property. While the commutative property is useful for writing proofs, it is not usually an important property of a neural network implementation. Instead, many neural network libraries implement a related function called the cross-correlation, which is the same as convolution but without flipping the kernel:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I\ (i+m, j+n) K(m,n). \qquad 2.14.6.$$

The convolution layer is the key layer of the CNN. It uses filters, or kernels, that basically is a smaller image than the input. Convolution is done with a part of the input and the kernel. This is done in a sliding window manner, ultimately covering the whole input image, as illustrated in Figure 2.16. It is done for every depth of the input. The output from this process is called a *feature map or an activation map*.

The region of the input the feature map is looking at, is called the *receptive field*. Each filter results in a feature map. The activation map for each filter are stacked outputting a 3-dimensional tensor. As the filters are trained they learn to recognize edges and patterns, and deeper in the network they can recognize more advanced shapes. The input to a convolution layer is either the NN input or the feature map output from another convolution layer

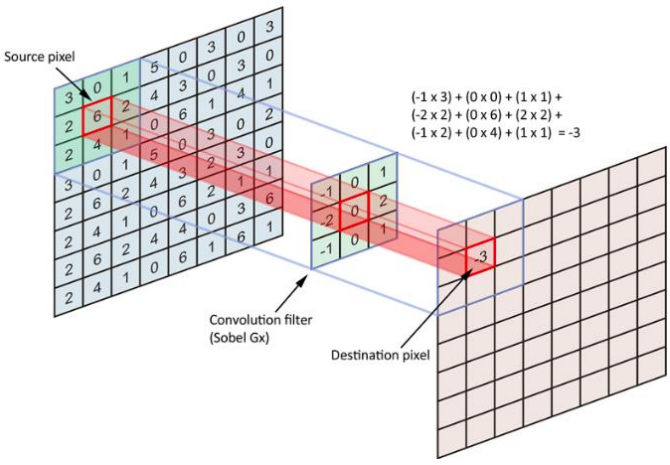


Figure 2.16: Convolution Engineering. [31].

Very commonly used in CNNs are the $ReLU$ activation functions. This layer basically takes all the negative inputs and sets them to zero. The $ReLU$ layer has no hyperparameters, i.e. parameters that are chosen by the designer.

The pooling layer reduces the size of the data. The most common version is $max\,pooling$, which outputs the maximum value of the given window size and ignores the rest. It does this operation over the whole input. The designer chooses the stride. With a common window size of 2x2 and a stride of 2 the reduction would be 75% of the original size like shown in Figure 2.17. Pooling doesn't care about where in that window the maximum value is, which makes it a little less sensitive to the position and helps to control $overfitting$.

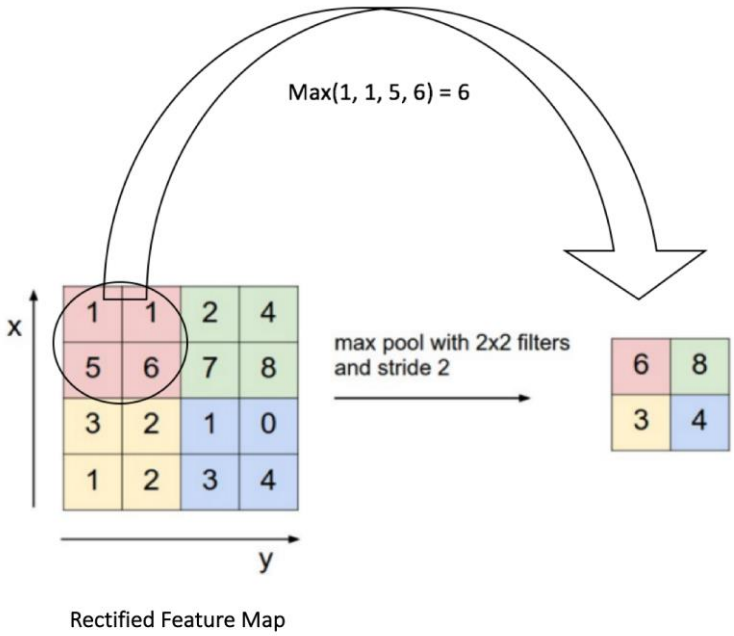


Figure 2.17 Max Pooling layer example [33].

Typically, at the output, or classification of the CNN, there are one or multiple fully connected layers. The classifier outputs probabilities for the different classes. Figure 2.18 shows an illustration of the famous CNN AleNet [33].

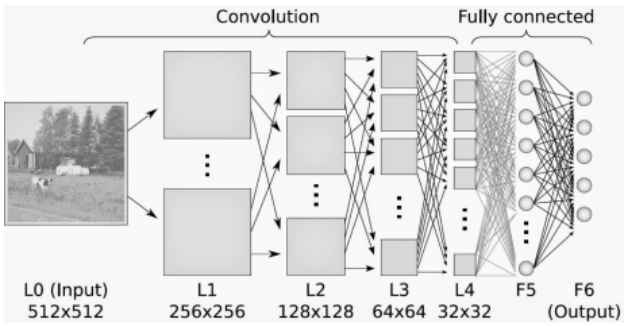


Figure 2.18: Fully connected neural network. [32].

2.15 Recurrent Neural Network (RNN)

Recurrent neural networks, or RNNs [34], are a family of neural networks for processing sequential data like $x^1, \ldots, x^\tau$, they can scale to much longer sequences than would be practical for networks without sequence-based specialization. Most recurrent networks can also process sequences of variables lenght.

To go from multilayer networks to recurrent networks, it is helpful to revise one of the early ideas found in machine learning and statistical models of the 1980s. Sharing parameters across different parts of a model. Parameter sharing makes it possible to extend and apply the model to examples of different forms and generalize across them. Such sharing is particularly important when a specific piece of information can occur at multiple positions within the sequence. For examples, the two sequences considered "I went to Budapest in 2016" and "In 2016 I went to Budapest". If a machine-learning model is asked to read each sentence and extract the year in which the narrator went to Budapest, the year 2016 is what has to be considered as the relevant piece of information, whether it appears in the sixth word or in the second word of the sentence. A traditional fully connected feedforward network would have separate parameters for each input feature, so it would need to learn all the rules of the language separately at each position in the sentence. By comparison, a recurrent neural network shares the same weights across several time steps.

A related idea is the use of convolution across a 1-D temporal sequence. This convolutional approach is the basis for time-delay neural networks [35], [36]. The convolution operation allows a network to share parameters across time but is shallow. The output of convolution is a sequence where each member of the output is a function of a small number of neighboring members of the input. The idea of parameter sharing manifests in the application of the same convolution kernel at each time step. Recurrent networks share parameters in a different way. Each member of the output is a function of the previous members of the output. Each member of the output is produced using the same update rule applied to the previous outputs.

This recurrent formulation results in the sharing of parameters through a very deep computational graph.

For the simplicity of exposition, RNNs are referred to as operating on a sequence that contains vectors $x(t)$ with the time step index t ranging from 1 to τ. In practice, recurrent networks usually operate on minibatches of such sequences, with a different sequence length τ for each member of the minibatch. To simplify notation, minibatch indices have been omitted. Moreover, the time step index need not literally refer to the passage of time in the real world. Sometimes it refers only to the position in the sequence

This chapter extends the idea of a computational graph to include cycles. These cycles represent the influence of the present value of a variable on its own value at a future time step. Such computational graphs facilitate the definition of recurrent neural networks. Many different ways to construct, train, and use recurrent neural networks are then described.

Figure 2.19: The classical dynamical system described by equation 2.15.1. [75].

2.15.1 Unfolding Computational Graphs

A computational graph is a way to formalize the structure of a set of computations, such as those involved in mapping inputs and parameters to outputs and loss. In this section it is explained the idea of unfolding a recursive or recurrent computation into a computational graph that has a repetitive structure, typically corresponding to a chain of events. Unfolding this graph results in the sharing of parameters across a deep network structure. For example, consider the classical form of a dynamical system:

For example, if the classical form of a dynamical system is considered:

$$s^t = f\left(s^{(t-1)}\ ;\ \theta\right) \qquad 2.15.1$$

where s^t called the state of the system.

Equation 2.15.1 is recurrent because the definition of s at time t refers to the same definition at $t-1$.

For a finite number of time steps τ, the graph can be unfolded by applying the definition $\tau-1$ times. For example, if equation 2.15.1 if unfolded for $\tau = 3$ time step, the result is.

$$s^{(3)} = f(s^2\ ;\theta) \qquad 2.15.2$$

$$s^{(3)} = f(f(s^1\ ;\ \theta)\ ;\ \theta). \qquad 2.15.3$$

Unfolding the equation by repeatedly applying the definition in this way has yielded an expression that does not involve recurrence. Such an expression can now be represented by a traditional directed acyclic computational graph. The unfolded computational graph of equation 2.15.1 and equation 2.15.3 is illustrated in figure 2.19.

Armed with the graph-unrolling and parameter-sharing ideas of section 2.15.1, it is possible to design a wide variety of recurrent neural networks.

Some examples of important design patterns for recurrent neural networks include the following.

- Recurrent networks that produce an output at each time step and have recurrent connections between hidden units, illustrated in figure 2.20

- Recurrent networks that produce an output at each time step and have recurrent connections only from the output at one-time step to the hidden units at the next time step, illustrated in figure 2.21

- Recurrent networks with recurrent connections between hidden units, that read an entire sequence and then produce a single output, illustrated figure 2.22

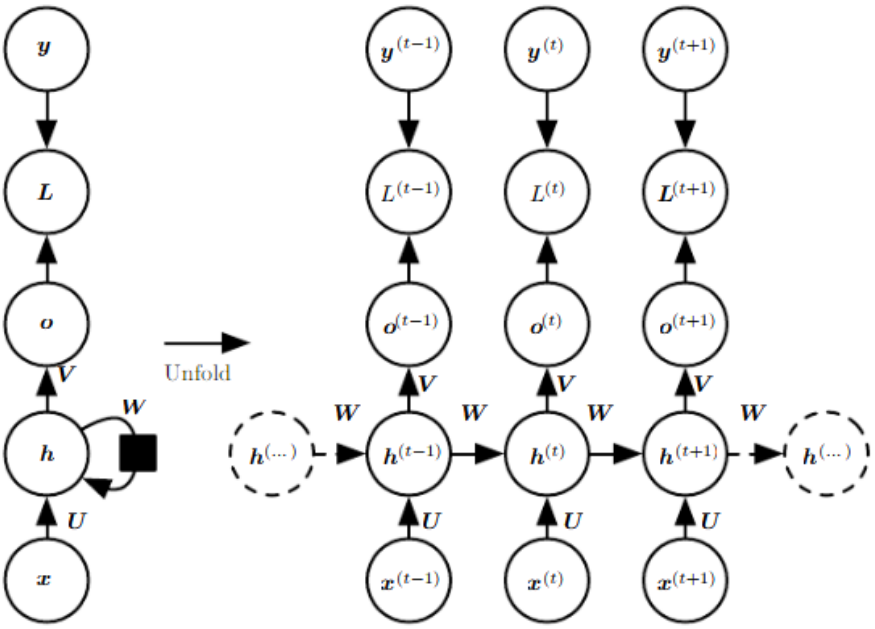


Figure 2.20: Graph to compute the training loss of a recurrent network. [75].

A loss function L measures how far each $\boldsymbol{o}$ is from the corresponding training target y. When using $softmax$ outputs, it is assumed that $\boldsymbol{o}$ is the unnormalized log probabilities. The L internally computes $\hat{y} = softmax(\boldsymbol{o})$ and compares this to the target y. The RNN has input to hidden connections parametrized by a weight matrix U. hidden to hidden recurrent connections parametrized by a weight matrix W, and hidden to output connection parametrized by a weight matrix V.

The recurrent neural network of figure 2.20 and equation 2.15.4 is universal in the sense that any function computable by a Turing machine[6] can be computed by such a recurrent network of a finite size. The output can be read from the RNN after several time steps that is asymptotically linear in the number of time steps used by the Turing machine and asymptotically linear in the length of the input [37], [38]. The functions computable by a Turing machine are discrete, so these results regard exact implementation of the function, not approximations. The RNN, when used as a Turing machine, takes a binary sequence as input, and its outputs must be discretized to provide a binary output. It is possible to compute all functions in this setting using a single specific RNN of finite size. [39].

[6]A Turin machine is a finite-state machine associated with a special kind of environment in which it can store and later recover sequences of symbols.

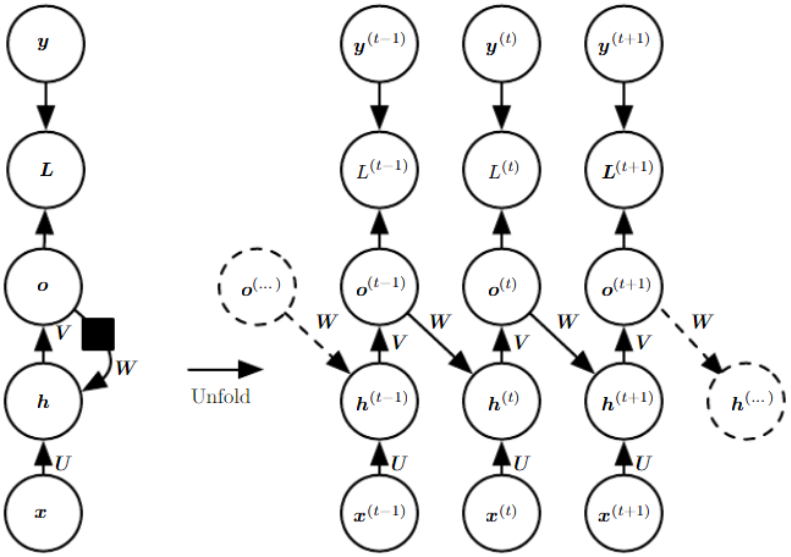


Figure 2.21: A RNN whose only recurrence is the feedback connection from the output to the hidden layer. [75]

The Input of the Turing machine is a specification of the function to be computed, so the same network that simulates this Turing machine is sufficient for all problems. The theorical RNN used for the proof can simulate an unbounded stack by representing its activations and weights with rational number of unbounded precision.

The forward propagation equations for a RNN are developed, depicted in figure 2.20 The figure does not specify the choice of activation function for the hidden units. The hyperbolic tangent activation function is assumed. Also, the figure does not specify exactly what form the output and loss function take. Here it is assumed that the output is discrete, as if the RNN is used to predict words or characters. A natural way to represent discrete variables is to regards the output o as giving the unnormalized log probabilities of each possible value of the discrete variable. It is possible then to apply the $softmax$ operation as a post-processing step to obtain a vector $\hat{y}$ of normalized probabilities over the output. Forward propagation begins with a specification of the initial state $h^{(0)}$. Then, for each time step from $t = 1\ to\ t = \tau$, the following update equations are applied:

$$a^{(t)} = b + Wh^{(t-1)} + Ux^{(t)} \quad 2.15.4$$

$$h^{(t)} = \tanh\left(a^{(t)}\right) \quad 2.15.5$$

$$o^{(t)} = c + Vh^{(t)} \quad 2.15.6$$

$$\hat{y} = softmax\left(o^{(t)}\right) \quad 2.15.7$$

where the parameters are the bias vectors b and c along with the weight matrices $U, V\ and\ W$, respectively, for input to hidden, hidden to output, and hidden to hidden connections. This is an example of a recurrent network that maps an input sequence to an output sequence of the same length. The total loss for a given sequence of x vaues paired

with a sequence of y values would then be just the sum of the losses over all the time steps. For example, if $L^{(t)}$ is the negative log-likehood of $y^{(t)}$ given $x^{(1)}, \dots, x^{(t)}$ then

$$L(\{x^1, \dots, x^\tau\}, \{y^1, \dots, y^\tau\}) \quad 2.15.8$$

$$= \sum_t L^{(t)} \quad 2.15.9$$

$$= \sum_t log p_{model}(y^t | \{x^1, \dots, x^t\}) \quad 2.15.10$$

where $p_{model}(y^t | \{x^1, \dots, x^t\})$ is given by reading the entry for $y^{(t)}$ from the model's output vector $\hat{y}^{(t)}$. Computing the gradient of this loss function with respect to the parameters is an expensive operation. The gradient computation involves performing a forward propagation pass moving left to right through our illustration of the unrolled graph in the figure 2.20, followed by a backward propagation pass moving right to left thought the graph. The runtime is $O(\tau)$. and cannot be reduced by parallelization because the forward propagation graph is inherently sequential: each time step may be computed only after the previous one. States computed in the forward pass must be stored until there are reused during the backward pass, so the memory cost is also $O(\tau)$. The back-progagation algorithm applied to the unrolled graph with $O(\tau)$ cost is called $back-propagation\ through\ time$ (BPTT).

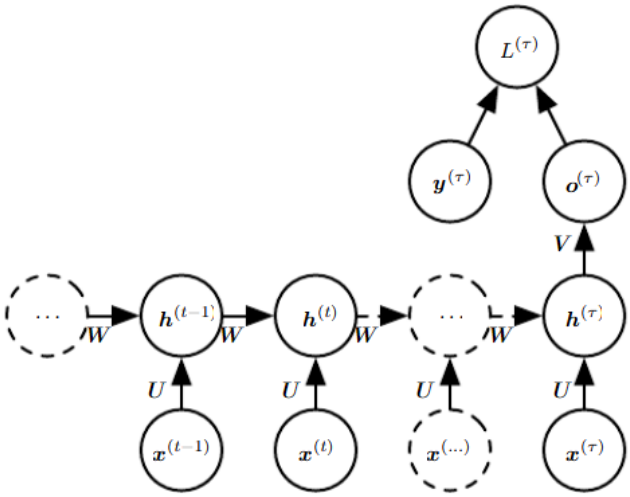


Figure 2.22: Time-unfolded recurrent neural network with a single output at the end of the sequence. [75]

Recurrent neural networks can be used for all sequential forms of data like video frames, text, music etc. The feed-forward networks input some value to the network and returns some value based on that input and the network parameters. A RNN has an internal state that is fed back to the input. It uses the current information on the input and the prediction of the last input. The time steps of a recurrent neural network are often illustrated as in Figure 2.23.

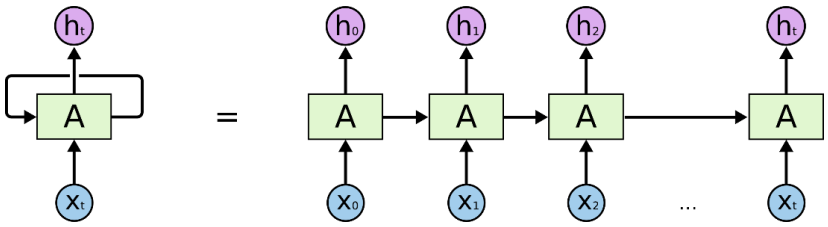


Figure 2.23: Illustration of the time steps of a RNN [40].

RNNs have a problem with vanishing gradient descent. This can happen when the gradient of the activation function becomes very small. When back-propagating through the network the gradient becomes smaller and smaller further back in the network. This makes it hard to model long dependencies. One way of getting around this is to use long short-term memory (LSTM), which is a variant of the RNN. The opposite of the vanishing gradient problem is the exploding gradient problem where the gradient gets to large.

2.15.2 The long short-term memory.

The clever idea of introduction self-loops to produce paths where the gradient can flow for long durations is a core contribution of the initial long-short term memory (LSTM) model [41]. A crucial addition has been to make the weight on this self-loop conditioned on the context, rather than fixed [42]. By making the weight on this self-loop gated (controlled by another hidden unit, the time scale of integration can be changed dynamically. In this case, what is meant is that even for an LSTM with fixed parameters, the time scale of integration can change based on the input sequence, because the time contents are output by the model itself. The LSTM has been found extremely successful in many applications, such as unconstrained handwriting [43] and speech recognition [44]. See figure 2.24.

The LSTM block diagram is illustrated in figure 2.23. The corresponding forward propagation equations are given below, for a shallow recurrent network architecture. Deeper architectures have also been successfully used [45]. Instead of a unit that simply applies an element-wise nonlinearity to the affine transformation of inputs and recurrent units, LSTM recurrent networks have "LSTM cells" that have an internal recurrence (a self-loop), in addition to the outer recurrent network, but also has more parameters and a system of gating units that controls the flow of information. The most important component is the state units $s_i^{(t)}$, which has a linear self-loop like the leaky units described in the previous section. Here, however, the self-loop weight (or the associated time constant) is controlled by a forgot gate unit $f_i^{(t)}$ (for time step t and cell i), which sets this weight to a value between 0 and 1 via sigmoid unit:

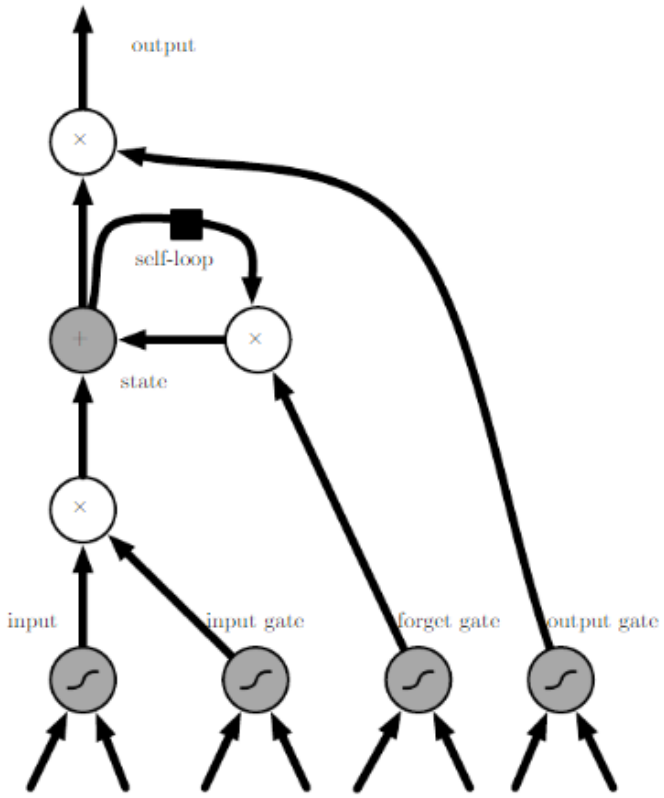


Figure 2.24: Block diagram of the LSTM recurrent network „cell". Cells are connected recurrently to each replacing the usual hidden units of ordinary recurrent networks. An input feature is computed with a regular artificial neuron unit. Its value can be accumulated into the state if the sigmoidal input gate allows it. The state unit has a linear self-loop whose weight is controlled by the forget gate. [75].

$$f_i^{(t)} = \sigma\left(b_i^{(t)} + \sum_j U_{i,j}^{(f)} x_i^{(t)} + \sum_j W_{i,j}^{(f)} h_i^{(t-1)}\right). \qquad 2.15.11$$

Where x^t is the current input vector and $h^{(t)}$ is the current hidden layer vector, containing the outputs of all the LSTM cells, and b^f, U^f, W^f are respectively biases, input weights, and recurrent weights for the forget gates. The LSTM cell internal state is thus updated as follows, but with a conditional self-loop weight $f_i^{(t)}$

$$s_i^{(t)} = f_i^{(t)} s_i^{(t-1)} + g_i^{(t)} \sigma\left(b_i^{(t)} + \sum_j U_{i,j}^{(f)} x_i^{(t)} + \sum_j W_{i,j}^{(f)} h_i^{(t-1)}\right). \qquad 2.15.12$$

where $\boldsymbol{b}, \boldsymbol{U}$ *and* $\boldsymbol{W}$ respectively denote the biases, input weights and recurrent weights into the LSTM cell. The external input gate unit $g_i^{(t)}$ is computed similarly to the forgot gate (with a sigmoid unit to obtain a gatin value between 0 and 1), but with its own parameters:

$$g_i^{(t)} = \sigma\left(b_i^{(g)} + \sum_j U_{i,j}^{(g)} x_i^{(t)} + \sum_j W_{i,j}^{(g)} h_i^{(t-1)}\right). \qquad 2.15.13$$

The output $h_i^{(t)}$ of the LSTM cell can also be shut off, via the output gate $q_i^{(t)}$ wich also use a sigmoid unit for gating.

$$h_i^{(t)} = \tanh\left(s_i^{(t)}\right) q_i^{(t)}. \qquad 2.15.14$$

$$q_i^{(t)} = \sigma\left(b_i^{(0)} + \sum_j U_{i,j}^{(0)} x_i^{(t)} + \sum_j W_{i,j}^{(0)} h_i^{(t-1)}\right). \qquad 2.15.15$$

Which has parameters b^0, U^0, W^0 for its biases, input weights and recurrent weights, respectively. Among the variants, one can choose to use the cell state $s_i^{(t)}$ as an extra input (with its weight) into the three gates of the $i-th$ unit, as shown in the figure 10.16. This would require three additional parameters.

LSTM networks have been shown to learn long-term dependencies more easily than the simple recurrent architectures, first on artificial datasets designed for testing the ability to learn long-term dependencies [26], [41]. The LSTM block consists of three so called gates; the forget gate, the input gate, and the output gate, in addition to the input and output blocks and the memory cell. Figure 2.14 shows an illustration of the block. The vector formulas for the LSTM can be found in LSTM: A Search Space Odyssey [29].

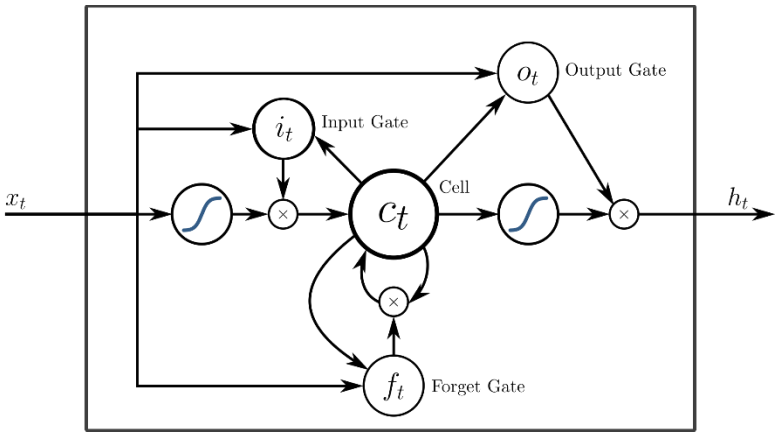


Figure 2.14: Illustration of a LSTM block. [46].

2.15.2.1 Forget Gate

The LSTMs lack of an effective way to reset itself was solved introducing the forget gate to the network [*47*]. The forget gate says how much of information from the input x_t and the last output h_{t-1} to keep. 1 is hold on to everything and 0 is forget everything

2.15.2.2 Input gate.

The input gate says how much of the information that should be stored in the cell state. It prevents the cell from storing unnecessary data.

2.15.2.3 Output Gate

Lastly, the output gate decides how much of the content in the memory cell to expose to the block output.

Part III

Methodology

3. General Structure

3.1 Method

The system consists of two sub systems: the pre-processing and application system shown in Figures 3.1 and 3.2, respectively.

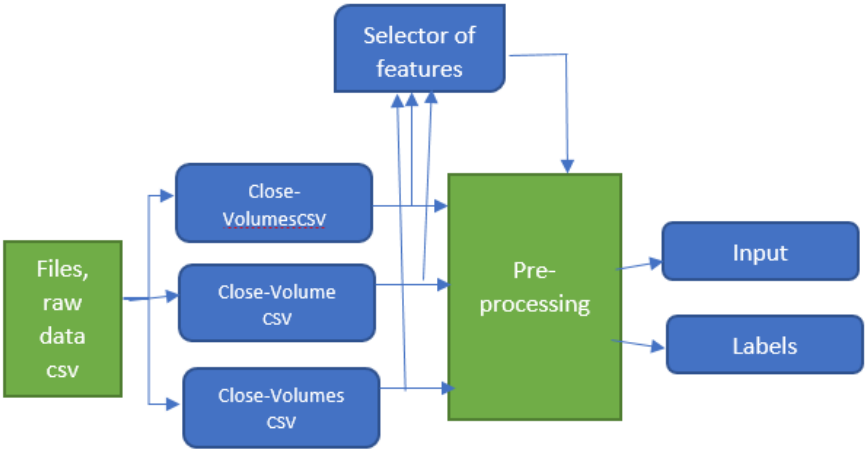


Figure 3.1: Pre-processing system.

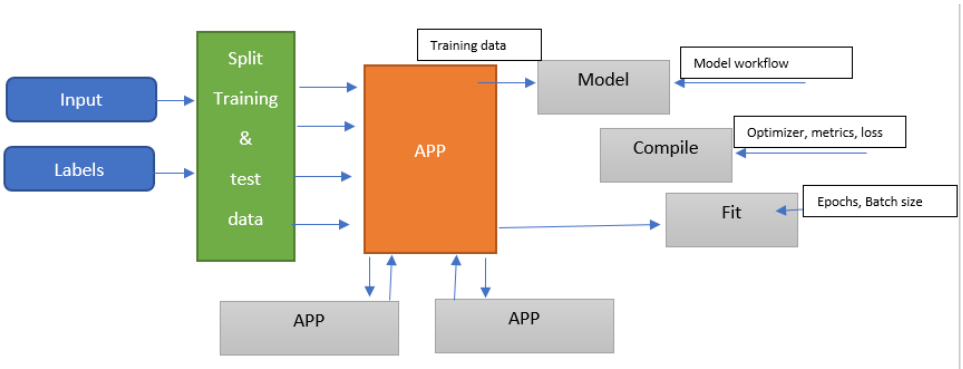


Figure 3.2: Application system.

The pre-processing system in Figure 3.1 inputs the raw data. There is one file for each item, and about 5 items in total in the set. These contain a few parameters each. All the parameters from these items make up the total number of available features. The Data fetcher combines all the features represented by each parameter into big pandas Data frames and saves them as csv files. I.e. each parameter from all the items is stored in their own csv[7] file.

The block diagram of the Data fetcher is shown in Figure 3.3. In cryptocurrencies most of the time the items are Open, Low, High, Close and Volume values within certain date frame.

[7] **CSV** is a simple file format used to store tabular data.

The pickled list of item names is manually put together from a much larger selection of items (open, high, low, close, Volume, Weighedvolume). The application system in Figure 3.2 inputs the numpy arrays of inputs and labels. These are split into a training, a validation and a test set. The model is built, compiled and fitted to the training data. Then the results are visualized and evaluated.

In this thesis, the features taken from the raw file are Close and Volume values from the cryptocurrencies prices: $Litecoin, Bitcoin\ Ethereum\ and\ Bitcoin\ Classic$, the values are in dollars due the importance of this currency in financial transactions.

The pre-processing system does some features extraction, i.e. choosing "close" and "volume values", in each parameter file. All the selected features are scaled prior to saving the data as one set of numpy arrays of inputs and labels. There will thus be one set of inputs/features and outputs/labels files.

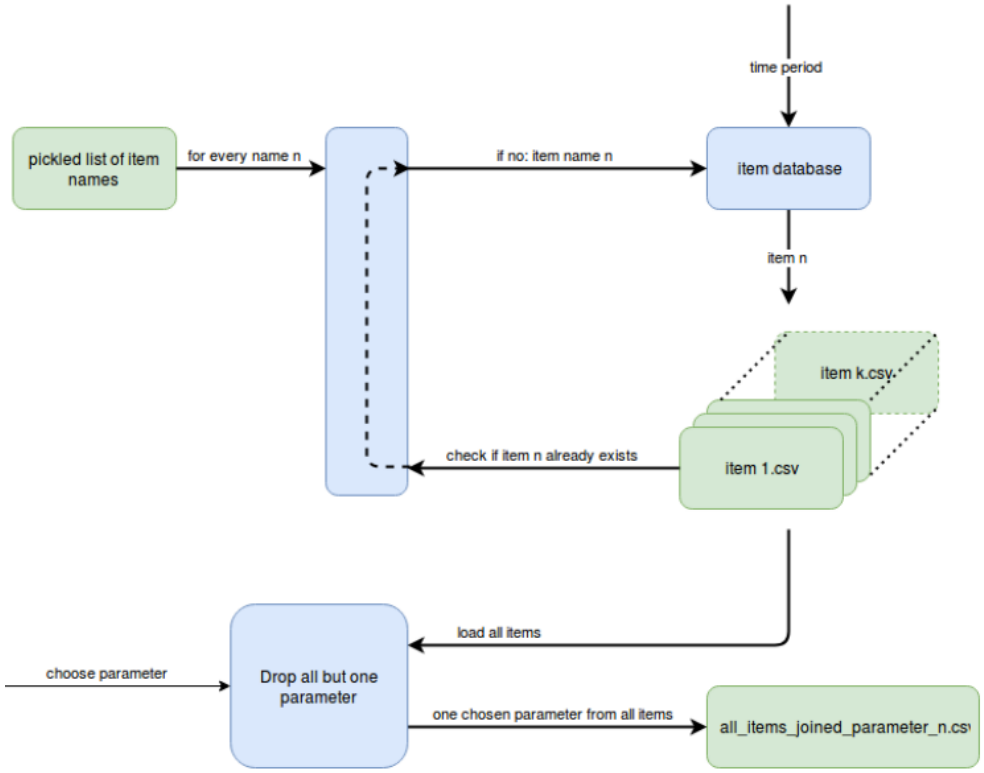


Figure 3.3: Block diagram of the data fetcher.

3.2 Specification

The specification of the pre-processing is easily summarized by three simple bullet points:

- Inputs raw files. One for each feature containing a few parameters each.
- Feature selection and scaling, in this thesis would be Close and Volume.
- Outputs one set of inputs and labels as *numpy* arrays.

The neural networks should be small, and kept shallow minimizing the latency. Their specification is shown in Table 3.1. Only the common specification for all the different topologies are included here. This common specification is used to ease the comparison of performance between the different architectures. Other specifics like number of epochs, batch size, and optimizers might vary for the different topologies and are found mostly by trial and error in the design chapter.

Input shape: look back · number of features
Output shape: binary classification
Number of features: < 50
Look back: < 100
Number of parameters: < 5000
Number of layers in depth: < 5
Number of training examples: > 60000
Number of test examples: > 5000
Output metrics: accuracy
Baseline accuracy: > 50 %

Table 3.1: Common NN specification.

3.3 Baseline criteria

The success criteria are simply to create a model that finds a generalization of the training data that achieves better than 50% accuracy on a binary classification on the test data, i.e. predicting if the next value of a given feature is rising or falling. This must be showed on multiple runs, using different features as the classification feature.

4. Pre-processing data and tools

This chapter will briefly describe the software framework chosen for the neural network implementation, and some of the pre-processing done prior to fitting the data to the model.

4.1 Keras - Software Framework

Based on the selection of software frameworks in the literature review, Keras is chosen as the software framework for testing the different neural network architectures for predicting time series data. This software framework is chosen because of its high-level user interface, enabling faster creation of different models compared to using a more low-level framework - yet still with a lot of options for tweaking and modifications. All information not

cited elsewhere about the Keras functional API[8] is taken from the official Keras Documentation [1] Keras uses either Tensorflow or Theano as backend. In this project Tensorflow is used. The graphs are displayed with tensorboard[9].

4.1.1. Model Creation

Building of models is done using the Sequential function. The Sequential model is basically a stack of layers specified by the user. Only the first layer needs to specify the input shape of the data.

When combining multiple models, the Model method is used.

4.1.2 Layers

A selection of the core layers in Keras taken directly from the documentation [1]:

- *Dense: Just your regular densely-connected NN layer.*
- *Activation: Applies an activation function to an output.*
- *Dropout: Applies Dropout to the input.*
- *Flatten: Flattens the input. Does not affect the batch size. E.g. used between the convolutional layer and the classifier.*
- *Input: Input () is used to instantiate a Keras tensor. The input can be specified as an argument in the first layer.*

4.1.3 Compilation

Configuration of the learning process is done with the compile method. The loss function and the optimizer must be specified. All the different optimizers and their arguments can be found in the Keras documentation.

The Adam optimizer, is based on *SGD*. This is designed to work well with sparse gradients and noisy inputs.

```
opt = tf.keras.optimizers.Adam(lr=0.001, decay=1e-6)

model.compile(
    loss='sparse_categorical_crossentropy',
    optimizer=opt,
    metrics=['accuracy']
```

[8] API is a set of functions and procedures allowing the creation of applications that access the features or data of an operating system, application, or other service

[9] Tensorboard is a tool to visualize TensorFlow graph, plot quantitative metrics about the execution of graphs, and show additional data like images that pass through it.

4.1.4 Training

The *fit* method to train the training data. It uses *numpy* arrays for inputs and labels. The number of epochs and the batch size are specified here.

```
history = model.fit(
    train_x, np.array(train_y),
    batch_size=BATCH,
    epochs=EPOCHS,
    validation_data=(validation_x, validation_y),
    callbacks=[tensorboard, checkpoint],)
```

4.1.5 Evaluation.

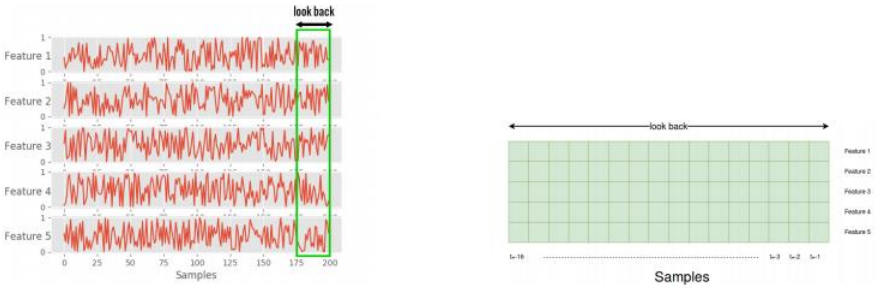
There is an evaluate method that returns the loss value and the metrics values. This provides the model accuracy for a given set of inputs and labels. There is also a method that predicts the output based on a given input. This method is used to predict future values of the time series. The evaluation is done on a separate (unseen) test set, which is not part of the training or validation set.

4.2 Pre- Processing

4.2.1 Shape of Input Frame

The input frame is inspired by the space-time arrangement from Wolfgang Groß's paper [30] in the literature review.

The input frame is made from a moving window over all the features in the dataset. Each input feature is a 1-dimensional array. The length of the array depends on how much history that is included in each input frame, denoted by look back. Combining all the 1-dimensional arrays of features form the input frame for each time stamp. Figure 4.1 illustrate the moving window, which results in a new input frame for each sample. The final shape of the input frame x will be on the format: (samples, look back, number of features).



(a) Time series of 5 features. **(b)** One input frame from 5 features.

Figure 4.1: The relationship between the input features and the input frame.

4.4.2 Scaling.

The different features are not necessarily in scale when acquiring the dataset. Normalization or standardization is done per input frame as shown in Equations 4.1 where $\bar{x}$ is the mean of x and σ is the standard deviation of x.

$$x_{standardization} = \frac{x - \bar{x}}{\sigma}. \tag{4.1}$$

The whole dataset cannot be scaled before use. When predicting future values, these futures vales cannot be a part of the scaling. Here each input frame of LOOK BACK length is scaled.

$$x_{normalization} = \frac{x - \min(x)}{\max(x) - \min(x)}. \tag{4.2}$$

4.4.3 Labeling

Supervised learning uses pre-labeled data to train the models. Labels are the outputs or the answers from the *NN* in response to the input data. Machine learning datasets may already be labeled, but not always. If the data is gathered e.g. by sensors they need to be labeled. In time series regression the labels are basically a time-shifted version of the input feature that is to be predicted. In standard classification with e.g. the MNIST dataset there are 10 output probabilities, one for each handwritten digit. Each number represents the model predictions for the specific classes. This can also be used for classification of time series data. The classifications could e.g. be rising or falling or any number of classifications within specified ranges.

Labeling e.g. if Feature 1 is rising or falling, is done by looking at the previous and current value like shown below:

```
def classify(current_price, future_price):

    if float(future_price) < float(current_price):
        return 0
    else:
        return 1
```

4.4.4 Feature Selection

The prices of cryptocurrencies studied in this thesis, contain many features. All the features can in principle be used, but this would increase computational time a lot, making that approach very inconvenient for this study. Using all the features can also contribute to overfitting. Some of the features have missing data fields, but since there are enough features in this case, the features with missing data are not used.

The dataset is highly random, containing little, if any, structure or repetitive patterns. This makes it hard to achieve any significant increase in classification accuracy. The goal here is to see if the NNs can pick up any structures giving a slightly better performance then 50% on a binary classification.

Multiple features can be used to predict to the rise or fall of one feature. Correlations between features are used to pick out which features to be included in the dataset. Several different setups with different numbers of features will be tested.

4.4.5 Selecting the Number of Samples

The dynamics of the data can change over time. Using too much history can lead to worse results, making the network fit to older data, which has little or nothing in common with newer data. On the other side, the NN needs a fair number of examples to make a good generalization of the problem. In this project, there is a finite number of samples available. Initially, and probably throughout the project, all available samples will be used.

5. Design

This chapter will cover the initial design of different network architectures used during this project.

Finding structures in highly random datasets, like the "mystery set", is a challenging task. All the topologies are therefore first tested using a different dataset that has the same shape of the input frame but contains much more structure to get the feel of the dynamics of each topology. Then, using this experience, the "mystery set" is tested to see if any structures can be extracted.

There is not one specific way of designing a *NN*. It differs a lot depending on the problem. Much is based on trial and error, but there exists a lot of tips and tricks, like the guidelines for building a neural network explained in the article from InfoWorld [49].

The designs used in this project are kept simple and are to a considerable extent based on trial and error. A small portion of the testing is included in this chapter. Some rules of thumb were applied; one obviously needs many input examples for each classification, and there should not be too many parameters compared to the number of samples. Using too large networks would almost certainly result in overfitting.

The dataset is split into training, validation, and a separate test set. It is a finite number of samples in the data set. The validation set is kept relatively small, using most of the data for training. 10% of the data is set aside as a separate test set. The smaller validation set will affect the validation graphs. There will be more variations or noise in the validation set, due to its small size, making it less informative. It is good enough to get some information about the generalization of the data not used in training, and to spot overfitting.

5.1 Neural Network Topologies

Intuitively, *RNNs* are a good start for predicting sequences of data. Replacing the *RNN* with *LSTM* cells would make the network able to learn longer dependencies. As seen in the literature review, CNN are also being used for forecasting. Here dilated convolutions, for example, can be used for enabling the network to exploit longer dependencies. Other papers also reveal good results using a combination of *LSTMs* and *CNN*.

The following topologies - from the literature review - will be tested:

- *MLP*
- *LSTM*
- *CNN*

All the networks will use a two neurons fully connected layer with softmax activation as the classifier. The other activations are ReLU if nothing else is mentioned. To make the comparison fair, each network should have approximately the same number of parameters. This number is kept low and limited to 10000 parameters. Larger networks would also result in overfitting. In addition, none of the networks should be very deep, since this will increase the computational requirements.

5.1.1 Multi-Layer Perceptron (MLP)

The number of parameters in a fully connected layer is basically the weights and the bias of each neuron, where the number of weights depends on the input size.

$$parameters = (input\ size + bias) \cdot n\ neurons. \tag{5.1}$$

For the first layer the input size would be the number of features multiplied with the look back length, i.e. all the samples in the input frame shown in Figure 4.1(b). Visualized with an image analogy; all pixels in the input image are connected to each neuron in the first layer through a unique weight. For the other layers the input size is the number of outputs from the previous layer.

Using wide networks tend to result in overfitting. The idea that substructures are identified in different layers, and that more layers lead to more non-linearity often make networks grow in depth and not in width.

5.1.2 Recurrent Neural Network (RNN)

In general, the number of parameters of a RNN layer with a given number of cells is shown in Equation 5.2. The simple RNN cell has only one hidden state. That state has many weights as inputs. If the RNN cell is the first layer, this equals the number of features in the dataset. In addition, it has a bias and the output state as shown in Figure 5.2.

$$parameters = hidden\ states \cdot cells \cdot ((weights + bias) + outputs) \tag{5.2}$$

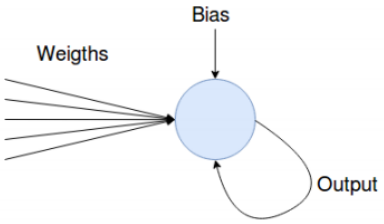


Figure 5.2: Illustration of a hidden state in a RNN layer.

5.1.3 Long Short-Term Memory (LSTM)

The number of parameters in a LSTM layer is like the RNN. It has four hidden states. The three gates and the cell, which makes one LSTM unit, use four times as many parameters compared to a RNN as shown in Equation 5.2.

$$parameters = hidden\,states \cdot units \cdot \big((weights + bias) + outputs\big). \quad 5.3$$

5.1.4 Convolutional Neural Network (CNN)

In a convolutional layer, the number of parameters is basically the size of the convolutional kernel and the bias, multiplied with the number of kernels like shown in Equation 5.4

$$parameters = (m \cdot n + bias) \cdot n\,kernels. \quad 5.4$$

5.2 Choosing Optimizer

Tensorflow provides a range of optimizers with different configuration parameters, though using most optimizers Keras recommend leaving the parameters at their default values. In this project, Adam optimizer is used.

5.3 Choosing Look Back window.

The model accuracy is dependent on how much history is included in the input frame. Also, the number of parameters, using a MLP model, increases with the size of the input frame, making the model more prone to overfitting. The highest accuracy is achieved using a low look back using MLP [42]. The different graphs in each plot are from runs using a different classification feature. In some cases, the model does not seem to learn anything using too low look back. To avoid this, choosing a look back of about 60 seems appropriate.

6 Implementation

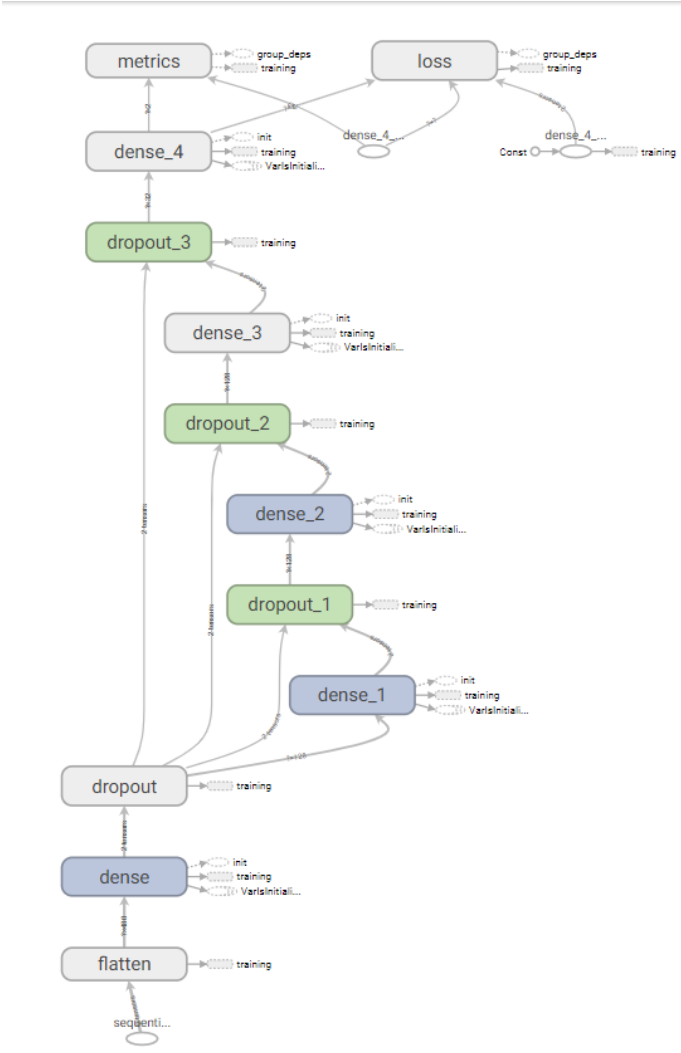
This chapter shows all the different model stacks using the different architectures. All the activation functions except for the classifiers in these model stacks are ReLU. Note that the activation functions can be a part of another layer, like dense 1 (Dense, ReLU), or as a stand-alone layer, activation 1 (ReLU), the estructure of the neural networks was chosen using trial and error method.

6.1 Multi-Layer Perceptron (MLP)

The model stack of the MLP is shown in Table 6.1 and in Graph 6.1

Layer (type)	Output shape
flatten 1 (Flatten)	(None, 480)
dense 1 (Dense, ReLU)	(None, 128)
dropout 1 (Dropout 0.3)	(None, 360)
dense 2 (Dense, ReLU)	(None, 128)
dropout 2 (Dropout 0.2)	(None, 70)
dense 3 (Dense, ReLU)	(None, 2)
activation 1 (softmax)	(None, 2)

Table 6.1: MLP model.



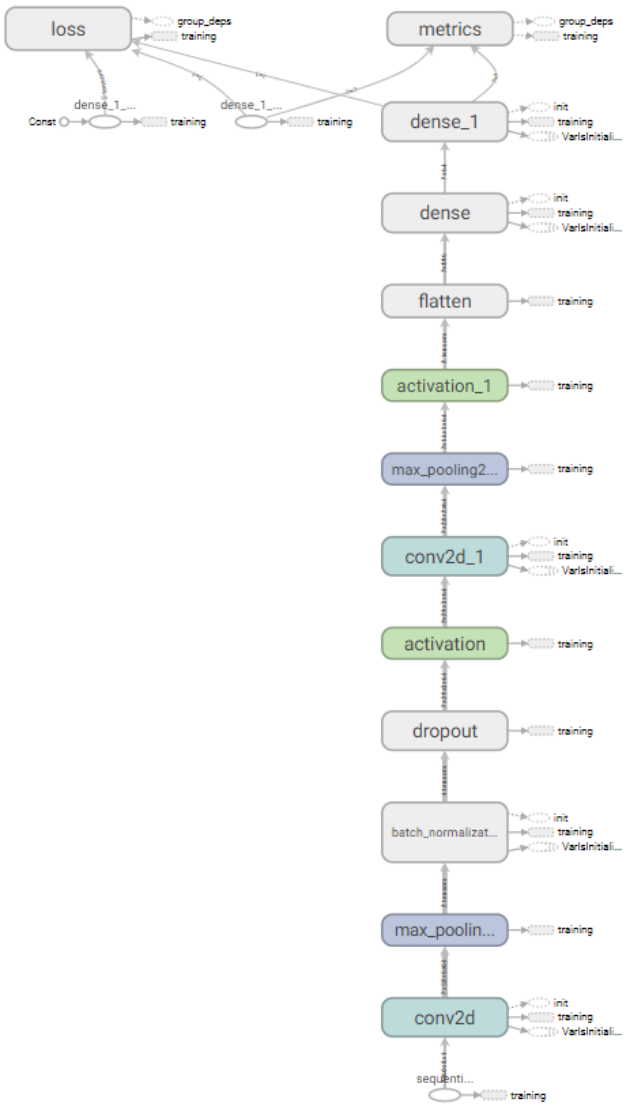
Graph 6.1: MLP model.

6.2 Convolutional Neural Network (CNN)

The model stack of the CNN is shown in Table 6.2 and in Graph 6.2.

Layer (type)
conv 1 (Conv2D)
Kernel Size= [3,3]
batch normalization 1
Pool size= (2,2)
activation 1 (ReLU)
dropout 1 (Dropout 0.3)
conv 2 (Conv2D)
Kernel Size= [3,3]
batch normalization 2
Pool size= (2,2)
activation 2 (ReLU)
dropout 2 (Dropout 0.3)
flatten 1 (Flatten)
dense 1(Dense)
activation 1 (softmax)

Table 6.2: CNN model.



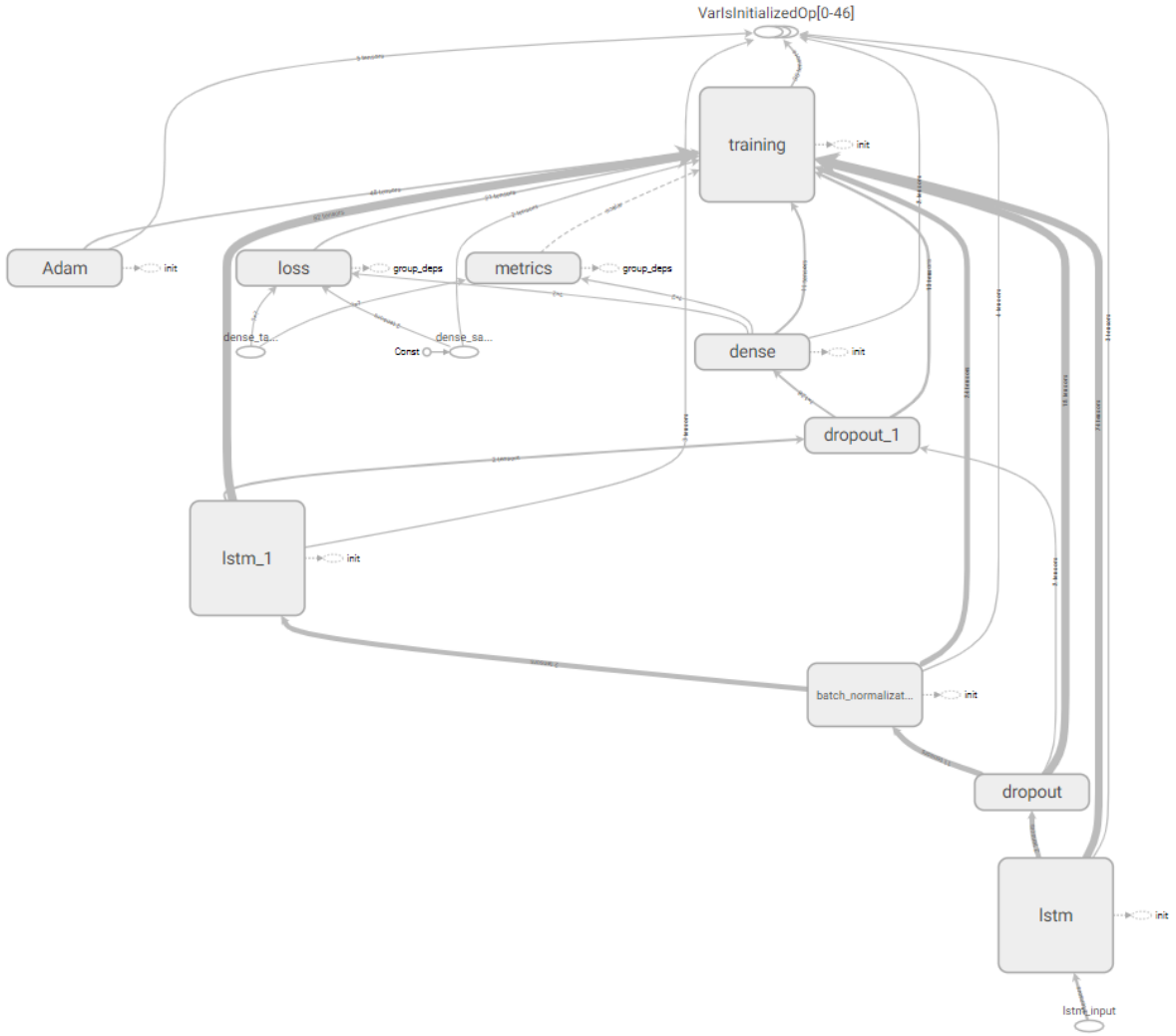
Graph 6.2: CNN model.

6.3 Long Short-Term Memory (LSTM)

The model stack of the LSTM is shown in Table 6.3 and Graph 6.3

Layer (type)
lstm 1 (5 units)
Drop-out 0.2
batch normalization 1
lstm 2 (5 units)
Activation= “relu”
Drop-out (0.1)
dense 1 (Dense)
activation 1 (softmax)

Table 9.3: LSTM model.



Graph 9.3: LSTM model.

Part IV

Results and Discussion

7. Results and Discussion

Many of the models predict a much higher amount of one class. This may give a false impression of performance when the test set contains more of the same class. For example: if there are 60% rising labels in the test set, predicting only that class would give 60% accuracy. To compensate for this, the training data will be splited 50% when the prices goes up and 50% when the prices goes down.

```
for seq, target in sequential_data:
    if target == 0:
        sells.append([seq, target])
    elif target ==1:
        buys.append([seq, target])

random.shuffle(buys)
random.shuffle(sells)

lower = min(len(buys), len(sells))

buys = buys[:lower]
sells = sells[:lower]

sequential_data = buys+sells
random.shuffle(sequential_data)
```

7.1 Multi-Layer Perceptron (MLP)

Results of MLP, due computational and time limitations, the model would run 9 times, each one with 50 epochs. The MLP network does seem to find some patterns in the dataset. In some cases, it seems like the network is learning some patterns as shown below, it is an example of the network giving in average about 53.5% accuracy on the validation set of the best epoch for each run. The network achieves right over 55% on the separate test set (Figure 7.1.2). Over time, the classification accuracy closes in on 50%. Figure 7.1.1 shows accuracy for 9 runs with 50 epochs each, using the MLP, the model was trained for 70 minutes which is extremely fast for a Neural Network.

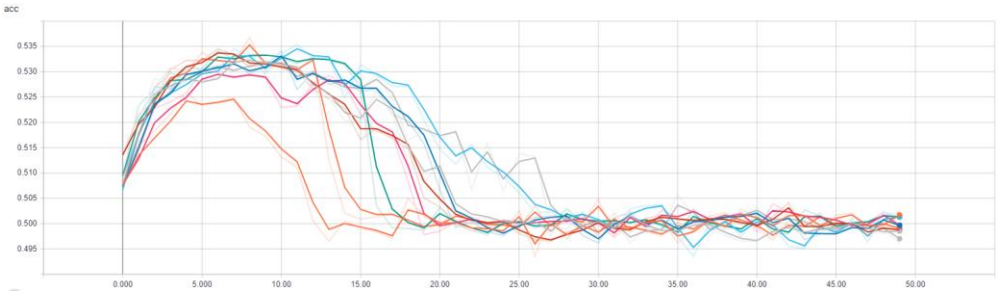


Figure 7.1.1 (a) Accuracy for 9 runs with 50 epochs.

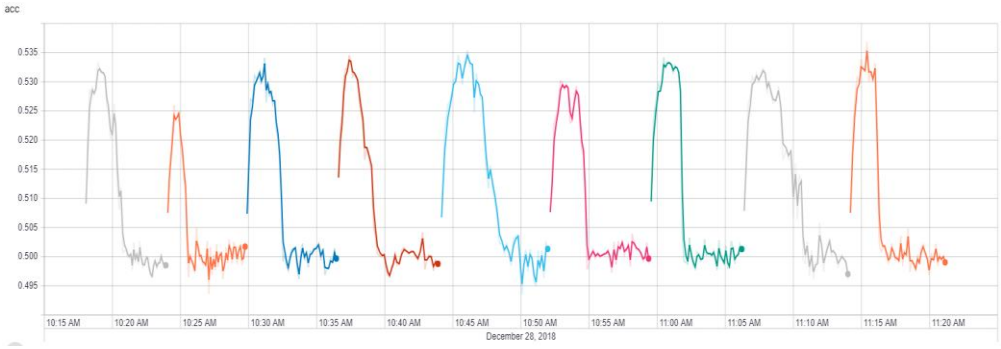


Figure 7.1.1 (b) Accuracy for 9 runs with 50 epochs.

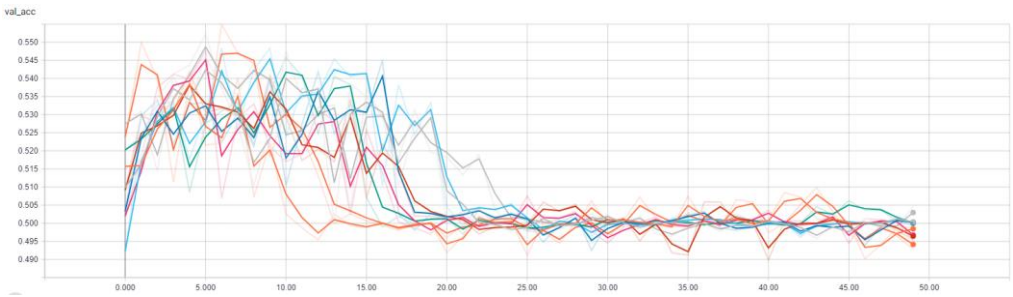


Figure 7.1.2. Accuracy with test data.

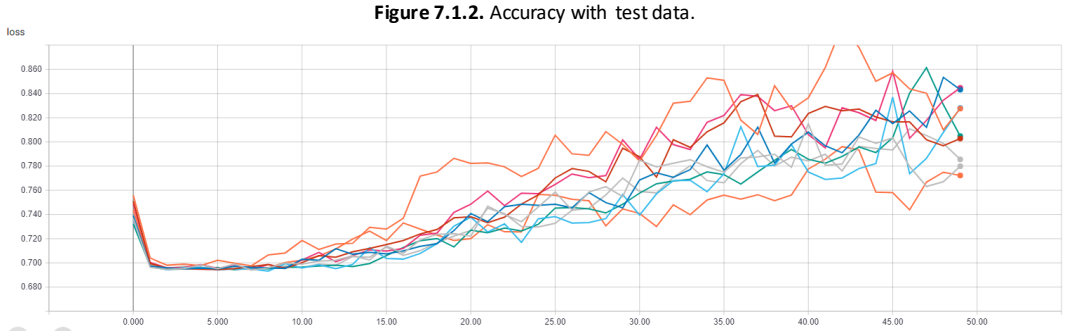


Figure 7.1.4: Loss function.

Epoch	Accuracy	Accuracy	Accuracy	Accuracy	Accuracy	Accuracy	Accuracy	Accuracy	Accuracy
0	50.9%	50.8%	50.7%	51.4%	50.7%	50.8%	50.9%	50.8%	50.8%
1	52.1%	51.5%	51.7%	52.2%	52.1%	51.4%	52.3%	51.8%	52.1%
2	52.9%	51.8%	52.7%	52.4%	52.6%	52.3%	52.7%	52.6%	52.7%
3	52.8%	52.2%	52.7%	53.0%	52.7%	52.4%	53.0%	52.7%	53.1%
4	52.9%	52.6%	53.1%	53.2%	52.8%	52.6%	52.9%	53.1%	53.0%
5	52.8%	52.3%	53.0%	53.2%	53.0%	53.0%	53.1%	53.0%	53.4%
6	52.9%	52.4%	53.1%	53.5%	53.0%	53.0%	53.4%	53.1%	53.2%
7	53.3%	52.5%	53.2%	53.3%	53.5%	52.9%	53.2%	53.0%	53.2%
8	53.2%	51.9%	53.0%	53.1%	53.3%	53.0%	53.3%	53.1%	53.7%
9	53.1%	51.7%	53.1%	53.1%	53.0%	52.9%	53.3%	53.2%	53.0%
10	53.1%	51.3%	53.4%	53.1%	53.4%	52.3%	53.3%	53.1%	53.2%
11	53.1%	51.1%	52.6%	53.0%	53.5%	52.3%	53.2%	52.8%	53.0%

12	52.6%	50.1%	53.0%	52.7%	53.3%	52.8%	53.3%	53.0%	53.3%
13	52.5%	49.7%	52.7%	52.5%	53.3%	52.9%	53.2%	52.8%	51.2%
14	52.0%	50.1%	52.8%	52.3%	52.5%	52.7%	53.1%	52.6%	50.2%
15	52.0%	49.9%	52.6%	51.7%	53.1%	52.2%	52.7%	52.7%	50.1%
16	52.6%	49.8%	52.7%	51.9%	52.9%	51.8%	50.4%	52.9%	50.1%
17	52.2%	49.7%	52.2%	51.7%	52.7%	51.7%	49.9%	52.5%	50.2%
18	51.2%	50.5%	52.0%	51.5%	52.7%	50.8%	49.9%	51.7%	50.0%
19	50.8%	50.1%	51.6%	50.5%	52.0%	49.8%	49.9%	51.8%	49.9%
20	51.2%	49.9%	50.7%	50.3%	51.5%	49.9%	50.3%	51.7%	50.0%
21	50.1%	50.0%	49.9%	50.1%	51.2%	50.0%	50.0%	51.8%	50.2%
22	50.1%	49.9%	50.0%	50.0%	51.6%	50.1%	49.9%	50.7%	50.1%
23	50.1%	49.9%	49.8%	50.0%	51.1%	50.0%	49.8%	51.6%	49.9%
24	50.0%	49.8%	49.8%	50.0%	50.9%	50.1%	50.1%	50.6%	50.0%
25	50.0%	50.3%	50.0%	49.8%	50.6%	50.0%	50.0%	51.4%	49.9%
26	50.1%	49.4%	50.1%	49.7%	50.2%	50.0%	49.9%	51.3%	50.4%
27	49.8%	50.2%	50.1%	49.7%	50.2%	50.1%	50.0%	49.9%	49.8%
28	50.2%	49.7%	50.2%	49.8%	50.1%	50.1%	50.3%	49.9%	50.1%
29	49.9%	50.0%	49.8%	49.9%	50.2%	50.1%	50.0%	50.3%	50.0%
30	50.1%	49.8%	49.6%	50.1%	50.0%	50.0%	50.0%	49.7%	50.5%
31	49.9%	50.2%	50.1%	49.9%	49.8%	49.7%	50.1%	50.1%	49.7%
32	50.3%	49.8%	50.2%	49.9%	50.3%	50.3%	50.0%	49.6%	49.9%
33	49.9%	50.1%	49.9%	50.1%	50.4%	49.9%	50.1%	49.9%	49.8%
34	49.8%	50.1%	50.1%	50.2%	50.4%	50.3%	49.9%	49.9%	49.8%
35	49.9%	49.6%	49.8%	50.1%	49.7%	50.1%	49.8%	50.6%	50.0%
36	50.0%	49.9%	50.1%	50.0%	49.4%	50.3%	50.3%	50.2%	49.9%
37	50.0%	50.2%	50.1%	50.1%	50.0%	50.0%	50.0%	49.7%	50.2%
38	49.8%	49.9%	50.2%	50.1%	50.1%	50.1%	49.9%	50.1%	50.2%
39	49.7%	50.3%	50.1%	50.0%	49.7%	50.2%	49.9%	50.2%	50.0%
40	49.6%	50.1%	50.2%	49.9%	50.1%	49.8%	50.2%	50.3%	49.9%
41	49.9%	49.9%	49.9%	50.1%	49.9%	50.4%	49.8%	50.0%	49.7%
42	50.0%	50.2%	50.1%	50.4%	49.6%	50.2%	49.8%	50.1%	50.1%
43	49.8%	50.2%	49.7%	49.8%	49.5%	50.1%	50.3%	50.0%	49.9%
44	49.9%	49.9%	49.8%	49.9%	50.1%	50.1%	50.2%	49.8%	50.2%
45	49.9%	50.2%	49.8%	50.0%	49.8%	50.0%	49.9%	50.0%	49.8%
46	50.1%	50.2%	50.0%	50.0%	50.1%	49.9%	50.0%	50.0%	50.0%
47	49.9%	49.9%	49.9%	49.8%	49.6%	50.0%	50.0%	49.9%	49.9%
48	49.8%	50.0%	50.1%	49.9%	50.2%	50.2%	50.2%	49.9%	50.0%
49	49.9%	50.3%	49.9%	49.9%	50.2%	49.9%	50.1%	49.6%	49.9%

Table 7.1.1: Model Accuracy.

Figure 7.4 shows how loss function increases after the 10^{th} epoch, which it is in agreement to the accuracy metric as it decreases in average with each epoch after the 10^{th} one. The average performance comparing to adjusted random is 3% better.

Name	Smoothed	Value	Step	Time	Relative
60-SEQ-3-Model-MLP-Time-28122018-run1	0.5314	0.5314	10.00	Fri Dec 28, 10:19:20	1m 16s
60-SEQ-3-Model-MLP-Time-28122018-run2	0.5136	0.5132	10.00	Fri Dec 28, 10:25:15	1m 11s
60-SEQ-3-Model-MLP-Time-28122018-run3	0.5338	0.5341	10.00	Fri Dec 28, 10:31:10	1m 17s
60-SEQ-3-Model-MLP-Time-28122018-run4	0.5308	0.5307	10.00	Fri Dec 28, 10:37:53	1m 16s
60-SEQ-3-Model-MLP-Time-28122018-run5	0.5332	0.5336	10.00	Fri Dec 28, 10:45:52	1m 42s
60-SEQ-3-Model-MLP-Time-28122018-run6	0.5235	0.5230	10.00	Fri Dec 28, 10:53:36	1m 27s
60-SEQ-3-Model-MLP-Time-28122018-run7	0.5328	0.5328	10.00	Fri Dec 28, 11:01:02	1m 28s
60-SEQ-3-Model-MLP-Time-28122018-run8	0.5314	0.5313	10.00	Fri Dec 28, 11:07:55	1m 32s
60-SEQ-3-Model-MLP-Time-28122018-run9	0.5317	0.5318	10.00	Fri Dec 28, 11:15:41	1m 30s

Figure 7.1.5: Table of results.

7.2 Convolutional Neural Network (CNN)

Due the complexity of CNN models and computational limitations, the example below consists of 8 runs with 10 epochs each; training the model took around 1920 minutes (32 hours). The average accuracy of the best epoch for each run is 0.001 % better than the adjusted baseline (Table 7.2.1).

The CNN does not seem to find any patters in the structure data. Besides, against any possibilities, this model does not show any advantage over random classification. The reason behind this could be because of chosen attributes for this neural network, for instants, the optimizer, number of epochs, kernel size, etc. Unfortunately, due to the time consuming for training the models for this project (almost one day and a half) in this thesis does not show other results with different properties of the model.

Accuracy in validation and in test set behaves in the same way, meaning that some properties and variables should be changed in future works for this model if some improvements are required.

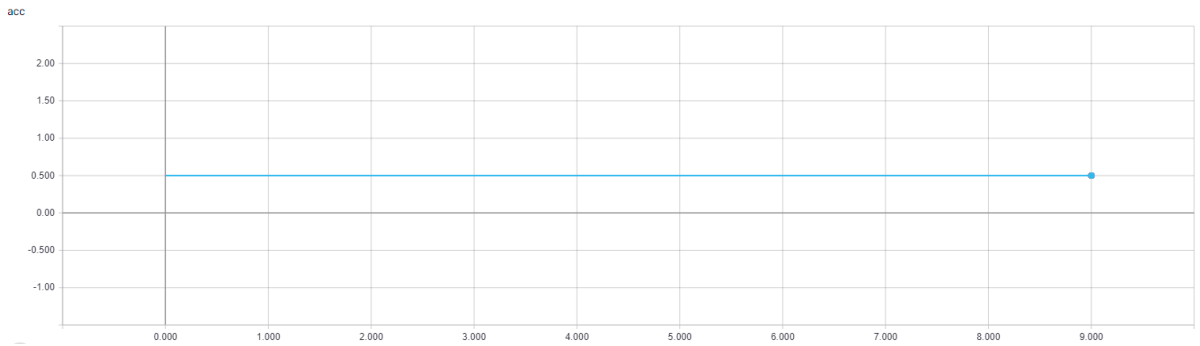


Figure 7.2.1 (b) Accuracy for 8 runs with 10 epochs.

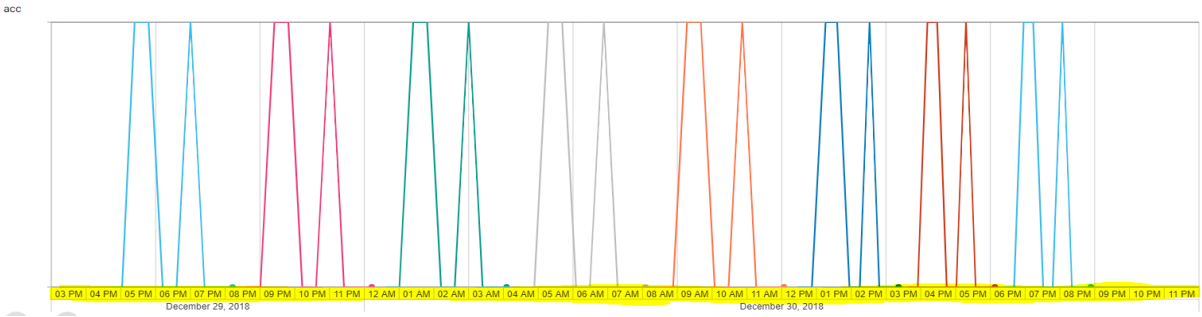


Figure 7.2.1 (b) Accuracy for 8 runs with 10 epochs.

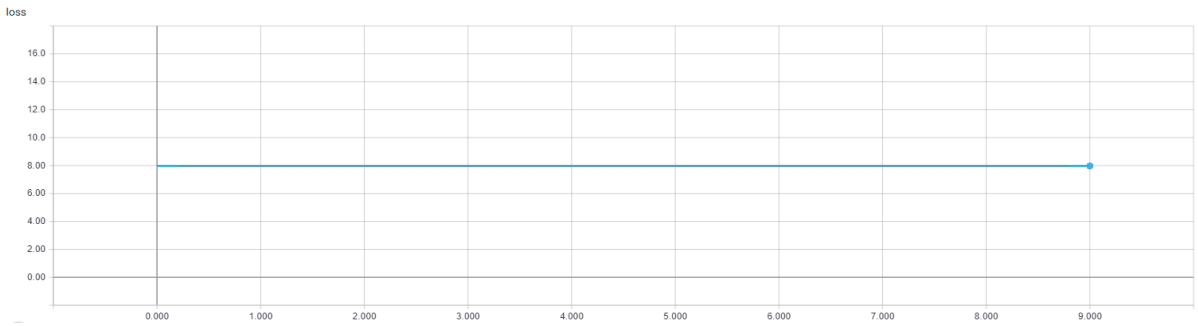


Figure 7.2.2 Loss for 8 runs with 10 epochs.

Epoch	Accuracy	Accuracy	Accuracy	Accuracy	Accuracy	Accuracy	Accuracy	Accuracy
0	0.502	0.491	0.509	0.513	0.503	0.492	0.509	0.497
1	0.489	0.501	0.508	0.502	0.51	0.496	0.496	0.506
2	0.51	0.501	0.5	0.493	0.503	0.5	0.494	0.491
3	0.504	0.508	0.496	0.498	0.513	0.495	0.49	0.495
4	0.504	0.511	0.491	0.508	0.506	0.49	0.501	0.508
5	0.51	0.507	0.511	0.498	0.513	0.5	0.509	0.493
6	0.505	0.505	0.513	0.503	0.497	0.503	0.51	0.494
7	0.51	0.492	0.512	0.499	0.506	0.493	0.49	0.506
8	0.504	0.504	0.497	0.506	0.498	0.511	0.496	0.506
9	0.5	0.501	0.503	0.508	0.498	0.5	0.509	0.497

Table 7.2.1: Model Accuracy.

	Name	Smoothed	Value	Step	Time	Relative
	60-SEQ-3-Model-CNN-Time-27122018-run1	0.5000	0.5000	4.000	Sat Dec 29, 18:12:09	1h 29m 37s
	60-SEQ-3-Model-CNN-Time-27122018-run2	0.5000	0.5000	4.000	Sat Dec 29, 22:12:46	1h 36m 1s
	60-SEQ-3-Model-CNN-Time-27122018-run3	0.5000	0.5000	4.000	Sun Dec 30, 02:12:11	1h 35m 9s
	60-SEQ-3-Model-CNN-Time-27122018-run4	0.5000	0.5000	4.000	Sun Dec 30, 06:05:10	1h 35m 48s
	60-SEQ-3-Model-CNN-Time-27122018-run5	0.5000	0.5000	4.000	Sun Dec 30, 10:04:23	1h 35m 36s
	60-SEQ-3-Model-CNN-Time-27122018-run6	0.5000	0.5000	4.000	Sun Dec 30, 13:57:01	1h 28m 57s
	60-SEQ-3-Model-CNN-Time-27122018-run7	0.5000	0.5000	4.000	Sun Dec 30, 16:44:41	1h 6m 25s
	60-SEQ-3-Model-CNN-Time-27122018-run8	0.5000	0.5000	4.000	Sun Dec 30, 19:31:08	1h 6m 28s

Figure 7.2.3: Table of result.

7.3 Long Short-Term Memory (LSTM)

Due to the complexity of LSTM models and computational limitations, the example below consists of 9 runs with 10 epochs each, training the model took around 840 minutes. The average accuracy of the best epoch for each run is 6 % better than the adjusted baseline (Table 7.3.1). EUREKA!

It can be observed that the model seems to find some patterns since the accuracy in validation data for most of the cases increases with each epoch, Figure 7.3.1. However, evaluating the accuracy in separate test set shows a diminution, Figure 7.3.2, this means that the model overfitted after some runs, and ideally the model should run with more epochs since the values seem to increase smoothly.

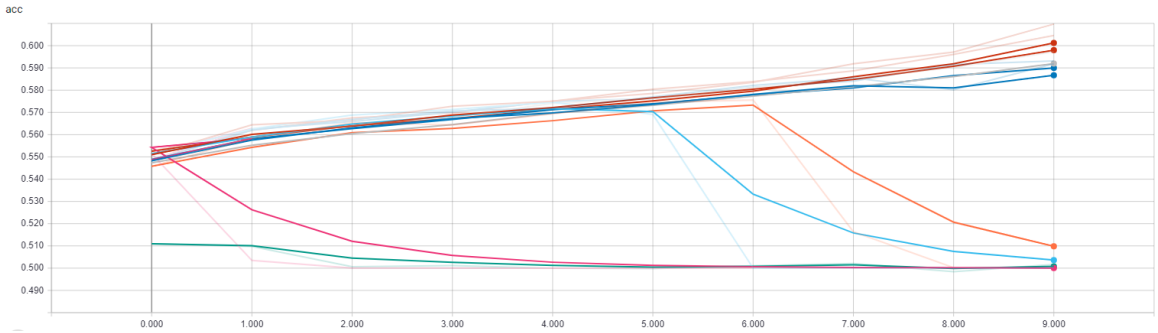


Figure 7.3.1 (a): Accuracy Validation Set (9 runs, 10 epochs).

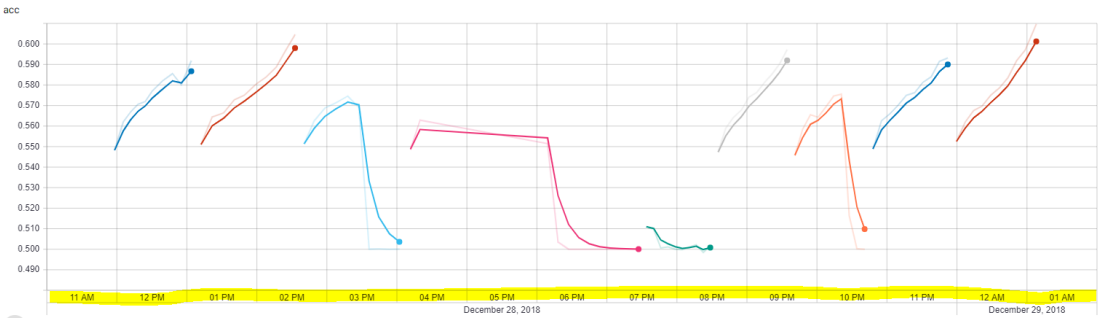


Figure 7.3.1 (b): Accuracy Validation Set (9 runs, 10 epochs, 14 hours training).

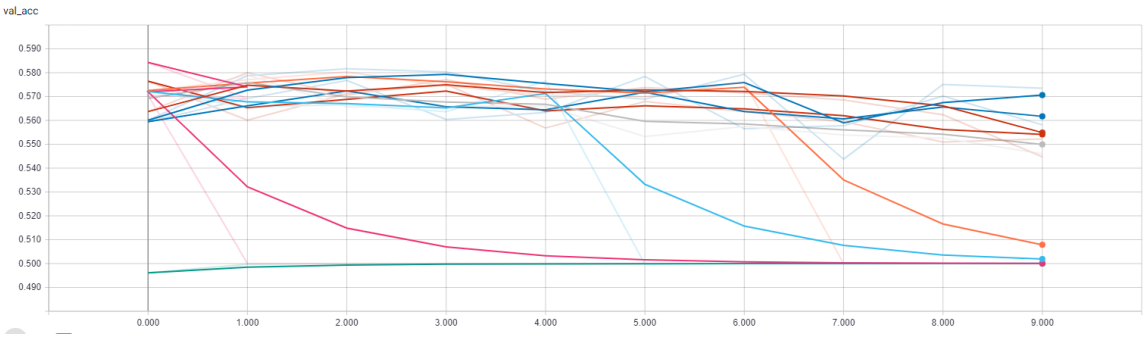


Figure 7.3.2: Accuracy test Set (9 runs, 10 epochs).

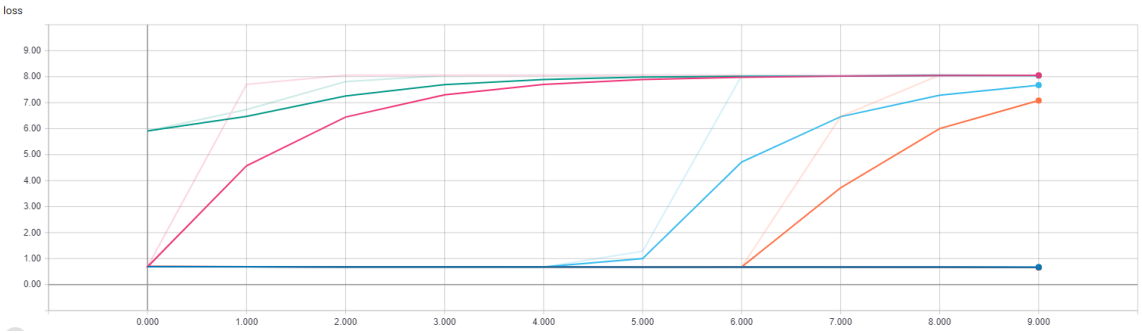


Figure 7.3.3: Loss function validation Set (9 runs, 10 epochs).

Epoch	Accuracy	Accuracy	Accuracy	Accuracy	Accuracy	Accuracy	Accuracy	Accuracy	Accuracy
0	54.8%	54.9%	55.3%	54.6%	54.7%	51.1%	54.9%	55.1%	55.1%
1	56.2%	56.3%	56.2%	55.8%	55.9%	51.0%	56.3%	56.3%	56.4%
2	56.7%	56.6%	56.8%	56.6%	56.4%	50.1%	55.1%	56.9%	56.6%
3	57.1%	57.0%	57.0%	56.4%	56.8%	50.1%	50.3%	57.1%	57.3%
4	57.2%	57.5%	57.5%	56.9%	57.4%	50.0%	50.0%	57.5%	57.5%
5	57.7%	57.6%	57.9%	57.5%	57.6%	50.0%	50.0%	56.9%	58.0%
6	58.2%	58.1%	58.4%	57.6%	58.1%	50.1%	50.0%	50.0%	58.4%
7	58.6%	58.4%	59.2%	51.6%	58.5%	50.2%	50.0%	50.0%	58.9%
8	58.0%	59.1%	59.7%	50.0%	59.0%	49.8%	50.0%	50.0%	59.6%
9	59.2%	59.3%	61.0%	50.0%	59.7%	50.2%	50.0%	50.0%	60.5%

Table 7.3.2: Accuracy test Set (9 runs, 10 epochs).

Name	Smoothed	Value	Step	Time	Relative
60-SEQ-3-Model-LSTM-Time-27122018-run1	0.5483	0.5483	0.000	Fri Dec 28, 11:58:03	0s
60-SEQ-3-Model-LSTM-Time-27122018-run2	0.5510	0.5510	0.000	Fri Dec 28, 13:12:10	0s
60-SEQ-3-Model-LSTM-Time-27122018-run3	0.5514	0.5514	0.000	Fri Dec 28, 14:40:30	0s
60-SEQ-3-Model-LSTM-Time-27122018-run4	0.5487	0.5487	0.000	Fri Dec 28, 16:11:37	0s
60-SEQ-3-Model-LSTM-Time-27122018-run5	0.5110	0.5110	0.000	Fri Dec 28, 19:33:49	0s
60-SEQ-3-Model-LSTM-Time-27122018-run6	0.5473	0.5473	0.000	Fri Dec 28, 20:35:20	0s
60-SEQ-3-Model-LSTM-Time-27122018-run7	0.5458	0.5458	0.000	Fri Dec 28, 21:41:07	0s
60-SEQ-3-Model-LSTM-Time-27122018-run8	0.5488	0.5488	0.000	Fri Dec 28, 22:48:01	0s
60-SEQ-3-Model-LSTM-Time-27122018-run9	0.5526	0.5526	0.000	Fri Dec 28, 23:59:37	0s

Figure 7.3.4: Table of results.

8. Results and Discussion

The success criteria from the functional description of achieving over 50% accuracy was accomplished using the LSTM and MLP. Using larger networks result in overfitting without having more training samples.

The main goal of the project was achieved using the LSTM and MLP. The results from this thesis indicate that there are structures and pattern to be found in the prices of the cryptocurrencies, moreover the results indicate that the accuracy for this model could be better with more computational power, due to these limitations, the model gets better results than random choice, but is not the optimal.

The main limitation factor for this thesis was the lack of powerful hardware, which is able to process more complex operations within less time. These models were limited to 8 GB RAM, resulting in less accuracy, whether it was in validation or in test set data.

It is very likely that the model performs better with further optimization models. Besides, that there is most probably a lot to improve in the pre-processing and feature selection phase, which is much based on trial and error during this project. There exist other methods which might give a better starting point, resulting in higher accuracy. From a better starting point, the other topologies, which did not work in this case, might also do better, although this thesis' results indicate that LSTM probably are the best choice for forecasting cryptocurrencies prices, it gives the indented proof of concept and starting point for further research and development.

The $softmax$ classifier outputs the probabilities, predicted by the model, for each class. Using two classes, this results in two probabilities that represent how strongly the model predicts that the next value of the classification feature is greater or smaller than the current value.

In addition to binary classification, one could increase the resolution of the output by adding more classes. E.g. using a three-way classifier for rising, falling, and staying within some limit. Regression can also be an option for future work. This project assesses one-step forecasts for the prices the four of the most capitalized cryptocurrencies. It is possible to also do multi-step forecasts, most probably, with a probable decrease in accuracy because of the complexity of the algorithm.

9. Conclusion and Future work

9.1 Overview

The main goal of this thesis was to investigate the application of three different types of artificial neural networks, used in one-step prediction of the prices of four of the most capitalized cryptocurrencies, specifically Litecoin [57], Bitcoin [55], Ethereum [56] and Bitcoin Cash [58]. The neural networks used in this project have been built and tested in Python with the most common and powerful libraries for deep learning, Tensorflow [2] and Keras [1].

The literature review focuses in 3 different types of neural networks, due their popularity within Data scientists and their wide research about them available in Internet, $MLPs$, $LSTM$ and $CNNs$. Models were built based on literature review and trial and error method.

The goal of this research was to see if it was possible to create functional deep neural networks that could generate a better result than 50%. Fortunately, the goals exceeded the expectations by creating two working models, which generates a good solution but not enough to take the risk to invest one savings life.

The results have been a bit better than the baseline of 50 %, most probably future researches and projects will improve this thesis performance, maybe with different pre-processing, construction of NN and with more computational power.

Given the wildly unpredictable nature of the price of the Cryptocurrencies, what was aimed for was creating an evenly remotely accurate model. Although the results aren not perfectly accurate, they do have some predictive capability. The results show large statistical significant improvements in point forecasting of time series when using combinations of univariate models. Hopefully, this analysis opens various research agenda in predicting prices of more cryptocurrencies on real time. For example, flexible multivariate combination schemes, which would allow for different weights across series to improve the model accuracy. Moreover, new predictors based on crypto–market sentiments might be considered to investigate cryptocurrencies sentiments and could result in forecast gains across all series.

Further works might mention portfolio optimization, risk management, or the usage of more complex structures and neural networks, for example, reinforcement neural networks. Also, this could help to create a new strategy for trading cryptocurrencies online, for instant**,** a bot trading 24 hour using deep learning.

9.2 Future Work

Based on the results, the future work could be:

- To try a better strategy, probably using Reinforcement Neural Networks
- To use more features for the model, time, day, moving average.
- The usage of Natural Language Process for sentiment analysis in social media.
- To explore other ways of Pre-processing and feature selection methods.
- To use other supervised machine learning algorithms.
- Multi-step forecasting.
- Regression.
- To create a bot to trade in live markets with real money and gain some profit.

Anexos.

Code: Pre-processing data

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, LSTM, CuDNNLSTM,
BatchNormalization, Flatten
from tensorflow.keras.callbacks import TensorBoard, ModelCheckpoint
import time
gpu_options = tf.GPUOptions(per_process_gpu_memory_fraction=0.15)
sess = tf.Session(config=tf.ConfigProto(gpu_options=gpu_options))

directory = 'C:/Users/HUPerezA/Desktop/Tesis/crypto_data/crypto_data/'

# df = pd.read_csv(directory, names=["time", "low", 'high', 'open', 'close',
'volume'])

LOOK_BACK_WINDOW = 60
FORWARD_WINDOW= 3
PRICE_TO_PREDICT = 'LTC-USD'
EPOCHS = 10
BATCH = 64
NAME = F"{LOOK_BACK_WINDOW}-SEQ-{FUTURE_PERIOD_PREDICT}-PRED-{int(time.time())}"



def classify(current_price, future_price):
    if float(future_price) < float(current_price):
        return 0
    else:
        return 1

def preprocess_df_price_vol(df_price_vol):
    df_price_vol = df_price_vol.drop('future',1)
    for col in df_price_vol.columns:
        if col != 'target':
            df_price_vol[col] = df_price_vol[col].pct_change()
            df_price_vol.dropna(inplace=True)
            # Scaling puede cambiar
            df_price_vol[col] = preprocessing.scale(df_price_vol[col].values)
    df_price_vol.dropna(inplace=True)

    sequential_data = []
    prev_days = deque(maxlen=LOOK_BACK_WINDOW)

    for i in df_price_vol.values:

        prev_days.append([n for n in i[:-1]])
        if len(prev_days) == LOOK_BACK_WINDOW:
            sequential_data.append([np.array(prev_days), i[-1]])
            # break

    random.shuffle(sequential_data)

    buys = []
    sells = []

    for seq, target in sequential_data:
        if target == 0:
            sells.append([seq, target])
        elif target ==1:
            buys.append([seq, target])

    random.shuffle(buys)
    random.shuffle(sells)
```

```
    lower = min(len(buys), len(sells))

    buys = buys[:lower]
    sells = sells[:lower]

    sequential_data = buys+sells
    random.shuffle(sequential_data)

    X = []
    y = []

    for seq, target in sequential_data:
        X.append(seq)
        y.append(target)

    return np.array(X), y



main_df_price_vol = pd.DataFrame()

# Could scan the directory
ratios = ['BTC-USD', 'LTC-USD', 'ETH-USD', 'BCH-USD']

for ratio in ratios:
    dataset = f"{directory}/{ratio}.csv"
    df_price_vol = pd.read_csv(dataset, names=["time", "low", 'high', 'open',
'close', 'volume'])
    df_price_vol.rename(columns={"close": f"{ratio}_close", "volume":
f"{ratio}_volume"}, inplace=True)

    df_price_vol.set_index("time", inplace=True)
    df_price_vol = df_price_vol[[f"{ratio}_close", f"{ratio}_volume"]]

    if len(main_df_price_vol) ==0:
        main_df_price_vol = df_price_vol
    else:
        main_df_price_vol = main_df_price_vol.join(df_price_vol)


main_df_price_vol['future'] =
main_df_price_vol[f"{PRICE_TO_PREDICT}_close"].shift(-FUTURE_PERIOD_PREDICT)
main_df_price_vol['target'] = list(map(classify,
main_df_price_vol[f"{PRICE_TO_PREDICT}_close"], main_df_price_vol['future']))

times = sorted(main_df_price_vol.index.values)
last_p5ct = times[-int(0.05*len(times))]


validation_main_df_price_vol = main_df_price_vol[(main_df_price_vol.index >=
last_p5ct)]
main_df_price_vol = main_df_price_vol[(main_df_price_vol.index < last_p5ct)]

train_x, train_y = preprocess_df_price_vol(main_df_price_vol)
validation_x, validation_y = preprocess_df_price_vol(validation_main_df_price_vol)
```

Code: Multilayer Perceptron

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, LSTM, CuDNNLSTM,
BatchNormalization, Flatten
from tensorflow.keras.callbacks import TensorBoard, ModelCheckpoint
from PreprocessingData import data_to_train
gpu_options = tf.GPUOptions(per_process_gpu_memory_fraction=0.50)
sess = tf.Session(config=tf.ConfigProto(gpu_options=gpu_options))
import time

LOOK_BACK_WINDOW = 60
FORWARD_WINDOW= 3
PRICE_TO_PREDICT = 'LTC-USD'
EPOCHS = 50
BATCH = 32
MODEL = "MLP"
TIME = "28122018"



train_x, train_y, validation_x, validation_y   = data_to_train()


for num_run in range(1, 10):
    NAME = F"{LOOK_BACK_WINDOW}-SEQ-{FUTURE_PERIOD_PREDICT}-Model-{MODEL}-Time-
{TIME}-run{num_run}"
    print(train_x.shape[1:])

    model = Sequential()

    model.add(Flatten())
    model.add(Dense(128,  input_shape=(train_x.shape[1:])))
    model.add(Dropout(0.2))


    model.add(Dense(128))
    model.add(Dropout(0.1))


    model.add(Dense(128))
    model.add(Dropout(0.2))

    model.add(Dense(32, activation='relu'))
    model.add(Dropout(0.2))

    model.add(Dense(2, activation='softmax'))


    opt = tf.keras.optimizers.Adam(lr=0.001, decay=1e-6)


    model.compile(
        loss='sparse_categorical_crossentropy',
        optimizer=opt,
        metrics=['accuracy']
    )

    tensorboard = TensorBoard(log_dir=f'logs/{NAME}')

    filepath = "RNN_Final-{epoch:02d}-{val_acc:.3f}"
    checkpoint = ModelCheckpoint("models/{}.model".format(filepath,
monitor='val_acc', verbose=1, save_best_only=True, mode='max'))
```

```
history = model.fit(
    train_x, np.array(train_y),
    batch_size=BATCH,
    epochs=EPOCHS,
    validation_data=(validation_x, validation_y),
    callbacks=[tensorboard, checkpoint],
)


score = model.evaluate(validation_x, validation_y, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])

model.save("models/{}".format(NAME))
```

Code: Convolutional Neural Network

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import  BatchNormalization, Flatten, Dense, Dropout,
Activation, Flatten, Conv2D, MaxPooling2D, Conv1D
from tensorflow.keras.callbacks import TensorBoard, ModelCheckpoint
from PreprocessingData import data_to_train
gpu_options = tf.GPUOptions(per_process_gpu_memory_fraction=0.15)
sess = tf.Session(config=tf.ConfigProto(gpu_options=gpu_options))
from tensorflow.keras import backend as k

import time

LOOK_BACK_WINDOW = 60
FORWARD_WINDOW= 3
PRICE_TO_PREDICT = 'LTC-USD'
EPOCHS = 10
BATCH = 64
MODEL = "CNN"
TIME = "27122018"

train_x, train_y, validation_x, validation_y = data_to_train()

train_x = np.array(train_x).reshape(train_x.shape[0], train_x.shape[1],
train_x.shape[2], 1)
validation_x = np.array(validation_x).reshape(validation_x.shape[0],
validation_x.shape[1], validation_x.shape[2], 1)

for num_run in range(1, 10):
    NAME = F"{LOOK_BACK_WINDOW}-SEQ-{FUTURE_PERIOD_PREDICT}-Model-{MODEL}-Time-
{TIME}-run{num_run}"
    model = Sequential()
    model.add(Conv2D(64, kernel_size=[3, 3]))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(BatchNormalization())
    model.add(Dropout(0.3))
    model.add(Activation('relu'))

    model.add(Conv2D(64, kernel_size=[2, 2]))
    model.add(MaxPooling2D(pool_size=(2,2)))
    model.add(Activation('relu'))

    model.add(Flatten())
    model.add(Dense(64, activation='relu'))

    model.add(Dense(1, activation='softmax'))


    opt = tf.keras.optimizers.Adam(lr=0.001, decay=1e-6)

    model.compile(
        loss='binary_crossentropy',
        optimizer=opt,
        metrics=['accuracy']
    )

    tensorboard = TensorBoard(log_dir=f'logs/{NAME}')

    filepath = "CNN_Final-{epoch:02d}-{val_acc:.3f}"
    checkpoint = ModelCheckpoint("models/{}.model".format(filepath,
monitor='val_acc', verbose=1, save_best_only=True, mode='max'))

    history = model.fit(
        train_x, np.array(train_y),
```

```
    batch_size=BATCH,
    epochs=EPOCHS,
    validation_data=(validation_x, np.array(validation_y)),
    callbacks=[tensorboard, checkpoint],
)


score = model.evaluate(validation_x, validation_y, verbose=0)


model.save("models/{}".format(NAME))
```

Code: Long-Short Term Memory

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, LSTM, CuDNNLSTM,
BatchNormalization, Flatten
from tensorflow.keras.callbacks import TensorBoard, ModelCheckpoint
from PreprocessingData import data_to_train
gpu_options = tf.GPUOptions(per_process_gpu_memory_fraction=0.50)
sess = tf.Session(config=tf.ConfigProto(gpu_options=gpu_options))
import time

LOOK_BACK_WINDOW = 60
FORWARD_WINDOW= 3
PRICE_TO_PREDICT = 'LTC-USD'
EPOCHS = 10
BATCH = 64
MODEL = "LSTM"
TIME = "27122018"

train_x, train_y, validation_x, validation_y   = data_to_train()


for num_run in range(1, 10):
    NAME = F"{LOOK_BACK_WINDOW}-SEQ-{FUTURE_PERIOD_PREDICT}-Model-{MODEL}-Time-
{TIME}-run{num_run}"
    model = Sequential()


    print((train_x.shape))
    print((train_x.shape[1:]))
    model = Sequential()
    model.add(LSTM(128, input_shape=(train_x.shape[1:]), return_sequences=True))
    model.add(Dropout(0.2))
    model.add(BatchNormalization())

    model.add(LSTM(128,  activation='relu', return_sequences=False))
    model.add(Dropout(0.1))
    model.add(Dense(2, activation='softmax'))


    opt = tf.keras.optimizers.Adam(lr=0.001, decay=1e-6)


    model.compile(
        loss='sparse_categorical_crossentropy',
        optimizer=opt,
        metrics=['accuracy']
    )

    tensorboard = TensorBoard(log_dir=f'logs/{NAME}')

    filepath = "RNN_Final-{epoch:02d}-{val_acc:.3f}"
    checkpoint = ModelCheckpoint("models/{}.model".format(filepath,
monitor='val_acc', verbose=1, save_best_only=True, mode='max')) # saves only the
best ones

    # Training
history = model.fit(
    train_x, np.array(train_y),
    batch_size=BATCH,
    epochs=EPOCHS,
    validation_data=(validation_x, np.array(validation_y)),
    callbacks=[tensorboard, checkpoint],
)
```

```
# Score model
score = model.evaluate(validation_x, validation_y, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
# Save model
model.save("models/{}".format(NAME))
```

Bibliography

[1] Chollet, F. (2015). Keras. Accessed: 31-12-2018 GitHub: https://github.com/fchollet/keras URL https://keras.io.

[2] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mane,´ D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viegas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., Zheng, X., (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org. Accessed: 31-12-2018. URL https://www.tensorflow.org/

[3] Keras-team, (2018). Keras git repository. Accessed: 31-12-2018. URL https://github.com/keras-team/keras/tree/master/examples

[4] Dodge, S. and Karam, L., (2017). A study and comparison of human and deep learning recognition performance under visual distortions. In *Computer Communication and Networks (ICCCN), 2017 26th International Conference on* (pp. 1-7). IEEE.

[5] Geirhos, R., Janssen, D.H., Schütt, H.H., Rauber, J., Bethge, M. and Wichmann, F.A., (2017). Comparing deep neural networks against humans: object recognition when the signal gets weaker. *arXiv preprint arXiv:1706.06969*.

[6] Gibbs, S., (2017). *Alphazero AI beats champion chess program after teaching itself in four hours*. *Retrieved from Guardian.* Accessed: 31-12-2018. URL https://www. theguardian. com/technology/2017/dec/07/alphazerogoogle-deepmind-ai-beats-champion-program-teaching-itself-to-play-four-hours.

[7] Lacey, G., Taylor, G.W. and Areibi, S., (2016). *Deep learning on fpgas*: Past, present, and future. *arXiv preprint arXiv:1602.04283*.

[8] Russell, S.J. and Norvig, P., (2016). *Artificial intelligence: a modern approach.* Malaysia; Pearson Education Limited.

[9] Patterson, J. and Gibson, A., (2017). *Deep Learning: A Practitioner's Approach.* " O'Reilly Media, Inc.".

[10] Stiles, J. and Jernigan, T.L., (2010). *The basics of brain development*. *Neuropsychology review*, *20*(4), (pp.327-348).

[11] Le, Q.V., 2013, May. *Building high-level features using large scale unsupervised learning. In Acoustics, Speech and Signal Processing (ICASSP),* (2013) IEEE International Conference on (pp. 8595-8598). IEEE.

[12] Hsu, J., (2015). *Biggest Neural Network Ever Pushes AI Deep Learning. IEEE Spectrum: Technology, Engineering, and Science News*.

[13] Michalski, R. S., Carbonell, J. G., & Mitchell, T. M. (2013). *Machine learning: An artificial intelligence approach*. Springer Science & Business Media.

[14] Goodfellow, I., Bengio, Y., Courville, A., & Bengio, Y. (2016). *Deep learning* (Vol. 1). Cambridge: MIT press.

[15] Taigman, Y., Yang, M., Ranzato, M. A., & Wolf, L. (2014). *Deepface: Closing the gap to human-level performance in face verification*. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 1701-1708).

[16] Cauchy, A. (1847). *Méthode générale pour la résolution des systemes d'équations simultanées. Comp. Rend. Sci. Paris*, *25*(1847), (pp. 536-538).

[17] Russell, S. J., & Norvig, P. (2016). *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited.

[18] Harry Fairhead. (2014). *The mcculloch-pitts neuron*. Accessed: 31-12-2018. URL http://www.i-programmer.info/babbages-bag/325- mcculloch-pitts-neural-networks.html

[19] Patterson, J., & Gibson, A. (2017). *Deep Learning: A Practitioner's Approach*. " O'Reilly Media, Inc.".

[20] Gupta, D. (2017). *Fundamentals of deep learning– activation functions and when to use them*? Accessed: 31-12-2018. URL: https://www. analyticsvidhya.com/blog/2017/10/fundamentals-deep-learning-activationfunctions-when-to-use-them.

[21] Glorot, X., Bordes, A., & Bengio, Y. (2011). *Deep sparse rectifier neural networks.* In Proceedings of the fourteenth international conference on artificial intelligence and statistics (pp. 315-323).

[22] Molina, C. R. R., & Vila, O. P. (2017). *Solving internal covariate shift in deep learning with linked neurons*. *arXiv preprint arXiv:1712.02609*.

[23] Maind, S. B., & Wankar, P. (2014). *Research paper on basic of artificial neural network*. International Journal on Recent and Innovation Trends in Computing and Communication, 2(1), (pp.96-100).

[24] Schmidhuber, J. (2015). *Deep learning in neural networks*: *An overview. Neural networks*, *61*, (pp.85-117).

[25] Glorot, X., Bordes, A., & Bengio, Y. (2011). Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics* (pp. 315-323).

[26] Bengio, Y. (2009*). Learning deep architectures for AI.* Foundations and trends® in Machine Learning, 2(1), 1-127.

[27] Liu, B., Wang, M., Foroosh, H., Tappen, M., & Pensky, M. (2015). *Sparse convolutional neural networks.* In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (pp. 806-814).

[28] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). *Dropout: a simple way to prevent neural networks from overfitting*. The Journal of Machine Learning Research, 15(1), (pp. 1929-1958).

[29] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). *Dropout: a simple way to prevent neural networks from overfitting.* The Journal of Machine Learning Research, 15(1), (pp. 1929-1958). Accessed: 20-01-2018. URL http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf

[30] Chen, Z., & Yi, D. (2017). *The Game Imitation: Deep Supervised Convolutional Networks for Quick Video Game AI. arXiv preprint arXiv:1702.05663*.

[31] Engineering, H. (2015). *Introduction to convolution neural networks*. Accessed: 31-12-2018. URL https://engineering.huew.co/introduction-toconvolution-neural-networks-18981d1cd09a

[32] FPGA, I. (2017). *Introduction to intel fpga ip cores*. Accessed: 31-12-2018. URL https://www.altera.com/documentation/ mwh1409960636914.html.

[33] Floydhub, (2018). Accessed: 31-12-2018. URL https://docs.floydhub.com/

[34] Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). *Learning representations by back-propagating errors*. nature, 323(6088), (pp. 533).

[35] Waibel, A., Hanazawa, T., Hinton, G., Shikano, K., & Lang, K. (1988). *Phoneme recognition: neural networks vs. hidden Markov models vs. hidden Markov models. In Acoustics, Speech, and Signal Processing,* 1988. ICASSP-88., 1988 International Conference on (pp. 107-110). IEEE.

[36] Reese, M. G. (2001). *Application of a time-delay neural network to promoter annotation in the Drosophila melanogaster genome. Computers & chemistry*, *26*(1), 51-56.

[37] Siegelmann, Hava T., and Eduardo D. Sontag. "Turing computability with neural nets." Applied Mathematics Letters4.6 (1991): (pp. 77-80).

[38] Hyötyniemi, H. (1996). *Turing machines are recurrent neural networks. Proceedings of step*, (pp. *96)*.

[39] Siegelmann, H. T., & Sontag, E. D. (1995). *On the computational power of neural nets. Journal of computer and system sciences*, *50*(1), (pp. 132-150).

[40] FPGA, I., Platform designer. (2018). Accessed: 31-12-2018 URL https://www.altera.com/products/design-software/fpgadesign/quartus-prime/features/qts-platform-designer.html.

[41] Hochreiter, S., & Schmidhuber, J. (1997). *Long short-term memory. Neural computation, 9*(8), (pp. 1735-1780).

[42] Gers, F. A., & Schmidhuber, J. (2000). *Recurrent nets that time and count.* In Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium Vol. 3, (pp. 189-194)

[43] Schnable, P. S., Ware, D., Fulton, R. S., Stein, J. C., Wei, F., Pasternak, S., ... & Minx, P. (2009). *The B73 maize genome: complexity, diversity, and dynamics*. *science*, *326* (5956), (pp. 1112-1115).

[44] Bartalena, L., & Fatourechi, V. (2014). *Extrathyroidal manifestations of Graves' disease*: a 2014 update. Journal of Endocrinological Investigation, 37(8), 691-700.

[45] Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014). *Empirical evaluation of gated recurrent neural networks on sequence modeling.* arXiv preprint arXiv:1412.3555.

[46] Chen, J., & Wang, D. (2017). *Long short-term memory for speaker generalization in supervised speech separation.* The Journal of the Acoustical Society of America, 141(6), (pp. 4705-4714).

[47] Gers, F. A., Schmidhuber, J., & Cummins, F. (1999*). Learning to forget*: Continual prediction with LSTM.

[48] Chen, Z., & Yi, D. (2017). *The Game Imitation: Deep Supervised Convolutional Networks for Quick Video Game AI*. arXiv preprint arXiv:1702.05663.

[49] World, I., (2018). *5 guidelines for building a neural network architecture*. Accessed: 31-12-2018. URL https://www.infoworld.com/article/3155052/technologybusiness/5-guidelines-for-building-a-neural-networkarchitecture.html

[50]. Tensorboard. (2018). Accessed: 31-12-2018. URL http://huscse0budlt216:6006/#graphs&_showDownloadLinks=true&runSelectionState=eyI2MC1TRVEtMy1Nb2RlbC1NTFAiOmZhbHNlLCI2MC1TRVEtMy1Nb2RlbC1NTFAtVGltZS0yN1xcMTJcXDIwMTgiOmZhbHNlLCI2MC1TRVEtMy1Nb2RlbC1NTFAtVGltZS0yNzEyMjAxOCI6ZmFsc2UsIjYwLVNFUS0zLU1vZGVsLUNOTi1UaW1lLTI3MTIyMDE4IjpmYWxzZSwiNjAtU0VRLTMtTW9kZWwtTFNUTS1UaW1lLTI3MTIyMDE4IjpmYWxzZSwiNjAtU0VRLTMtTW9kZWwtTUxQLVRpbWUtMjgxMjIwMTgiOnRydWV9&run=60-SEQ-3-Model-CNN-Time-27122018 Accessed: 31-12-2018

[51] Tensorboard. (2018). Accessed: 31-12-2018. URL http://huscse0budlt216:6006/#graphs&_showDownloadLinks=true&runSelectionState=eyI2MC1TRVEtMy1Nb2RlbC1NTFAiOmZhbHNlLCI2MC1TRVEtMy1Nb2RlbC1NTFAtVGltZS0yN1xcMTJcXDIwMTgiOmZhbHNlLCI2MC1TRVEtMy1Nb2RlbC1NTFAtVGltZS0yNzEyMjAxOCI6ZmFsc2UsIjYwLVNFUS0zLU1vZGVsLUNOTi1UaW1lLTI3MTIyMDE4IjpmYWxzZSwiNjAtU0VRLTMtTW9kZWwtTFNUTS1UaW1lLTI3MTIyMDE4IjpmYWxzZSwiNjAtU0VRLTMtTW9kZWwtTUxQLVRpbWUtMjgxMjIwMTgiOnRydWV9&run=60-SEQ-3-Model-CNN-Time-27122018 Accessed: 31-12-2018

[52] Tensorboard. (2018). Accessed: 31-12-2018. URL http://huscse0budlt216:6006/#graphs&_showDownloadLinks=true&runSelectionState=eyI2MC1TRVEtMy1Nb2RlbC1NTFAiOmZhbHNlLCI2MC1TRVEtMy1Nb2RlbC1NTFAtVGltZS0yN1xcMTJcXDIwMTgiOmZhbHNlLCI2MC1TRVEtMy1Nb2RlbC1NTFAtVGltZS0yNzEyMjAxOCI6ZmFsc2UsIjYwLVNFUS0zLU1vZGVsLUNOTi1UaW1lLTI3MTIyMDE4IjpmYWxzZSwiNjAtU0VRLTMtTW9kZWwtTFNUTS1UaW1lLTI3MTIyMDE4IjpmYWxzZSwiNjAtU0VRLTMtTW9kZWwtTUxQLVRpbWUtMjgxMjIwMTgiOnRydWV9&run=60-SEQ-3-Model-LSTM-Time-27122018

[53] Brownlee, J. (2017). Gentle Introduction to the Adam Optimization Algorithm for Deep Learning.

[54] Tensorboard. (2018). Accessed: 31-12-2018. URL http://huscse0budlt216:6006/#scalars&_showDownloadLinks=true&runSelectionState=eyI2MC1TRVEtMy1Nb2RlbC1NTFAiOmZhbHNlLCI2MC1TRVEtMy1Nb2RlbC1NTFAtVGltZS0yN1xcMTJcXDIwMTgiOmZhbHNlLCI2MC1TRVEtMy1Nb2RlbC1NTFAtVGltZS0yNzEyMjAxOCI6ZmFsc2UsIjYwLVNFUS0zLU1vZGVsLUNOTi1UaW1lLTI3MTIyMDE4IjpmYWxzZSwiNjAtU0VRLTMtTW9kZWwtTFNUTS1UaW1lLTI3MTIyMDE4IjpmYWxzZSwiNjAtU0VRLTMtTW9kZWwtTUxQLVRpbWUtMjgxMjIwMTgtcnVuMSI6dHJ1ZSwiNjAtU0VRLTMtTW9kZWwtTUxQLVRpbWUtMjgxMjIwMTgiOmZhbHNlLCI2MC1TRVEtMy1Nb2RlbC1MU1RNLVRpbWUtMjcxMjIwMTgtcnVuMSI6ZmFsc2V9&_smoothingWeight=0.308 Accessed: 31-12-2018

[55] Bitcoin (2008). Accessed: 31-12-2018. URL https://bitcoin.org/en/

[56] Ethereum. Accessed: 31-12-2018. URL https://www.ethereum.org

[57] Litecoin. Accessed: 31-12-2018. URL https://litecoin.org/

[58] Bitcoin Cash. Accessed: 31-12-2018. URL https://www.bitcoincash.org/

[59] Li, J., Cheng, K., Wang, S., Morstatter, F., Trevino, R. P., Tang, J., & Liu, H. (2018). *Feature selection*: A data perspective. *ACM Computing Surveys (CSUR)*, *50*(6), (pp. 94).

[60] Nakamoto, S. (2008). Bitcoin: *A peer-to-peer electronic cash system.* Accessed: 31-12-2018. URL https://bitcoin.org/bitcoin.pdf

[61] Grinberg, R. (2012). *Bitcoin: An innovative alternative digital currency. Hastings Sci. & Tech. LJ*, *4*, (pp. 159).

[62] ElBahrawy, A., Alessandretti, L., Kandler, A., Pastor-Satorras, R., & Baronchelli, A. (2017). *Evolutionary dynamics of the cryptocurrency market*. Royal Society open science, *4*(11).

[63] Chapron, G. (2017). *The environment needs cryptogovernance*. Nature News, *545*(7655), (pp. 403).

[64] Binance.com. (2017). Accessed: 31-12-2018. URL Binance https://www.binance.com/.

[65] P. Inc. (2012). Accessed: 31-12-2018. URL Kraken https://www.kraken.com/.

[66] Python. Accessed: 11-01-2018. URL. https://www.python.org/

[67] Atul. (2018) *Ai vs machine learning vs deep learning*. Accessed: 11-01-2018 URL https://www.edureka.co/blog/author/atul-harshaedureka-co/

[68] Le, Q. V., Ngiam, J., Coates, A., Lahiri, A., Prochnow, B., & Ng, A. Y. (2011). On optimization methods for deep learning. In Proceedings of the 28th International Conference on International Conference on Machine Learning Omnipress (pp. 265-272).

[69] Dabbura Imad, (2017). *Gradient Descent Algorithm and its Vairants.* Accessed: 11-01-2018 URL https://towardsdatascience.com/gradient-descent-algorithm-and-its-variants-10f652806a3

[70] Ramendra Thakur. (2016). *What is the difference between local minima & maxima and absolute minima & maxima?* Accessed: 11-01-2018 URL https://www.quora.com/What-is-the-difference-between-local-minima-maxima-and-absolute-minima-maxima.

[71] Antony Pallupetta, (2015). *Maxima and minima application problems are difficult?* Accessed: 11-01-2018 URL. https://www.quora.com/Maxima-and-minima-application-problems-are-difficult-I-got-stuck-on-how-to-begin-What-should-be-the-approach-and-line-of-thinking-to-be-followed.

[72] Rencher, A. C., & Schaalje, G. B. (2008). *Linear models in statistics*. John Wiley & Sons.

[73] Seber, G. A., & Lee, A. J. (2012). *Linear regression analysis* (Vol. 329). John Wiley & Sons.

[74] Vieira, Sara. Neural Networks. Accessed: 20-01-2018. (2017) URL https://www.researchgate.net/Figura/a-The-building-block-of-deep-neural-networks-artificial-neuron-or-node-Each-input-x_fig1_312205163

[75] Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning (adaptive computation and machine learning series).* Adaptive Computation and Machine Learning series.

[76] Goodfellow, I., Bengio, Y., Courville, A., & Bengio, Y. (2016). *Deep learning* (Vol. 1). Cambridge: MIT press.

[77] Bishop, C. M. (1995). *Neural networks for pattern recognition*. Oxford university press.

[78] Brownlee, J. (2014). Machine learning mastery. Accessed: 21-01-2018. *URL: http://machinelearningmastery. com/discover-feature-engineering-howtoengineer-features-and-how-to-getgood-at-it*.

[79] Ali, A., Shamsuddin, S. M., & Ralescu, A. L. (2015). Classification with class imbalance problem: a review. *Int J Adv Soft Comput Appl*, *7*(3), (pp. 176-204).

[80] Shalev-Shwartz, S., & Ben-David, S. (2014). *Understanding machine learning: From theory to algorithms*. Cambridge university press.