



DEEP NEURAL NETWORKS FOR CRYPTOCURRENCIES PRICE PREDICTION

Bruno Spilak 586460

A thesis submitted for the degree of Master of Science

Reviewers: Prof. Dr. Wolfgang Karl Härdle
Prof. Dr. Stefan Lessmann

filed on May 21, 2018

Contents

1	Deep neural networks	1
1.1	Deep feedforward networks	1
1.1.1	Nonlinear Neurons	1
1.1.2	Multilayer perceptron	4
1.1.3	Back-propagation	5
1.2	Recurrent neural networks	8
1.2.1	Architectures	8
1.2.2	Back-propagation for recurrent neural network	9
1.2.3	The problem of long-term dependencies	11
1.3	Long Short-Term Memory network	12
1.3.1	LSTM architecture	12
1.3.2	Back-propagation for LSTM networks	14
2	Learning a neural network	17
2.1	Statistical learning	17
2.1.1	Maximum likelihood estimation	17
2.1.2	Loss functions	18
2.2	Optimization for neural networks	20
2.2.1	Gradient descent	20
2.2.2	Adaptive optimization algorithms	21
2.2.3	Batch normalization	21
2.3	Generalization methods	22
2.3.1	Model selection	22
2.3.2	Regularization techniques	24
3	Neural networks for time series forecasting	26
3.1	Cryptocurrencies as financial assets	26
3.2	Stock prices and underlying assumptions	26
3.2.1	Time series modelization	26
3.2.2	NLARX model	27
3.3	Cryptocurrencies price movement analyze	28
3.3.1	Data analysis	28
3.3.2	Application: examples with btc returns	29
4	Application: Cryptocurrency portfolio	33
4.1	Data analysis	33
4.1.1	Cryptocurrencies price	33
4.1.2	Technical indicators	34
4.1.3	Fundamental indicators	35
4.2	Model architecture	35
4.2.1	Data preprocessing	36

4.2.2	Baseline model	37
4.2.3	Best model selection: optimization methods and tuning of hyper-parameters	39
4.2.4	Evaluation of the final models	40
4.3	Evaluation of different trading strategies	43
4.3.1	Buy-and-hold strategy	44
4.3.2	Long short strategy	48
5	Discussion and future works	52
5.1	Discussion on the method	52
5.1.1	Data	52
5.1.2	Model architecture	52
5.2	Discussion on the results	52
5.3	Towards a real portfolio management strategy	53
5.3.1	Model, metric and loss functions	53
5.3.2	Risk management	53
5.4	Reinforcement learning	53
6	Conclusion	54
A	Cryptocurrencies	55
A.1	Cryptocurrencies prices	55
A.2	Cryptocurrencies quarterly returns	55
B	Models architectures	58
C	Long short portfolio	62

Abstract

Price prediction is one of the main challenge of quantitative finance. This paper presents a Neural Network framework to provide a deep machine learning solution to the price prediction problem. The framework is realized in three instants with a Multilayer Perceptron (MLP), a simple Recurrent Neural Network (RNN) and a Long Short-Term Memory (LSTM), which can learn long dependencies. We describe the theory of neural networks and deep learning in order to be able to build a reproducible method for our applications on the cryptocurrency market. Since price prediction is used in order to make financial decisions such as trade signals, we compare different approaches of the prediction problem by exploring supervised learning methods in classification tasks. We study these models to predict out-of-sample price directions of height major cryptocurrencies with a rolling window regression method. For that goal, we build a classification problem that predicts if the price of each cryptocurrency will increase or decrease considerably, as a basis for three-months trading strategies. We build different trading strategies, based on long or long/short positions build on our predictions and compare their performance with a passive index investment on the cryptocurrency market that follows CRIX (Trimborn and Härdle; 2016). Cryptocurrencies, Bitcoin being the most famous, are electronic money based on Blockchain technology that can be used as a decentralized alternative to fiat currencies. Thanks to their numerous applications, the cryptocurrency market has experienced an exponential growth during 2017. We compare different weighted portfolios to test how an investor can benefit from fundamental indicators such as market capitalization. We find that LSTM has the best accuracy for predicting directional movements for the most important cryptocurrencies of CRIX and that an equally weighted portfolio beats CRIX on the first quarters of 2017.

Keywords: Deep learning, multilayer perceptron, recurrent neural network, long short-term memory network, cryptocurrencies, CRIX

Acknowledgments

I would like to express my gratitude to my supervisor Professor Härdle for his continuous help on my thesis. His support helped me to understand the cryptocurrency market which was essential for my research. I would also like to thank my second supervisor Professor Lessmann for introducing me to Machine Learning and Neural Networks and helping me with forecasting theory. Besides my advisor, I would like to thank Simon Trimborn for providing the CRIX data that was crucial for my thesis, Victor Cluzel who helped me with LateX, Alla Petukhina who gave me advice on Quantlet format, Faruk Mustafic and Camil Humajick for their support while I was working with DynamicaSoft and all the researcher from the IRTG 1792 for their comments and remarks at Privatissimum Seminar from the Ladislaus von Bortkiewicz Chair of Statistics of Humboldt Universität.

Introduction

On December 17th 2017, Bitcoin prices worth almost over 20 000 dollars while at the beginning of the year, it was selling for 1000 dollars. Was this rapid growth predictable ?

Stock price forecasting is one of the most important task of quantitative finance. Indeed, profits are the guiding force behind most investment choices. Stock market investors need to know the appropriate time to buy or sell stocks in order to maximize their investment return. However, stock market prices do not behave as simple time series. The theory of price prediction is a major discussion topic in Finance. The *Efficient Market Hypothesis*, which states that price prediction is useless for profit maximization, has attempted to give a definite answer. However, with the appearance of Behavioral Finance, many financial economists believe that stock prices are at least partially predictable on the basis of historical stock price patterns, which reinvigorate *Fundamental* and particularly *Technical analysis* as tools for price prediction.

Deep Learning models, particularly deep feedforward neural networks, have already found numerous applications in quantitative finance, such as volatility forecasting. In a supervised learning scheme, neural networks are useful tool for price prediction since no strong assumption is needed for their application, which contrasts with traditional time series models, such as ARIMA and its extensions. Moreover, deep learning architectures catch patterns with an important generalization power and most recent LSTM networks seem more appropriate for sequential data such as time series. Nevertheless, Deep Learning is frequently criticized for lacking a fundamental theory that could crack open its black box.

In this thesis, we explain the neural network theory and investigate how LSTM networks can outperform, in terms of prediction accuracy, former deep neural network architectures, such as MLP and RNN, on the cryptocurrency market, which recently sparked the interest of new investors on the financial market. In order to reflect trading decisions, we show how, in supervised learning, a deep neural network successfully predicts price movements within a classification task. Then, we build a simple investment strategy based on a long-term portfolio comprised of the most important cryptocurrencies from CRIX, the cryptocurrency index from Trimborn and Härdle (2016). These results may be used as a basis for a trading strategy.

We first overview deep MLP, RNN and LSTM models by explaining the basic concept of neural networks, their elements and architectures. Then, we explain the deep learning method for neural networks. Finally, we present our application of these models on the cryptocurrency market by building a simple trading strategy.

Chapter 1

Deep neural networks

In this chapter, we discuss different architectures of *deep neural networks*. We explain how they corresponds to different representation of nonlinear functions, from a static relation thanks to deep *feedforward networks* (section 1.1) to dynamic process thanks to *recurrent neural networks* which, as we will see in sections 1.2 and 1.3, are necessarily very deep architectures. We explain also how these models are built, defining their elements by discussing essentially the notions of neurons and activations functions.

1.1 Deep feedforward networks

Deep feedforward networks, also called *feedforward network* or *multilayer perceptron* (MLP) are of extreme importance to machine learning. They form the basis of all the extended neural networks architectures and have many commercial applications.

The idea behind feedforward neural networks is a non-linear system, which goal is to approximate a function $y = f^*(x)$. A neural network maps the input x into the output \hat{y} through a function $\hat{y} = f(x, \theta)$ that is an approximation of f^* . Its goal is to learn the parameters θ of the mapping to get the best approximation \hat{y} of the targeted output y .

It is called *feedforward* because the information flow is unidirectional, from the input x through the function f and to the output \hat{y} . There are no cycle, or *feedback* connection, where the output is fed back into the network. Networks with such cycles are called *recurrent networks* presented in Section 1.2.

It is called *network* because we can see the model as a directed graph whose nodes are grouped in several levels, which are called *layers*. For example, Figure 1.1 shows a feedforward network with one *input layer*, one *hidden layer* and one *output layer*. By composing in a chain two functions, this network maps the output \hat{y} as follows

$$\hat{y} = f^{(2)}(f^{(1)}(x)) \quad (1.1)$$

where $f^{(1)}$ is the hidden layer and $f^{(2)}$ is the output layer. The number of hidden layers, or the *length* of the chain, gives the *depth* of the network.

Finally, because the role of each node is analogous to a brain neuron, we call the network *neural* and nodes *neurons*. The number of neurons in the hidden layers gives us the *width* of the model. We will first explain how neurons in neural networks works.

1.1.1 Nonlinear Neurons

The basic idea of nonlinear models is to start from a linear model and apply it to a transformed input $\phi(x)$ instead of x , where ϕ is nonlinear transformation. The goal of deep learning is then to determine how to choose the function ϕ . In practice, we do not know *a priori* this function,

thus we need to *learn* it. We can then reformulate the model as follows:

$$\hat{y} = f(x; \theta, w) = \phi(x; \theta)^\top w$$

where we use the parameter θ to learn ϕ and parameter w to map $\phi(x)$ to the desired output (Goodfellow et al.; 2016).

A famous primary example is the **XOR** function, "exclusive or" function, which takes the binary variable $x = (x_1, x_2)^\top \in \{0, 1\}^2$ as input and produce the output $y = f^*(x)$ as follows:

$$y = 1, \text{ if either } x_1 = 1, \text{ or } x_2 = 1,$$

$$y = 0, \text{ if } x_1 = x_2 = 0, \text{ or } x_1 = x_2 = 1$$

To approximate f^* , we need to learn the parameter θ of the function $\hat{y} = f(x; \theta)$. It is easy to show that a linear model, such as Rosenblatt's perceptron, (Rosenblatt; 1958), cannot learn the **XOR** function. The perceptron uses a linear combiner, $g : x \mapsto w^\top x + b$, followed by a threshold function $x \mapsto \mathbb{1}_{x \geq 0}$. By noting $w = (w_1, w_2)$, the weights vector of the perceptron and b , a bias vector, the parameters of the model are $\theta = (w, b)$. The perceptron maps the input x to the output \hat{y} as follows:

$$\hat{y} = f(x; w, b) = \mathbb{1}_{w^\top x + b \geq 0}$$

We can easily see that it is a linear classifier that map an object x , into two classes C_0 and C_1 , when the output variable \hat{y} is 0, respectively 1. Nevertheless, the latter is true only when the two classes are *linearly separable*.

Definition 1.1 (linearly separable)

For $p \geq 1$ two subsets $\mathcal{X}_0, \mathcal{X}_1 \subseteq \mathbb{R}^p$ are called linearly separable if $w \in \mathbb{R}^p$, $b \in \mathbb{R}$ exists with

$$\begin{aligned} b + w^\top x &> 0 \quad \text{for } x \in \mathcal{X}_1, \\ b + w^\top x &\leq 0 \quad \text{for } x \in \mathcal{X}_0. \end{aligned}$$

The perceptron corresponds to an hyperplane defined by $w_1 x_1 + w_2 x_2 + b = 0$, that separates the two classes. However, in the case of the **XOR** classifier, we can easily see that no hyperplane exists, making the linear perceptron incapable of learning the **XOR** function.

For example, let the input-output matrix be:

x_1	x_2	\hat{y}
0	0	0
0	1	1
1	0	1
1	1	0

We have the following equations:

$$0 * w_1 + 0 * w_2 + b \leq 0 \quad \Leftrightarrow \quad b \leq 0 \quad (1.2)$$

$$0 * w_1 + 1 * w_2 + b > 0 \quad \Leftrightarrow \quad b > -w_2 \quad (1.3)$$

$$1 * w_1 + 0 * w_2 + b > 0 \quad \Leftrightarrow \quad b > -w_1 \quad (1.4)$$

$$1 * w_1 + 1 * w_2 + b \leq 0 \quad \Leftrightarrow \quad b \leq -w_1 - w_2 \quad (1.5)$$

We can see that Equation (1.5) is contradictory with Equations (1.3) and (1.4). Indeed, because the inputs are not linearly separable, the **XOR** problem cannot be learned with a linear perceptron. We can solve this problem by using a different function ϕ that map the inputs into another feature space that can represent the classifier.

Let us create a simple feedforward network with one hidden layer with two units, see Figure 1.1. The vector of hidden units $h = (h_1, h_2)$ is computed by the function $f^{(1)} : x \mapsto f^{(1)}(x; W, c)$, the output layer \hat{y} is then computed by the function $f^{(2)} : x \mapsto f^{(2)}(x; w, b)$ using h as input. Thus, the network is represented in the chain function: $f : x \mapsto \hat{y} = f(x; W, c, w, b) = f^{(2)}(f^{(1)}(x))$. We must choose a function for $f^{(1)}$ and for $f^{(2)}$ but we can not use linear functions otherwise the network would be linear. Neural networks uses *nonlinear* perceptrons as a regular function for hidden units. The nonlinear perceptron consists of a fixed nonlinear function g , called an *activation function*, or *squashing function*, applied to Rosenblatt's linear perceptron:

$$h = g(w^\top x + b)$$

where $w = (w_1, \dots, w_p)^\top$ is the weights of the hidden layer, b its biases and g the activation function chosen.

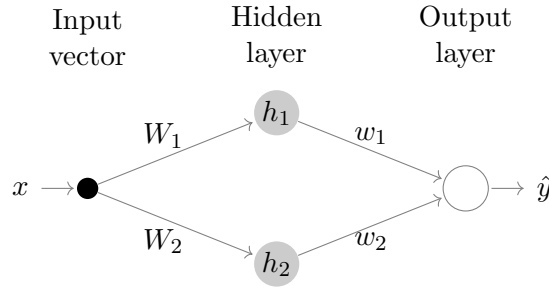


Figure 1.1: Simple neural network with two hidden units, for clarification the biases are not represented

Definition 1.2 (Activation function)

The activation function at hidden neuron i applies element-wise the operation $h_i = g(W_{:,i}^\top x + b_i)$, where $W_{:,i}$ is the vector of weights connected to the neuron i and b_i its bias.

Neural networks use different activation functions and one of the most common function, for a classification task, is the logistic function, often called *sigmoid* function:

$$\sigma(x) = \frac{1}{1 + \exp(-x)}, \quad x \in \mathbb{R} \quad (1.6)$$

Using the sigmoid function as activation function both for the hidden and output neurons, the complete network is then:

$$\hat{y} = f(x; W, c, w, b) = \frac{1}{1 + \exp(-w^\top h - b)}, \text{ with } h = \frac{1}{1 + \exp(-W^\top x - c)}$$

We can now give a solution to the XOR problem. Let

$$W = \begin{pmatrix} 20 & -20 \\ 20 & -20 \end{pmatrix},$$

$$c = \begin{pmatrix} -10 \\ 30 \end{pmatrix},$$

$$w = \begin{pmatrix} 2 \\ 2 \end{pmatrix}$$

and $b = -3$.

Let us explain how the model processes a batch of inputs. Let $x^{(1)}, x^{(2)}, x^{(3)}, x^{(4)} \in \mathbb{R}^2$ be the set of inputs, called the *training set* and which we can represent in matrix form with one example per column:

$$X = \begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

During the first step in the neural network, we multiply the input matrix by the first layer's weight matrix:

$$W^\top X = \begin{pmatrix} 0 & 20 & 20 & 40 \\ 0 & -20 & -20 & -40 \end{pmatrix}$$

Then we add the bias vector c , to obtain:

$$\begin{pmatrix} -10 & 10 & 10 & 30 \\ 30 & 10 & 10 & -10 \end{pmatrix}$$

To get the value of h for each input example, we apply the sigmoid function:

$$h \simeq \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

After this transformation, the inputs are not on a single line anymore, thus, we can find a hyperplane. Let us process this inputs through the final output layer, we multiply by the weight vector w , add the bias b and apply the sigmoid function to get:

$$(1 \ 0 \ 0 \ 1)$$

We can solve the **XOR** problem with different neural networks, using other activation functions. In modern applications, the default use is the *rectified linear unit* or *ReLU* (Jarrett et al.; 2009), defined by the activation function $g(z) = \max(0, z)$, $z \in \mathbb{R}$. A neural network with only one hidden *ReLU* unit and a linear perceptron as output layer solves the **XOR** problem.

1.1.2 Multilayer perceptron

As we can see with the latter classification problem, determining the architecture of a neural network is of great importance. The architecture refers to the structure of the network that is defined by its *depth*, its *width* and the connections between its units. Most neural networks combine hidden layers in a chain structure, as in Equation (1.1), where the output of the previous layer is the input of the next layer:

$$\begin{aligned} h^{(1)} &= g^{(1)}(W^{(1)\top} x + b^{(1)}) \\ h^{(2)} &= g^{(2)}(W^{(2)\top} h^{[1]} + b^{(2)}) \\ &\dots \\ h^{(d)} &= g^{(d)}(W^{(d)\top} h^{[d-1]} + b^{(d)}) \end{aligned}$$

We call this architecture a *multilayer perceptron* (MLP) or *multilayer neural network*. Figure 1.2 represents such a multilayer perceptron.

The question of the depth of the network, when determining the architecture of a neural network is very important. Hornik et al. (1989) showed that a feedforward network with a single hidden layer with sufficiently many hidden units and arbitrary bounded and nonconstant activation functions, such as sigmoid activation function, are universal approximators for finite input. Thus, a multilayer perceptron can approximate any continuous function mapping from a finite dimensional discrete space to another.

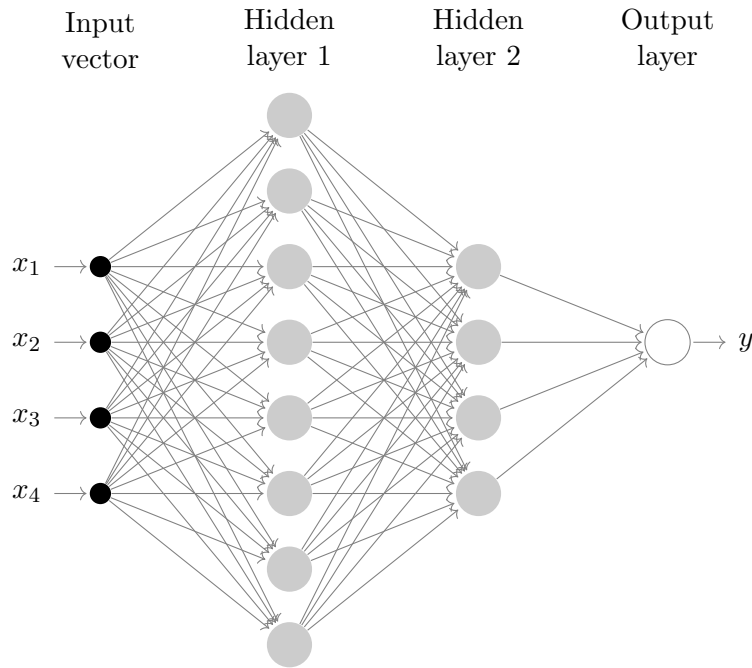


Figure 1.2: Multilayer perceptron

MLP with 4 inputs units corresponding to the input vector $x = (x_1, x_2, x_3, x_4)$, 2 hidden layers with 8 and 4 units respectively and 1 output unit corresponding to the output variable y

The *universal approximation theorem* has also been proved for a wider class of activation functions, such as *ReLU*, (Leshno et al.; 1993). While this theorem means that a sufficiently large MLP can *represent* any function, it does not mean that we can necessarily *learn* it. As we will see in the next chapter, very large network are very slow to train, that is why we often prefer deep rather than large networks.

1.1.3 Back-propagation

Introduction to statistical learning

We know that we can implement a neural network to represent any function from input to output, but we need to define how to find the network parameters based on a training set and output targets.

In the **XOR** function example, we gave a specific solution to the classification problem that has no error. In real application, the training set can have billions of observations that we want to classify in a model that uses many parameters. In that case, we need to *learn* the parameters of the model in a way that they produce the smallest *error* as possible, to get a "good" if not a "perfect" classification (Franke et al.; 2015) of the training set. We define that error as a *loss* function of the parameters, $Q(\theta)$, that we want to minimize with respect to the parameter θ . Each machine learning problem is different, for example, we can use neural networks for regression and classification problems, that is why we define different loss functions, each one corresponding to a specific problem, see Section 2.1.2.

Let us consider the training set $(x_1, y_1), \dots, (x_n, y_n)$ where x_i and y_i are input vectors and target vectors respectively. We estimate the target y by the neural network function $\hat{y} = f(x; \theta)$ and we estimate the parameter θ so that $Q(\theta)$ is minimum, thanks to an optimization algorithm such as *gradient descent*, see Section 2.2. In learning algorithms, the gradient of the cost function, called *error gradient* with respect to the parameter, $\nabla Q(\theta)$, is required at each layer, in order to understand how changes on the weights at the input layer can impact the loss function,

computed at the output layer. This computation is very complicate for deep neural networks and we need a particular numerical method.

Back-propagation algorithm

- **Forward pass:** In feedforward networks, when we feed an input x , information propagates forward through the hidden layers in the network to produce an output \hat{y} . This step is called the *forward pass* (Graves; 2012). This *forward* propagation of information continues while we train the network over the training set until it produces the scalar loss $Q(\theta)$. To be able to update the weights of the network correctly, we need to be able to propagate the information from the loss, backward in the network, during a second step called the *backward pass*.
- **Backward pass:** The *back-propagation* algorithm, or *backprop*, (Rumelheart et al.; 2012), allows this feedback of information that enable the computation of the gradient. It is the most well known method for supervised training of MLP. Backprop is not a learning algorithm; it is only a computation method that allows the network to learn using an optimization algorithm such as *gradient descent*. We will describe the back-propagation algorithm applied to a neural network function, f , with a single output.

To understand how back-propagation works, we can represent a neural network as a computational graph, as on Figure 1.2, where each node applies the nonlinear perceptron operation. An example of computational graph for linear regression is given on Figure 1.3.

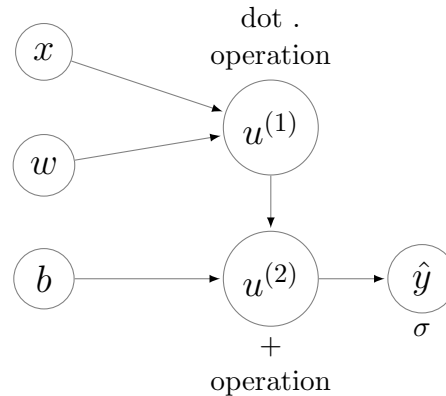


Figure 1.3: Computational graph of linear regression: $\hat{y} = \sigma(w^\top x + b)$

Backpropagation is an algorithm that expressed the error gradient with respect to quantities of a given neuron as a function of its outgoing neurons. This is possible thanks to the *chain rule of calculus* that computes the derivatives of the composition of several functions. Let x be a real number, and f and g real functions. We take the composition of f and g to get the output $z = f(y) = f(g(x))$, the *chain rule* states that:

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x}$$

Using the chain rule, it is easy to explicit the expression for the error gradient of a scalar with respect to any node in the computational graph that produced that scalar. Figure 1.4 represent a part of a multilayer perceptron computational graph, with z_i the activation of unit i , u_i the input of unit i , w_{ij} the weight between unit i and j , b the bias of the perceptron and $Q(\theta)$ the loss function.

From this graph, we can write the error back-propagation explicitly using the partial derivatives.

$$\frac{\partial Q(\theta)}{\partial u_i} = \underbrace{\frac{\partial z_i}{\partial u_i}}_{g'(u_i)} \sum_j \underbrace{\frac{\partial Q(\theta)}{\partial u_j} \frac{\partial u_j}{\partial z_i}}_{w_{ij}},$$

which gives us the backward equation of the back-propagation algorithm:

$$\frac{\partial Q(\theta)}{\partial u_i} = g'(u_i) \sum_j \frac{\partial Q(\theta)}{\partial u_j} w_{ij}$$

We can then compute the gradient parameter as follows:

$$\frac{\partial Q(\theta)}{\partial w_{ij}} = \frac{\partial Q(\theta)}{\partial u_j} \underbrace{\frac{\partial a_j}{\partial w_{ij}}}_{z_i} \text{ and } \frac{\partial Q(\theta)}{\partial b_j} = \frac{\partial Q(\theta)}{\partial u_j} \underbrace{\frac{\partial u_j}{\partial b_j}}_1$$

which gives:

$$\frac{\partial Q(\theta)}{\partial w_{ij}} = \frac{\partial Q(\theta)}{\partial u_j} z_i \text{ and } \frac{\partial Q(\theta)}{\partial b_j} = \frac{\partial Q(\theta)}{\partial u_j}$$

We can generalize the implementation of the back-propagation algorithm to any multilayer perceptron, using its matrix form.

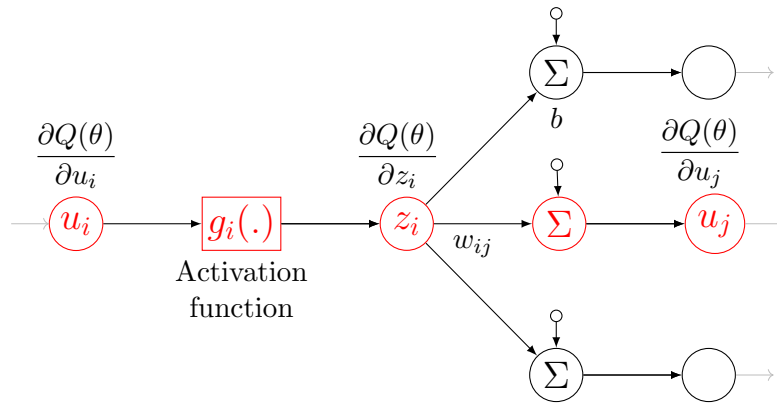


Figure 1.4: Error back-propagation

Back-propagation in matrix form

Let us consider a multilayer perceptron with:

$$\mathbf{z}_l : \text{vector of neuron activations at layer } l \quad (1.7)$$

$$\mathbf{W}_{l,l+1} : \text{matrix of weights connecting layer } l \text{ to layer } l+1 \quad (1.8)$$

$$\mathbf{b}_l : \text{vector of biases at layer } l \quad (1.9)$$

$$\mathbf{g}_i(\cdot) : \text{function that applies elementwise the activation} \quad (1.10)$$

$$\text{function of unit } i, g_i(\cdot) \quad (1.11)$$

- **Forward equations:**

$$\mathbf{u}_{l+1} = \mathbf{W}_{l,l+1}^\top \cdot \mathbf{z}_l + \mathbf{b}_{l+1} \quad (\text{linear perceptron}) \quad (1.12)$$

$$\mathbf{g}_{l+1} = \mathbf{g}(\mathbf{u}_{l+1}) \quad (1.13)$$

- **Backward equations:**

$$\frac{\partial Q(\theta)}{\partial \mathbf{z}_l} = \mathbf{W}_{l,l+1} \frac{\partial Q(\theta)}{\partial \mathbf{u}_{l+1}} \quad (1.14)$$

$$\frac{\partial Q(\theta)}{\partial \mathbf{u}_l} = \mathbf{g}'(\mathbf{u}_l) \odot \frac{\partial Q(\theta)}{\partial \mathbf{z}_l} \quad \text{where } \odot \text{ is the element-wise product} \quad (1.15)$$

- **Weights updates:**

$$\frac{\partial Q(\theta)}{\partial \mathbf{W}_{l,l+1}} = \mathbf{z}_l \cdot \frac{\partial Q(\theta)}{\partial \mathbf{u}_{l+1}}^\top \quad (1.16)$$

$$\frac{\partial Q(\theta)}{\partial \mathbf{b}_l} = \frac{\partial Q(\theta)}{\partial \mathbf{u}_l} \quad (1.17)$$

1.2 Recurrent neural networks

In the previous section, we considered MLPs without any cyclic connection, which makes MLP a static model in the sense that the input-output pairs are mutually independent (Back and Tsoi; 1991). This seems inefficient if we want to modelize sequential data, where the input features are interdependent. If we allow such cyclic connections, we obtain a *recurrent neural networks*, (RNN), which can modelize dynamic processes, for example time series.

Indeed, when we are interested in such dynamic process, we need a model that can represent the relation between the previous input-output pair and the next one, where the output is a function of the previous output. That is the model need to process examples one at a time, retaining memory that represents a contextual information that we can reuse at the next time step. Thanks to this recurrent formulation, by comparison to a multilayer perceptron, a recurrent neural network shares the same weights across several time steps (Goodfellow et al.; 2016). Recurrent networks are now used in various applications such as stock price forecasting, language or music processing.

1.2.1 Architectures

Jordan (1986) presented a first architecture as a superset of feedforward artificial neural networks that has one or more cycles. Each cycle make possible to follow a path from a neuron back to itself, allowing feedback of information. These cycles, or recurrent edges, allow the network's hidden units to see its own previous output so they give the network memory (Elman; 1990) and introduce the notion of time into the model. The recurrent neurons are sometime referred to as *context neurons* or *state neurons*. The structure of a simple recurrent neural network is shown on Figure 1.5.

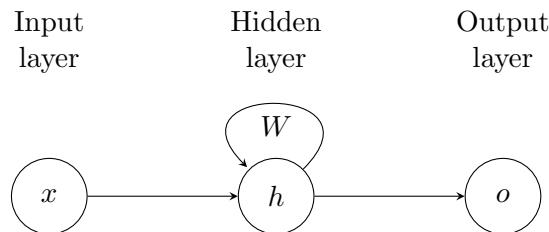


Figure 1.5: Simple Recurrent Neural Network with one hidden layer

Jordan (1986) introduced the notion of time in the model thanks to state units that represent the "temporal context". The output information is produced from the input and state units

through the hidden units. Finally, the recurrent connections from the state units to themselves and from the output units to the state units allow the output information to be fed back in the network at the following time step. These recurrent connections constitute Jordan network's memory as in Figure 1.6.

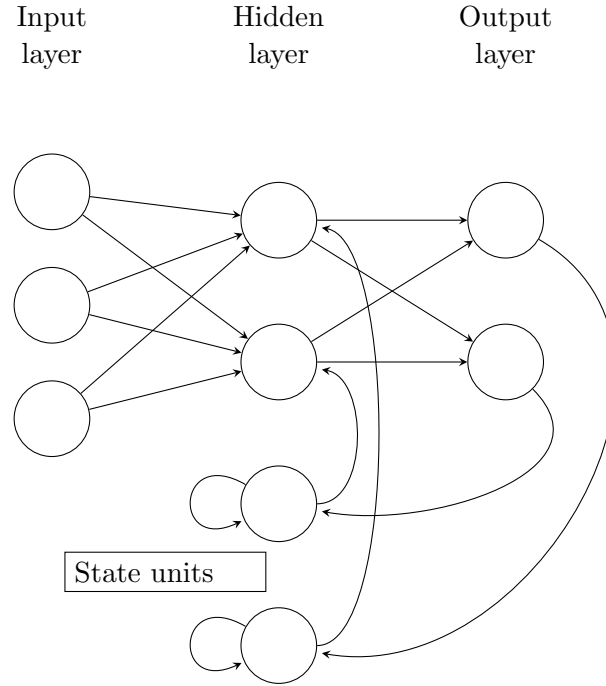


Figure 1.6: Jordan Recurrent Neural Network

Let us now explain how RNN process a sequence of inputs.

1.2.2 Back-propagation for recurrent neural network

The most used algorithm to calculate weights derivatives for RNN is a slightly modified version of the back-propagation algorithm from Section 1.1.3. It is called back-propagation through time, BPTT. The idea behind BPTT is to *unfold* the recurrent neural network in time (Rojas; 1996), by stacking identical copies of the RNN and then apply the standard back-propagation algorithm. Indeed, this unfolding process results from the fact that recurrent neural networks share weights across time steps. In comparison, a traditional MLP has separate parameters for each input feature. Recurrent neural networks lie at the core of deep learning because, by unfolding these networks, we obtain very deep architectures. The unfolded graph of the simple recurrent neural network from Figure 1.5 is given on Figure 1.7.

Back-propagation through time

As for the multilayer perceptron, BPTT as two steps:

- **Forward pass:** The forward pass of an RNN is the same as that of an MLP with a single hidden layer, but the activations at the hidden layer comes from both the current input and the hidden layer activations at the previous time step (Graves; 2012).

Let us consider an input sequence, x , of length T presented to a RNN with I input units, one hidden layer with H hidden units and K output units. Let us also denote \mathcal{I} the index interval for the input to hidden layer connections, \mathcal{H} the index interval for the hidden to

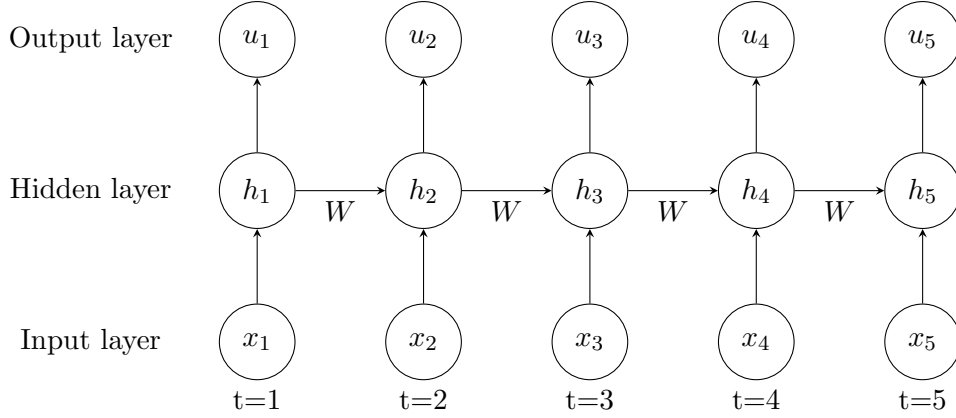


Figure 1.7: Unfolded Recurrent Neural Network for 5 time steps

hidden layer connections (recurrent connections) and \mathcal{K} the index interval for the hidden to output layer connections.

Consider the following notation:

- x_i^t : the value at time t of the input unit i , $i \in \mathcal{I}$
- u_j^t the network input to unit j , $j \in \mathcal{I}$ or $j \in \mathcal{K}$
- z_j^t the activation of unit j at time t , $j \in \mathcal{H}$ or $j \in \mathcal{K}$
- w_{ij} the weights of the network where i and j belongs to \mathcal{I} , \mathcal{H} , \mathcal{K} indifferently. That is w_{ih} is the weight of the connection between input unit i and hidden unit h , $w_{h'h}$ the weight of the connection between hidden unit h' and hidden unit h and finally, w_{hk} the weight of the connection between hidden unit h and output unit k .

This notation, inspired from Williams and Zipser (1995) and Graves (2012), will make the equations of the algorithm easier.

As in a regular MLP, see Section 1.1.3, we have for the hidden units:

$$u_h^t = \sum_{i=1}^I w_{ih} x_i^t + \sum_{h'=1}^H w_{h'h} z_{h'}^{t-1} \quad h = 1, \dots, H \quad (1.18)$$

Activation functions are then applied exactly as for MLPs to get the output of hidden unit h , z_h^t , and the output units of the network, u_k^t at time t :

$$z_h^t = g(u_h^t) \quad h = 1, \dots, H \quad (1.19)$$

$$u_k^t = \sum_{h=1}^H w_{hk} z_h^t \quad k = 1, \dots, K \quad (1.20)$$

To ease the notation we did not consider the bias vector, b_i at unit i . By applying Equations (1.18) and (1.19) for every $t = 0, \dots, T$, we get all the activations for the hidden layer. Nevertheless, we must choose an initializer for z_i^0 , corresponding to the network's state before it receives any information for the input dataset. We often set it to 0, which works well for sequence-to-sequence learning, but in some cases, we can use non-zero or noisy initial state, as in Zimmermann et al. (2006).

- **Backward pass:** To compute the backward pass, we need derivatives with respect to the weights. We use the back-propagation algorithm that applies standard back-propagation

to the unfolded RNN as in Figure 1.7 thanks to the chain rule (see Section 1.1.3). But in the case of RNN, both the output layer at the current time step and the hidden layer at the next time step are influenced by the activation on the hidden layer at the current time step (Graves; 2012), which gives the following back-propagated error:

$$\frac{\partial Q(\theta)}{\partial u_h^t} = g'(u_h^t) \left(\sum_{k=1}^K \frac{\partial Q(\theta)}{\partial u_k^t} w_{hk} + \sum_{h'=1}^H \frac{\partial Q(\theta)}{\partial u_{h'}^{t+1}} w_{h'h} \right) \quad t = 1, \dots, T-1 \quad (1.21)$$

We first compute $\frac{\partial Q(\theta)}{\partial u_h^T}$ with $t = T$ and then apply Equation 1.21 recursively to get each term for $t = 1, \dots, T-1$.

Finally, because, in the unfolded graph, the repeated weights are duplicated to get unique ones across time steps in the hidden layer, we can use standard back-propagation as usual, the difference being that we sum up the gradients at each time step. Thus for all weights in the network, we get the following update rule:

$$\frac{\partial Q(\theta)}{\partial w_{ij}} = \sum_{t=1}^T \frac{\partial Q(\theta)}{\partial u_j^t} \frac{\partial u_j^t}{\partial w_{ij}} = \begin{cases} \sum_{t=1}^T \frac{\partial Q(\theta)}{\partial u_i^t} x_i^t, & \text{if } i \in I \text{ and } j \in H \\ \sum_{t=1}^T \frac{\partial Q(\theta)}{\partial u_h^t} z_{h'}^t, & \text{if } (i, j) \in H^2 \\ \sum_{t=1}^T \frac{\partial Q(\theta)}{\partial u_k^t} z_h^t, & \text{if } i \in H \text{ and } j \in K \end{cases} \quad (1.22)$$

1.2.3 The problem of long-term dependencies

We saw that RNN are very useful when we want to modelize a present input thanks to previous information, but in practice, they cannot always explain this relation. We refer to this problem as *the vanishing gradient problem* or *the exploding gradient*. Indeed, what if the context information, that we need to explain the current task, must be find at a very deep time step, rather than the previous one. The computational graph of such a recurrent network would be very deep. A simple RNN is unlikely to understand this long-term dependency. We need to explain further the computation of the gradient with the back-propagation algorithm.

Let us illustrate this phenomenon with the simple RNN from Figure 1.7 with $t = 1, \dots, T$. Let us suppose that the activation function is the logistic sigmoid, we can rewrite Equation (1.22) using the chain rule:

$$\frac{\partial Q(\theta)}{\partial W} = \sum_{t=1}^T \frac{\partial Q(\theta)}{\partial u_t} \frac{\partial u_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W} \quad 1 \leq k \leq T$$

and applying again the chain rule on $\frac{\partial h_t}{\partial h_k}$, we get:

$$\frac{\partial Q(\theta)}{\partial W} = \sum_{t=1}^T \frac{\partial Q(\theta)}{\partial u_t} \frac{\partial u_t}{\partial h_t} \left(\prod_{i=k+1}^t \frac{\partial h_i}{\partial h_{i-1}} \right) \frac{\partial h_k}{\partial W} \quad 1 \leq k \leq T \quad (1.23)$$

$\prod_{i=t+1}^T \frac{\partial h_j}{\partial h_{j-i}} = \prod_{i=k+1}^T W^\top \text{diag}(\sigma'(h_{j-1}))$, where σ is the sigmoid activation function, is a Jacobian matrix and as a norm inferior to 1. Indeed, we use the sigmoid activation function that squashes the output values in $[0, 1]$ and the derivatives $[0, 1/4]$, see Pascanu et al. (2013) for a complete demonstration. Thus, sigmoid layers can easily squash their input to a smaller output region. If it happens repeatedly on stacked multiple sigmoid layers, even a large change in the parameters of the first layer can finally have a really small impact on the output.

Indeed, according to Equation (1.23), long-term contributions, for which $T - k$ is large, can go to 0 exponentially fast with the $T - k$ order matrix multiplication, if the absolute value of the largest eigenvalue of the recurrent weight matrix W is inferior to some boundary γ .

Pascanu et al. (2013) showed that for \tanh and *sigmoid* activation functions, $\gamma = 1$ and $\gamma = 1/4$ respectively.

Moreover, the long-term components can explode instead of vanishing. We can invert the latter condition to get a necessary condition for *exploding gradient*.

Solution to the vanishing gradient problem

- Williams and Zipser (1995) built an approximation to the BPTT algorithm by truncating the backward propagation of information to a fixed number of prior time steps, called Truncated Back-Propagation Through Time, TBPTT. They showed that when vanishing gradient occurs the TBPTT algorithm can give a good approximation to the true error gradient.
- Using a L_1 or L_2 weight penalty on the recurrent weights can also be a solution. Pascanu et al. (2013) address the vanishing gradients problem using a regularization term such that back-propagated gradients neither increase or decrease too much in magnitude.
- Pascanu et al. (2013) proposed a *gradient clipping* to deal with exploding gradients. They rescale the gradients, g , whenever their norm, $\|g\|$ go over a threshold, v :

$$\text{If } \|g\| > v \quad \text{then } g \leftarrow \frac{gv}{\|g\|}$$

- Glorot et al. (2011) proposed to use Rectifier Unit, ReLU, which activation function is $\text{rectifier}(x) = \max(0, x)$, $x \in \mathbb{R}$. The function computed by each neuron is then linear by parts and because of this linearity, there is no vanishing gradient due to nonlinear activations like *tanh* or *sigmoid*. Indeed, ReLU does not have this property of squashing the input space into smaller region.
- We can also change the structure of the model to cope with the vanishing gradients problem. By introducing a new set of units called Long Short-Term Memory units (LSTM network), Hochreiter and Schmidhuber (1997) enforce a constant error flow through "constant error carousels", thanks to input and output gates.

1.3 Long Short-Term Memory network

This section of the chapter discusses one of the most used approaches to cope with the difficulty of learning long-term dependencies, *LSTM units*. This problem remains an important challenge in deep learning and we could have chosen other solutions such as *Echo State Networks* or *Leaky Units*, one can refer to Goodfellow et al. (2016) for a general introduction.

We chose to discuss about *LSTM networks*, because it is one of the most effective sequence models, with the *Gated Recurrent Unit* (GRU). Both are *gated* recurrent neural networks. The idea behind GRU and LSTM units is to create connections through time with a *constant error flow*, thus the gradient neither explodes nor vanishes. LSTM networks are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior, that is why they are the most popular neural network model for sequence learning.

1.3.1 LSTM architecture

The architecture of LSTM is similar to RNN, the difference being that, as RNN are supersets of MLP, LSTM networks are supersets of recurrently connected subnets, known as memory blocks, (Graves; 2012). The basic memory block has three single neural network layers, interacting with each other instead of one:

- One (or more) memory cell s_c , called the *cell state*, is the central feature. It is referred to as *constant error carousel* (CEC) in Hochreiter and Schmidhuber (1997). The cell state produces only some minor linear transformations, achieving a constant error flow through the memory block. It is a linear unit with a fixed recurrent self-connection. Controlling the cell state, *gate cells* are added to the memory cell(s).
- One multiplicative *input gate unit* is introduced to protect the current memory content stored in s_c to be perturbed by previous states.
- One multiplicative *output gate unit* is also introduced to protect next units to be perturbed by the currently irrelevant memory content.

These gates gives to LSTM the ability to control the information flow in the cell state and have a sigmoid activation function, which gives the amount of information to let trough. They are closed when the activation is close to 0 and are opened when the activation is close to 1. Thus, the input gate decides when to *keep* or *override* information in the memory cell.

With this architecture, the cell state, s_c , is updated based on its current state and three sources of inputs: net_c , the input to the sell itself through the recurrent connection, net_{in} and net_{out} the inputs to the input and output gates, (Gers et al.; 2000). At each time step, during the forward pass, all units are updated and the error signals for all weights are computed during the backward pass.

If LSTM have found a large success and numerous applications, Gers et al. (2000) identified a weakness when they process continual input streams, without explicitly resetting the network state. In some case, the cell state tends to grow linearly during learning which can make the LSTM cell degenerate into an ordinary recurrent network, where the gradient vanishes, see Gers et al. (2000). Gers et al. (2000) proposed to add to the memory block a *forget gate* that allows the memory block to reset itself, thanks to a *sigmoid* activation function. For now on, we will only consider the extended LSTM with forget gates from Gers et al. (2000). Figure 1.8 illustrates such a LSTM cell.

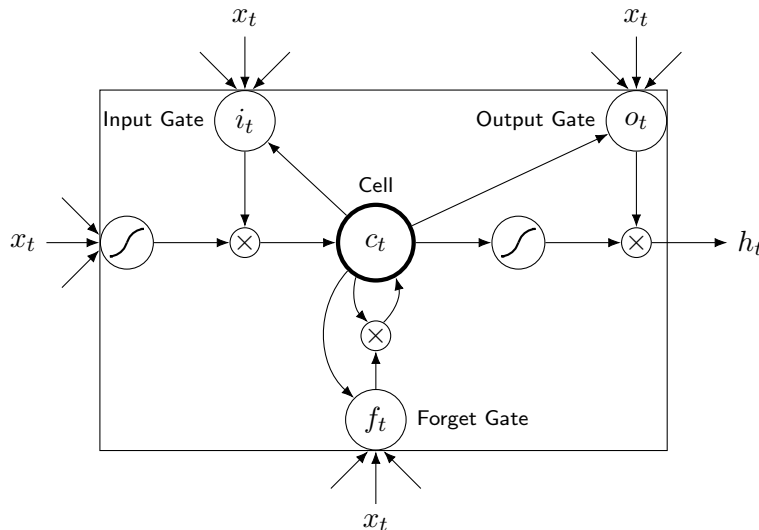


Figure 1.8: LSTM cell with a forget gate, source: Graves (2013)

Let x_t be one observation at time t of the input vector, the LSTM cell from Figure 1.8 is implemented by the following equations (Graves; 2013):

$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i) \quad (1.24)$$

$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f) \quad (1.25)$$

$$c_t = f_t c_{t-1} + i_t \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c) \quad (1.26)$$

$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_t + b_o) \quad (1.27)$$

$$h_t = o_t \tanh(c_t) \quad (1.28)$$

where σ is the logistic *sigmoid* function, and i , f , o , c are respectively the input gate, forget gate, output gate and memory cell activations. W_{xi} , W_{hi} , W_{xf} , W_{hf} , W_{cf} , W_{xc} , W_{hc} , W_{xo} , W_{ho} , W_{co} , which indexes are intuitive, are the input-input gate weight matrix, hidden-input gate weight matrix, etc.

1.3.2 Back-propagation for LSTM networks

Like previous neural networks, LSTM are trained with gradient descent that requires the error gradient. In this section, we present the computation of the exact LSTM gradient with BPTT. We use the following notations:

- $w_{i,j}$ the weight of the connection between units i and j
- u_j^t the network input to unit j at time t
- z_j^t the activation of unit j at time t
- indexes i , f and o refer to input gate, forget gate and output gate respectively.
- c refers to one of the C memory cells
- s_c^t is the state for cell c at time t
- f the activation function of the gates
- g and h are respectively the cell input and output activations functions.
- I the number of inputs
- K the number of outputs
- H the number of cells in the hidden layer
- h refer to the hidden to hidden unit connections.

The forward and backward passes are calculated as in Section 1.2.2. For more details, the reader can refer to Graves (2012) from which we took the equations.

Forward pass

Let us introduce:

$$\delta_j^t = \frac{\partial Q(\theta)}{\partial u_j^t} \quad (1.29)$$

We have the following equations for each gate:

Input gates

$$u_i^t = \sum_{i=1}^I w_i x_i^t + \sum_{h=1}^H w_{hi} z_h^{t-1} + \sum_{c=1}^C w_{ci} s_c^{t-1} \quad (1.30)$$

$$z_i^t = f(u_i^t) \quad (1.31)$$

$$(1.32)$$

Forget gates

$$u_f^t = \sum_{i=1}^I w_{if} x_i^t + \sum_{h=1}^H w_{hf} z_h^{t-1} + \sum_{c=1}^C w_{cf} s_c^{t-1} \quad (1.33)$$

$$z_f^t = f(u_f^t) \quad (1.34)$$

Cells

$$u_c^t = \sum_{i=1}^I w_{ic} x_i^t + \sum_{h=1}^H w_{hc} z_h^{t-1} \quad (1.35)$$

$$s_c^t = z_f^t s_c^{t-1} + z_i^t g(u_c^t) \quad (1.36)$$

Output gates

$$u_o^t = \sum_{i=1}^I w_{io} x_i^t + \sum_{h=1}^H w_{ho} z_h^{t-1} + \sum_{c=1}^C w_{co} s_c^{t-1} \quad (1.37)$$

$$z_o^t = f(u_o^t) \quad (1.38)$$

Cell Outputs

$$z_c^t = z_o^t h(s_c^t) \quad (1.39)$$

Backward pass

Let us introduce some notations:

$$\epsilon_c^t = \frac{\partial Q(\theta)}{\partial z_c^t} \quad \epsilon_s^t = \frac{\partial Q(\theta)}{\partial s_c^t} \quad (1.40)$$

Cell Outputs

$$\epsilon_c^t = \sum_{k=1}^K w_{ck} \delta_k^t + \sum_{h=1}^H w_{ch} \delta_h^{t+1} \quad (1.41)$$

Output gates

$$\delta_o^t = \frac{\partial f(u_o^t)}{\partial u_o^t} \sum_{c=1}^C h(s_c^t) \epsilon_c^t \quad (1.42)$$

states

$$\epsilon_s^t = z_o^t \frac{\partial h(s_c^t)}{\partial s_c^t} \epsilon_c^t + z_f^{t-1} \epsilon_s^{t+1} + w_{ci} \delta_i^{t+1} + w_{cf} \delta_f^{t+1} + w_{co} \delta_o^t \quad (1.43)$$

Cells

$$\delta_c^t = z_i^t g'(u_c^t) \epsilon_s^t \quad (1.44)$$

Forget gates

$$\delta_f^t = \frac{\partial f(u_f^t)}{\partial u_f^t} \sum_{c=1}^C s_c^{t-1} \epsilon_s^t \quad (1.45)$$

Input gates

$$\delta_i^t = \frac{\partial f(u_i^t)}{\partial u_i^t} \sum_{c=1}^C g(u_c^t) \epsilon_s^t \quad (1.46)$$

Thanks to its complex architecture, LSTM is easier to train than RNN. Nevertheless, we need to define how training occurs in neural networks.

Chapter 2

Learning a neural network

In the previous chapter, we discussed the architecture of neural networks and how they can represent a mapping from an input x to an output $\hat{y} = f(x; \theta)$ that approximates the targeted output $y = f^*(x)$ where f^* is unknown. To do that, neural networks need to learn the parameter θ . In this chapter, we will explain how to learn θ to get the best approximation of y . The deep learning procedure consists of choosing a specification for the dataset, $f(x; \theta)$, a *loss function*, that corresponds to the task, and an optimization algorithm for a given model (Goodfellow et al.; 2016). We will explain these three steps.

2.1 Statistical learning

In machine learning, we distinguish mainly three learning methods (Duda et al.; 2000):

- *Supervised learning*, where each input of a dataset is associated with a label or target. We have a teacher that provides a cost for each input that we want to reduce. Most tasks are classification or regression problems. In this thesis, we only discuss about supervised learning algorithms.
- *Unsupervised learning* or *clustering* tries to learn structures in the dataset. There is no explicit teacher and the model forms clusters of input patterns.
- *Reinforcement learning* interacts with an environment and lies in between supervised and unsupervised learnings. It is similar to supervised learning with a feedback, a *critic*, that we get from the real targeted category label to improve the classifier.

2.1.1 Maximum likelihood estimation

The goal of supervised learning algorithms is to estimate an unknown probabilistic distribution $p(y | x)$. This distribution is called the *true probability*. To estimate it, we can use a family of known distributions indexed by a parameter θ , $p(y | x; \theta)$. For example, if we want to predict the future prices of a stock, we can assume that the price follows a nonlinear autoregressive process (NAR) and estimate it with a RNN. The *maximum likelihood estimation* for the parameter θ will give us the best approximation for the true probability. Most modern neural networks are trained using maximum likelihood (Goodfellow et al.; 2016).

Definition 2.1 (Maximum likelihood estimator) *Consider the data generating distribution $p(x)$, which is the true probability. Consider the training set $\mathbf{x} = \mathbf{x}_1, \dots, \mathbf{x}_n$ independently distributed from $p(x)$. Let $p(x; \theta)$ be a parametric family of probability distributions over the space*

\mathbf{x} estimating the true probability. The maximum likelihood estimator for θ is then defined as:

$$\begin{aligned}\theta_{MV} &= \arg \max_{\theta} p(\mathbf{x}; \theta) \\ &= \arg \max_{\theta} \prod_{i=1}^n p(\mathbf{x}_i; \theta)\end{aligned}$$

and if we take the logarithm, which has no incidence to the value of $\arg \max$, we transform the product in a sum, which is much more easier to deal with:

$$\theta_{MV} = \arg \min_{\theta} \sum_{i=1}^n \ln p(\mathbf{x}_i; \theta) \quad (2.1)$$

and we define $L(\theta)$ as the log-likelihood function:

$$L(\theta) = \ln p(\mathbf{x}; \theta) = \sum_{i=1}^n \ln p(\mathbf{x}_i; \theta) = \mathbb{E}_{\hat{p}} \ln p(\mathbf{x}; \theta) \quad (2.2)$$

The usual approach to find θ_{MV} is then to minimize the negative log-likelihood, $-\sum_{i=1}^n \ln p(\mathbf{x}_i; \theta)$.

We can interpret maximum likelihood estimation as minimizing the Kullback-Leibler distance (KL distance) between the model probability, $p(\mathbf{x}; \theta)$ and the empirical distribution, $\hat{p}(\mathbf{x})$.

Definition 2.2 (Kullback-Leibler distance) The KL divergence between the two probability distributions $p(x; \theta)$ and $\hat{p}(x)$ is given by:

$$D_{KL}(\hat{p}(\mathbf{x}) \parallel p(\mathbf{x}; \theta)) = \mathbb{E}_{\hat{p}} [\ln \hat{p}(\mathbf{x}) - \ln p(\mathbf{x}; \theta)]$$

However, as $\ln \hat{p}(\mathbf{x})$ is not a function of the model, minimizing this KL divergence corresponds exactly to minimizing the *cross-entropy* between the training data generated by $\hat{p}(\mathbf{x})$ and $p(\mathbf{x}; \theta)$.

Definition 2.3 (Cross-entropy) The cross-entropy between the two probability distributions $\hat{p}(x)$ and $p(x; \theta)$ is given by:

$$Q(\theta) = -\mathbb{E}_{\hat{p}} [\ln p(\mathbf{x}; \theta)]$$

where \hat{p} is the empirical distribution of the true probability on the training set \mathbf{x} .

For some problems, such as simple linear regression, computing the maximum likelihood estimator can be easy by using the normal equations. However in practice, it is much more challenging for neural networks. Indeed, there is no analytical solution for their optimal weights and we must search for them using the log-likelihood.

2.1.2 Loss functions

In the previous section, we discussed the specification of the dataset we present to the network and how we can estimate it. We consider the training set $(x_1, y_1), \dots, (x_n, y_n)$ where x_i and y_i are input vectors and target vectors respectively. We estimate the target y by the neural network function $\hat{y} = f(x; \theta)$.

We must now define a *loss function*, $L(x, y, \theta)$ that will measure the quality of the estimation \hat{y}_i for each input x_i in the training set. The most common loss function is the negative log-likelihood, or equivalently the cross-entropy between the network output and the targeted output, because, as we saw, minimizing this loss function brings to maximum likelihood estimation.

The per-example loss function measure the distance between each network output \hat{y}_i and the targeted output y_i , but we need to get a measure of performance over all of the training set. That is why we sum all the elementary loss functions to get the loss function on the whole training set, also called *risk function*:

$$Q(\theta) = -\mathbb{E}_{\mathbf{x}, \hat{y}} L(x, y, \theta) = \frac{1}{n} \sum_{i=1}^n L(x_i, y_i, \theta)$$

where $L(x, y, \theta) = -\ln p(y | x; \theta)$

The choice of loss function is directly related to the type of output units in the neural network, which are themselves depending on the task.

Linear regression

For linear regression, we can simply use a linear perceptron as output. Given the hidden intermediate activations $h = f(x; \theta)$, the output of a linear perceptron will be $\hat{y} = W^\top h + b$. In that case, maximizing the log-likelihood is similar to minimize the *Root Mean Squared Error* (RMSE), defined as follow:

$$RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^n \hat{y}_i - y_i} \quad (2.3)$$

Classification

If we want to represent a K -class classifier, the standard approach is to use K output units each having a *softmax* activation function to obtain the estimations of the class probabilities, $p(y = j | x) = \hat{y}_j$. Because \hat{y}_j is a probability, it needs to be between 0 and 1 and the vector $\hat{y} = (\hat{y}_1, \dots, \hat{y}_K)$ needs to sum to 1. The outputs are then:

$$\hat{y}_j = p(y_j | x; \theta) = \text{softmax}(z_j) = \frac{\exp(z_j)}{\sum_{k=1}^K \exp(z_k)} \quad (2.4)$$

where $z = W^\top h + b$.

It is convenient to use a one-hot-encoded vector for the target class y which is a binary vector with all elements equal to zero except for element k , corresponding to the correct class, which equals one. The target probabilities are then:

$$p(y | x; \theta) = \prod_{k=1}^K \hat{y}_k^{y_k}$$

The loss function is then:

$$Q(\theta) = -\frac{1}{n} \sum_{i=1}^n \sum_{k=1}^K y_{i,k} \ln p(y_{i,k} | x_i; \theta) = -\frac{1}{n} \sum_{i=1}^n \sum_{k=1}^K y_{i,k} \ln \hat{y}_{i,k} \quad (2.5)$$

In the particular case of a binary classifier, the network needs only to predict one class probability, $\hat{y} = P(y = 1 | x)$. In that case, we can use a single output unit with a *sigmoid* activation function and the output is:

$$\hat{y} = \sigma(w^\top h + b)$$

where σ is the *sigmoid* function defined in Equation (1.6). The loss function is then:

$$\begin{aligned}
Q(\theta) &= -\frac{1}{n} \sum_{i=1}^n y_i \ln p(y_i | x_i; \theta) + (1 - y_i) \ln(1 - p(y_i | x_i; \theta)) \\
&= -\frac{1}{n} \sum_{i=1}^n y_i \ln \hat{y}_i + (1 - y_i) \ln(1 - \hat{y}_i)
\end{aligned}$$

2.2 Optimization for neural networks

In the previous sections, we only described how to compute the derivatives of the loss function through the network so it can be trained with gradient descent. However, we need to ensure that training is effective. For that, we need to choose an optimization algorithm for the loss function so we can learn the parameters of the model.

2.2.1 Gradient descent

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

$$\theta = \theta - \eta \nabla Q(\theta)$$

where η is the *learning rate* and $\nabla Q(\theta)$ is the gradient of the cost function with respect to the parameter.

Gradient descent is difficult to use in practice, because it gets easily stuck in a local minimum. Moreover, it can be very slow to converge, because it follows the gradient of the entire training set, technique that is called *batch learning*. That is why most deep learning algorithm use *Stochastic Gradient Descent* (SGD).

Indeed, instead of following the gradient on the whole training set, SGD computes the gradient on *minibatches*, that is small samples of the training set that can be randomly chosen. The size of the minibatches is determined by the *batch size*. As we can see, the batch size controls when to update the network weights during training, it is one of the most important hyper-parameters of the model. Then we estimate the whole gradient with the average of the minibatches gradients, which is an unbiased estimator (Goodfellow et al.; 2016, Chapter 8). We get the following update for a given iteration:

$$\theta = \theta - \eta \nabla \sum_{i=1}^n Q(\theta, x_i, y_i)$$

where $\{x_1, \dots, x_m\}$ are m mini-batch examples of the training set, \mathbf{x} .

SGD is a *mini-batch* learning algorithm, it can also be used as an *online learning* algorithm that updates the weights for every samples presented to the network, where the batch size is set to 1. For large networks with redundant information in the training sets, it is known to be better to use mini-batch learning algorithm. For more information on online learning, see Bottou (1998).

If SGD is very popular, it can sometimes be slow. Qian (1999) proposed a method that accelerates SGD convergence by helping SGD to take the right direction. It introduces the contribution of previous gradients to the current gradient through a variable v and a hyper-parameter γ determining the fraction of that contribution. The update rule for an iteration is then:

$$\begin{aligned}v &= \gamma v + \eta \nabla Q(\theta) \\ \theta &= \theta - v\end{aligned}$$

2.2.2 Adaptive optimization algorithms

SGD is the basis for many other learning algorithms, such as *AdaGrad* or *RMSProp*, the difference being, that these algorithms have an adaptive learning rate. Indeed the *learning rate* is another important hyper-parameter of neural networks because the loss function can be sensitive or not in some directions in the parameter space. For example, the gradient can get stuck in local minima or flat regions.

- *RMSProp* algorithm is a modified version of *AdaGrad* algorithm which goal is to perform better with non-convex function (Hinton; 2012). It changes the gradient, g , by dividing the learning rate, η , by an exponentially decaying average of squared gradients. We have the update learning rule for a given iteration as follows:

$$\theta = \theta - \frac{\eta}{\sqrt{\mathbb{E}[g^2] + \epsilon}} g$$

- *Adam* (Kingma and Ba; 2015) is another adaptive algorithm and is nowadays one of the most used optimization algorithm. It is a combination of RMSProp and momentum SGD algorithms. We have for given parameters value β_1 , β_2 , e :

$$\begin{aligned}m &= \beta_1 m + (1 - \beta_1)g \\ v &= \beta_2 v + (1 - \beta_2)g^2\end{aligned}$$

where m and v are estimates of the first moment (mean) and the second moment vectors of the gradient, g . These estimations are biased, so the authors compute a bias-correction at time step t :

$$\begin{aligned}\hat{m} &= \frac{m}{1 - \beta_1^t} \\ \hat{v} &= \frac{v}{1 - \beta_2^t}\end{aligned}$$

Finally, we can formulate the update rule for a given iteration:

$$\theta = \theta - \frac{\eta}{\sqrt{\hat{v} + e}} \hat{m}$$

2.2.3 Batch normalization

Batch normalization is an optimization strategy that is not an algorithm. It is a method of adaptive reparametrization developed by Ioffe and Szegedy (2015) to make training of very deep models easier. Indeed, training deep neural networks is complicated by the fact that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change. Ioffe and Szegedy (2015) refer to this phenomenon as *internal covariate shift*. To reduce it, they normalize the layers' inputs with a *Batch Normalization* layer. While the network is trained, the method performs normalization for each training mini-batch allowing higher learning rates. Batch Normalization also acts as a regularizer preventing the need of L^1 or L^2 penalties to the loss function (Goodfellow et al.; 2016).

Let us define the Batch Normalization transformation as in Ioffe and Szegedy (2015).

Definition 2.4 (Batch Normalization layer) Consider a mini-batch of size m , $\mathcal{B} = \{x_1, \dots, x_m\}$ with x the values over the mini-batch \mathcal{B} . Let $\mu_{\mathcal{B}} = \frac{1}{m} \sum_{i=1}^m x_i$ be the mini-batch mean and $\sigma_{\mathcal{B}}^2 = \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2$ the mini-batch variance. Let $\hat{x}_{1,\dots,m}$ be the normalized values of x , $\hat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$, where ϵ is a constant. Finally, let $y_{1,\dots,m}$ be their linear transformation. We have:

$$BN_{\gamma,\beta}(x_i) = y_i = \gamma \hat{x}_i + \beta$$

where γ and β have to be learned.

Now that we know how to train neural networks, we need to explain how can we measure that the learning is indeed effective.

2.3 Generalization methods

The ability of a model to perform well on previously unobserved inputs is called *generalization*. The challenge of machine learning model is to have a good generalization power, which indicates that the model can perform well on *new* inputs. In this section, we present some problems that deep neural networks can face for generalization and some solutions.

2.3.1 Model selection

To establish the superiority of a learning machine over the other, we must make data-generating assumptions, $\hat{y} = f(x; \theta)$, and know the optimal parameter θ , under these assumptions. However, it is difficult to know in advance which learning machine to select. To find the best model corresponding to our data, defined by the parameters θ , we need to establish a clear and reproducible procedure based on some evaluation criteria.

Holdout selection procedure

The holdout selection procedure consists in partitioning the available data in three sets, a training set, a validation set and a test set. We use the training set to consider different learning machines (MLP, RNN and LSTM with different parameters) and we apply these models to the validation set. We select the model which has the best performance on the validation set and apply this model on the test set in order to measure the *generalization error*. The generalization error evaluates the prediction power of a model on unknown data, here the test set, which was not used for the training of the model.

The generalization error is indeed essential because of the *overfitting* phenomenon described in the next paragraph.

Underfitting and overfitting problems

Underfitting and *overfitting* problems are the main challenge in machine learning.

- *Underfitting* occurs when the model cannot learn the underlying structure of the data, which implies a large training error. Underfitting phenomenon can appear when the complexity of the model, defined in the next paragraph, is too small.
- *Overfitting* occurs when the model learns too exactly the patterns in the training data and cannot generalize these patterns for prediction. In other terms, overfitting happens when the trained model learns also the noise in the sample. Thus, the model will have a good performance on the train set, but not on the test set. Overfitting implies a large gap between training and test error and can appear when the complexity of the model is too large.

Model selection

To select the best model, we use the generalization error on the validation set and we select the one with the lowest error with respect to the first Occam's Razor from Domingos (1999): "Given two models with the same generalization error, the simpler one should be preferred because simplicity is desirable in itself."

We have different possible measures of complexity such as the number of parameters of the model, the choice of variables the model receives as input, the properties of the function (continuity, slope), the VC-dimension that provides a bound on generalization error or the number of training iterations. Overfitting phenomenon is essentially related to the complexity of the model.

Generalization error measure

How can we measure the generalization error? The statistical learning theory gives the famous *bias-variance decomposition*, which is defined as follows:

$$\text{Error}(f_{\hat{\theta}}) = \text{Bias}(f_{\hat{\theta}}) + \text{Variance}(f_{\hat{\theta}})$$

where $\hat{\theta}$ is a random variable. From this equation, we can see that bias and variance contribute in different proportions to the error depending on the complexity. From this, we are invited to use while training the model, the famous method of *early stopping* developed by Yao et al. (2007), which we described in Section 2.3.2. Yao et al. (2007) show that the number of training iteration is connected to the overfitting phenomena. Stopping training too early may reduce variance, but enlarge bias; and stopping too late may enlarge variance though reducing the bias. Thus solving this bias-variance trade-off leads to the early stopping rule.

Metric

For the model evaluation, we need to look at the generalization power of the model, which is given by the scalar loss, but we need also an evaluation metric which is a performance measure for a given goal. The metric used will depend on the task. For a classification task, the usual metric is the *accuracy* of the model, which computes the fraction of correct predictions. The accuracy is given by the formula:

$$\text{Accuracy}(y, \hat{y}) = \frac{1}{n} \sum_{i=0}^{n-1} \mathbb{1}_{\{\hat{y}_i = y_i\}} \quad (2.6)$$

However, the accuracy does not reflect imbalanced class, which is a problem in our application in the next chapter. Indeed, let us consider a binary classifier on 100 data points where 2 points correspond to class 0 and 98 points correspond to class 1. If the classifier predict 2 points in class 0 and 98 points in class 1, it is a *perfect* classifier which achieve optimal performance with an accuracy of 100%. However, if the classifier predict 100 points in class 1, the accuracy is then 98% and the classifier looks as if it achieves a near optimal performance, but in fact, it fails to predict any points in class 0. It is a *naive* classifier. Thus, we need to use an alternative performance measure with higher robustness to class skew, for example the *F-measure* or *F1-score*.

For that, we introduce the notion of *precision*, a measure of exactness, which in a binary classification task is the fraction of correctly classified positives among all examples classified as positive. Intuitively, it is the ability of a classifier not to label as positive a sample that is negative. We also introduce the notion of *recall*, a measure of completeness, which in a binary classification task is the fraction of correctly classified positives among all positives. Intuitively, it is the ability of a classifier to find all the positive samples.

Let us introduce some notation. TP_k (True positives) is the number of points assigned correctly to class k ; FP_k (Falses positives), the number of points that do not belong to class k

but are assigned to class k incorrectly by the classifier; and FN_k (False Negatives) is the number of points that are not assigned to class k by the classifier, but actually belong to class k .

The recall, ρ_k , and precision, π_k , measure for class k are then (Özgür et al.; 2005):

$$\pi_k = \frac{TP_k}{TP_k + FP_k}, \quad \rho_k = \frac{TP_k}{TP_k + FN_k} \quad (2.7)$$

We take the *macro-averaged F-measure*, which is computed locally over each category first and then the average over all categories is taken. We have:

$$F_k = \frac{\pi_k \rho_k}{\pi_k + \rho_k}, \quad F_{macro} = \frac{1}{K} \sum_{k=1}^K F_k$$

The F_{macro} does not take into account imbalanced class so it is influenced by the classifier performance on rare categories. That is why we use a weighted F-measure, which calculates metrics for each class and find their average weighted by support (the number of true instances for each label). This alternative $F_{weighted}$ takes into account label imbalance.

$$F_{weighted} = \frac{1}{K} \sum_{k=1}^K w_k F_k$$

All the latter introduced metrics gives some performance measures in order to compute the generalization error of a model. A model with a low generalization error should have a low loss and a good metric on both train and test sets.

2.3.2 Regularization techniques

Early stopping

We can see early stopping as a very efficient hyper-parameter selection algorithm, where the number of training steps (epochs) is just another hyper-parameter (Goodfellow et al.; 2016). Early stopping is a method that stops training when a monitored quantity has stopped improving. Indeed, deep neural networks tend to overfit the data. Even if we often observe a decreasing training loss, sometimes the test set error begins to increase after some training steps, which indicates overfitting. In that case, we will have a better generalization power if we stop training before the test set error increases.

With early stopping method, we monitor for example the loss or the metric on the validation set after each training step and we stop training if the loss starts to increase or if the accuracy starts to decrease (in the case that the metric used is the accuracy).

Early stopping is one of the most used regularization technique in deep learning, because of its simplicity and its effectiveness on reducing training time.

Dropout

The most common way to avoid overfitting in deep neural networks is to use a *dropout* layer as proposed Srivastava et al. (2014). The key idea is to randomly drop units (along with their connections) from the neural network during training, by forcing the weights of the units to be equal to 0, which reduces the number of parameters in the model.

Dropout can also be thought as an effective bagging method for many large neural networks (Goodfellow et al.; 2016). Bagging evolves training multiple models and evaluating multiple models on each test example. With the dropout method, we train an ensemble of all sub-networks that can be constructed by removing some input units from an underlying base network.

Now that we explained the different architectures of neural networks and that we have a reproducible and objective method to train and select the best models corresponding to a specific task, let us give examples of applications of deep neural networks on financial data.

Chapter 3

Neural networks for time series forecasting

3.1 Cryptocurrencies as financial assets

The first cryptocurrency, Bitcoin, was created in 2008 by Satoshi Nakamoto. With Bitcoin, he invented the first unregulated digital currency designed to work as a medium of exchange thanks to the *Blockchain* technology which is a distributed ledger based on a decentralized peer-to-peer network that confirm transactions. Ten years later, 1512 alternative cryptocurrencies, called *altcoins*, are identified on CoinMarketCap proving that a real cryptocurrency market has emerged. Indeed, the cryptocurrency market is experiencing a strong growth over the last years, which can be inferred from CRIX, developed by Trimborn and Härdle (2016).

Because the economy is becoming more and more digital, it is natural to think that the role of digital assets, such as cryptocurrencies, in investment decisions will also grow. Indeed, investors from the former existing financial markets are now interested in cryptocurrencies, as new financial products on the cryptocurrency market are created with the apparition of options and futures on Bitcoin. Eisl et al. (2015) analyzed Bitcoins returns to successfully show that adding Bitcoin to portfolios can have an optimal diversification effect. Elendner et al. (2017) proved that cryptocurrencies are interesting for investors due to the diversification effect, because they are uncorrelated with each other and uncorrelated with traditional asset. Finally, Trimborn et al. (2017) showed that mixing cryptocurrencies with stocks could improve the risk-return trade-off of portfolio formation.

As in Elendner et al. (2017), we will investigate cryptocurrencies as alternative investment assets by studying their returns. In the following section, we will focus on the application of the different models we presented in Chapter 1 for time series modeling, especially cryptocurrencies prices.

3.2 Stock prices and underlying assumptions

3.2.1 Time series modelization

Numerous time series present an autoregressive structure where past observation can be used to explain the present and future observations as in Equation 3.1. This autoregressive structure is unknown and numerous model with different assumptions try to represent this structure.

$$S_{t+1} = \phi(S_t, S_{t-1}, \dots, S_{t-p}) \quad p \in \mathbb{N} \quad (3.1)$$

The most common linear models are *ARIMA* or *SARIMA* models, which are simple models and explain major aspects of the structure of time series such as the autoregressive process itself, the moving average or the seasonality. These models have a considerable explanation power of

the mechanism behind time series, nevertheless finding time series that can be modeled by them is challenging. Indeed, *ARIMA* models have strong assumptions that in practice are very hard to meet, such as constant parameters, white noise residuals or homoscedasticity. For example, stock prices often present heteroscedastic residuals. To face these problems, econometricians built a wide variety of different models such as ARCH or GARCH models.

However, if we can find a model that explain the structure of stock prices, it is questionable that it can actually predict future values, especially in long-term horizon. In fact, the random walk theory for stock prices implies that any predictive model for stock prices is useless.

Stock prices as time series

Stock price forecasting is one of the most important task of quantitative finance. It is particularly challenging because of the properties of stock prices, which does not behave as simple time series. Indeed, the random walk theory suggest that stock price returns are independently and identically distributed over time so the past values of a stock returns cannot explain their future values. The direct result is that the best prediction for tomorrow's price is the price of today. This theory led to the *Efficient Market Hypothesis* (EMH) which asserts that the price of stocks reflects all relevant information available, implying that investor cannot outperform the market with a trading strategy, based on decisions made from the available information.

To be short *fundamental analysis* which tries to evaluate the intrinsic value of a security, would be irrelevant for financial market if the EMH is true. Moreover, *technical analysis* would be also irrelevant. Opposed to fundamental analysis, technical analysis aims to forecast the future price movements based on statistics, such as technical market indicators (moving average, Bollinger intervals) or underlying variables such as price movement, volume, market capitalization, global economic variables.

Technical analysis forbid us to use linear autoregressive models such as *ARIMA* for price movements' modelization, because *ARIMA* models use only the past observation of the time series itself as regressors and supposed that the relation is linear.

In the past years, the Efficient Market Hypothesis has weakened its credibility among economics, specially with the appearance of Behavioral Economics that revitalized Fundamental and Technical analysis

For this thesis, we use both Technical and Fundamental analysis and we supposed that stock prices is represented by a *Non-Linear AutoRegressive with exogenous variables* model (NLARX).

3.2.2 NLARX model

Let us first define the NLARX model.

Definition 3.1 (Non-Linear Autoregressive with exogenous variables model) Let $\{S_t\}_{1 \leq t \leq T}$ be real values of a time series, $\{X_t = (X_{1,t}, \dots, X_{d,t})\}_{1 \leq t \leq T}$ the d -vectors of exogenous variables and ε_t independent real random variables. A *NLARX*(p, x, q), $(p, x, q)^3 \in \mathbb{N}$, process represent a mapping $f^* : \mathbb{R}^{p+xd} \rightarrow \mathbb{R}^q$ defined as follows:

$$(S_{t+q}, S_{t+q-1}, \dots, S_{t+1}) = f^*(S_t, S_{t-1}, \dots, S_{t-q+1}, X_t, X_{t-1}, \dots, X_{t-x}) + \varepsilon_t$$

For example if $p = 3$, $q = 1$ and $x = 0$ we obtain the following NLARX(3,0,1) model:

$$S_{t+1} = f^*(S_t, S_{t-1}, S_{t-2}, X_t) + \varepsilon_t$$

The mapping of a *NLARX*(p, x, q) can be represented by neural network function $f : \mathbb{R}^{p+xd} \rightarrow \mathbb{R}^q$ with parameters θ defined as follows:

$$(S_{t+q}, S_{t+q-1}, \dots, S_{t+1}) = f(S_t, S_{t-1}, \dots, S_{t-q+1}, X_t, X_{t-1}, \dots, X_{t-x}; \theta) + \varepsilon_t$$

The structure of the network depends on its nature. For example, a MLP network would have $p + xd$ input neurons and q output neurons.

The architecture of such network can be challenging to represent because of different time lags in the autoregressive part of the model p and the exogenous part x .

As we can see from its definition, a NLARX process gives a great liberty in the forecasting procedure. Indeed, we distinguish two main procedures to predict the time window of length q of a time series $S_{t1 \leq t \leq T}$

- **Recursive strategy:** We assume that S_t can be represented by a NLARX(p,1) network, we have:

$$\begin{aligned}\hat{S}_{t+1} &= f(S_t, \dots, S_{t-p+1}, S_t; \theta) + \varepsilon_t \\ \hat{S}_{t+2} &= f(\hat{S}_{t+1}, \dots, S_{t-p+2}, X_t; \theta) + \varepsilon_t \\ &\dots \\ \hat{S}_{t+q} &= f(\hat{S}_{t+q-1}, \dots, S_{t-p+q}, X_t; \theta) + \varepsilon_t\end{aligned}$$

This strategy has the advantage to be parsimonious in the number of parameters to estimate, thus the network has minimum complexity. However, the crucial weakness is that it accumulate the estimation error at each forecasting step. Thus, if forecast window, q , is large, the performance can quickly degrade and the estimation of the price q -step ahead can be really unreliable. Moreover, if this model represent the relations between the past values (S_t, \dots, S_{t-p+1}) , it does not learn the structure between the output values $(S_{t+1}, \dots, S_{t+q})$. In sequence learning theory, it is a *many-to-one* model that takes as input a sequence and produce a scalar.

Nevertheless, the recursive strategy can be applied for small forecast window. Indeed, a smaller forecast window might need a smaller look-back window (q), which leads to less parameters to estimate and a simpler model, which is easier and faster to train.

- **Multiple output strategy:** We assume that S_t can be represented by a NLARX(p,q) network, we have:

$$(\hat{S}_{t+q}, \dots, \hat{S}_{t+1}) = f(S_t, \dots, S_{t-p+1+q}, X_t; \theta) + \varepsilon_t$$

This model is capable of predicting the entire forecast window in one operation, preventing the error forecast to increase as the forecast window grows. This model is more complex as it needs more parameters which allow us to learn the dependence between inputs and outputs as well as between outputs: it is a *many-to-many* sequence model. Nevertheless, this complexity implies costs from a longer training time and from a larger amount of needed data to avoid overfitting problem.

3.3 Cryptocurrencies price movement analyze

3.3.1 Data analysis

Selection of important cryptocurrencies based on market capitalization

Thanks to CRIX dataset (<http://crix.hu-berlin.de/>), we have at our disposal daily data of 631 cryptocurrencies over more than 3 years, from 2014-07-31 to 2017-10-25. Since the cryptocurrencies market is in full evolution, a lot of daily price are missing. To reduce our analysis to

the most important cryptocurrencies, we selected the 8 cryptocurrencies with the largest market capitalization over the period and without any missing values to avoid missing data imputation problems. At the time of the start of our study (February 2017), the cryptocurrencies were Bitcoin (btc), Dash (dash), Ripple (xrp), Monero (xmr), Litecoin (ltc), Dogecoin (doge), Nxt (nxt) and Namecoin (nmc) from which we have data from 2014-07-31 to 2017-10-25.

	nmc	nxt	doge	xmr	dash	ltc	xrp	btc
min	2.5	5.3	8.6	0.5	5.3	40.9	40.8	2362.6
max	15.2	47.6	46.2	248.9	337.7	311.3	862.0	20884.9
mean	5.9	13.0	19.9	32.2	37.0	146.2	255.4	6898.0
median	5.4	9.4	21.2	5.4	20.1	150.7	238.6	6037.8
standard deviation	2.3	8.2	5.9	53.0	35.9	47.9	115.3	3700.9

Table 3.1: Market capitalization statistics (in Millions Dollars) of the 8 most import cryptocurrencies from 2014-07-31 to 2017-10-25

From Table 3.1, we can observe that btc dominates the market with a market capitalization 30 times bigger in average than xrp, the second most important cryptocurrency, and 1000 times larger than nmc, the smallest cryptocurrency considered.

The structure of the cryptocurrency market is changing everyday and at the time of writing this paper (January 2018) the 8 cryptocurrencies with the largest market capitalization are different but the one we selected are still important.

Indeed if we only consider the cryptocurrencies for which we have daily prices at our disposal until 2014-07-31, the cryptocurrencies are still in the top 13 cryptocurrencies with the largest market capitalization in average over the period considered. We can say that the 8 cryptocurrencies are dominating the market of old cryptocurrencies in terms of market capitalization.

If we consider also recent cryptocurrencies, the market structure is different and this result doesn't hold as we can see in Table 3.2

	nmc	nxt	doge	xmr	dash	ltc	xrp	btc
position	112	93	68	26	17	12	4	1

Table 3.2: Market capitalization importance in the 631 cryptocurrencies, average on the period

Nevertheless, btc is still dominating the market over the period and xrp, ltc and dash are major cryptocurrencies in terms of market capitalization. Over the last months, from March to October 2017, we can say that eth, bch, neo, xem, miota and etc are the major new cryptocurrencies ranking in the top 10 of cryptocurrencies with the largest market capitalization in average from March to October 2017. It would be interesting to add them in a future analysis.

3.3.2 Application: examples with btc returns

In this section, we will show the different step to build a reproducible method for time series forecasting with neural networks.

Let us imagine a forecast problem where we want to predict btc price two days in the future. We can apply a NLARX(5, 0, 2) model to the btc price using technical and fundamental indicators. We do not discuss in this part the choice of the parameters p , q , x . Let P_t be btc price at time t , we estimate the NLARX(5, 0, 2) process with the following MLP function

$$(P_{t+2}, P_{t+1}) = f(P_t, P_{t-1}, \dots, P_{t-4}, X_t; \theta) + \varepsilon_t$$

where X_t is the 16-dimensional vector of exogenous variables which are:

- suitable transformation of the daily returns such as the 14 days, 30 days and 90 days moving averages and their respective Bollinger bands which are defined in Section 4.1.2
- the CRIX daily returns at time t
- Euribor interest rates at different horizon (year, 6 months and 3 months)
- Different exchange rates (EURO/UK, EURO/USD, US/JPY)
- Because cryptocurrencies price is available everyday, we replace the missing values on the weekend with the last value observed at the closing of the exchange on Friday

Before, training the neural network, we need to preprocess the data. Indeed, input representation is very important for neural network. Because the price is not stationary, as we saw in the previous section, we consider the logarithm of the differenced prices, also called *log-returns*.

- We take the differenced logarithm of the price series to eliminate trend and seasonality. We get the logarithm return of holding btc for each period considered, here one day:

$$R_t = \ln\left(\frac{P_{t+1}}{P_t}\right)$$

- We standardized the input variables to avoid any scaling problem: $J_t = \frac{I_t - \mu}{\sigma}$ where μ and σ are respectively the mean and the standard deviation of one input variable I_t on the training set. We will get standardized predictions on the test set so we will use the mean and the standard deviation of the training set to get the predictions in their original scale.

We then estimate the NLARX(5,0,2) with the following MLP funtion:

$$(R_{t+2}^{scaled}, R_{t+1}^{scaled}) = f(R_t^{scaled}, R_{t-1}^{scaled}, \dots, R_{t-4}^{scaled}, X_t^{scaled}; \theta) + \varepsilon_t$$

where R_t^{scaled} , X_t^{scaled} represent the scaled daily returns of btc and the scaled exogenous variables respectively. This network will give us prediction for btc returns. However, we want to predict btc returns in order to make simple trading decisions (buy btc if the price is expected to grow and sell btc if the price is expected to fall), thus the actual value of the price is irrelevant. We are only interested in the future price movement. We can then reformulate the problem as a binary classification of the future trend, that is:

$$T_{t,k} = \begin{cases} 0, & \text{if } \ln\left(\frac{P_{t+k}}{P_t}\right) \leq 0 \\ 1, & \text{if } \ln\left(\frac{P_{t+k}}{P_t}\right) > 0 \end{cases}$$

We can now build the MLP network corresponding to our task by the function:

$$T_{t,2} = f(R_t^{scaled}, R_{t-1}^{scaled}, \dots, R_{t-4}^{scaled}, X_t^{scaled}; \theta) + \varepsilon_t$$

In this thesis, for the implementation of the different models, we use *Keras Python* library with *Tensorflow* as backend. We use the data up to 2017-03-01 for the train set and the data from 2017-03-02 to 2017-10-25 for the test set.

We will represent f with different MLP architectures. In terms of trading strategy, the performance measure of the network should be the accuracy of the model (see Equation 2.6), since it measures the number of correct predictions.

We use only tanh as activation function for the hidden layers. The final output layer has two neurons, corresponding to the number of classes of the target variables with a softmax activation functions in order to get the probability classes. We first build a baseline model, with two hidden layers with 5 neurons each. We trained the network with RMSProp algorithm (see Section 2.2.2) on 50 epochs with a default batch size of 32 and a default learning rate of 0.001. We evaluate our baseline model with a 5-fold cross validation (see next paragraph) on the train set. We obtain an accuracy score of 54 % with a variance of 0.06.

Comparison of different MLP architectures: hyper-parameter tuning

Model tuning is an important step of the model construction. It consists of building various models that have different hyper-parameters values to find the one that best fits the data using an evaluation criterion. Each model has different hyper-parameters, here we focus on the architecture of the neural networks (width and depth).

To select the best model, we tune the number of neurons and layers with *scikit-learn* python library. In order to realize this tuning with respect to the hold-out procedure, we realize a *k-fold cross validation* on the train set. Let us explain briefly this method.

We split the train set in k subsamples chronologically ordered (S_1, S_2, \dots, S_k) , we train the model on the first k samples and test it on the last sample, we repeat this operation k times. That is we first train the model on the first sample S_1 and validate it on S_2 . We then retrain the model on $S_1 \cup S_2$ and test it on S_3 . We repeat this operation until we train the model on $S_1 \cup S_2 \cup \dots \cup S_{k-1}$ and test it on S_k . The final score is then an average of k -scores. We select the model that has the best average score. Finally, the score on the test set will give us the generalization error.


For our example, we realize a 5-fold cross validation on the train set. The model with the best average accuracy is the final model selected. As a grid search, we tested MLP architectures with two, three, four and ten hidden layers.

Hidden layers	Neurons per layers
2	(5, 10, 15, 20, 25, 50, 100)
	(5, 10, 15, 20, 25, 50, 100)
3	(5, 10, 15, 20, 25, 50)
	(5, 10, 15, 20, 25, 50)
	(5, 10, 15, 20, 25, 50)
4	Best step 3
	(5, 8, 10, 12, 15, 18, 20, 25, 50)
10	Best step 4
	(5 per layer)

Table 3.3: Grid search for hyper-parameter tuning

We first realize this tuning with a maximum number of 50 epochs and then 500 epochs. In the following table, we present the accuracy for each final model.

Step	Epochs	Hidden layers	Neurons per layers	Accuracy (%)
Model 1	50	2	(100, 15)	55
Model 2	50	3	(50, 15, 15)	56
Model 3	50	4	(50, 15, 15, 5)	56
Model 4	50	10	(50, 15, 15, 5, 5, 5, 5, 5, 5, 5)	53
Model 5	500	2	(50, 100)	56
Model 6	500	3	(25, 10, 25)	58
Model 7	500	4	(25, 10, 25, 12)	55
Model 8	500	10	(25, 10, 25, 12, 5, 5, 5, 5, 5, 5)	53

Table 3.4: Grid search for hyper-parameter tuning  BTCtuning

From this table, we can see that all architectures have an equivalent performance. Especially adding layers does not necessarily improve the accuracy of the model. In particular, very deep MLP architectures (model 4 and model 8) have a lower performance than the baseline model. Nevertheless, architectures with three or four hidden layers perform at least as good as model 1 which has only two hidden layers, indicating that adding a few layer to the baseline architecture

can improve the generalization power. Finally, model 6 is the best model with a cross validation score of 58%. The final measure of generalization error is obtain by computing the accuracy on the test set that is unseen from the model. We obtain a final accuracy score of 51%.

Chapter 4

Application: Cryptocurrency portfolio

This application is inspired by the pilot study that was carried out in cooperation with Commerzbank AG, Franke (1999). The goal was to develop a trading strategy for a portfolio made up of 28 of the most important stocks from the Dutch CBS-Index based on predicted returns only. We transposed this strategy to a portfolio made up of the 8 cryptocurrencies that dominates CRIX, Trimborn and Härdle (2016) from 2014 to early 2017. As in Franke's study we restrict ourselves to the buy-and-hold strategy with a time horizon of a quarter of a year (90 trading days because cryptocurrency exchanges are open on weekends). The portfolio is created at the beginning of a quarter and then held for three months without any alteration. We include the cryptocurrencies whose prices are predicted to rise significantly, hold them up to the end of the end of the quarter and sell them afterwards. At the end of the three months, the value of the portfolio should be as large as possible. As a basis for the trading strategy a three months forecast of the cryptocurrencies is used.

To modelize the time series $S_{i,t}$, we use a NLARX process (see Definition 3.1) where $S_{i,t}$ represents the price of cryptocurrency i (see Table 4.1). We have to build one model for each cryptocurrency in order to predict the price three months ahead. Let us first present the input data we use for each model.

4.1 Data analysis

4.1.1 Cryptocurrencies price

The prices of the height most important cryptocurrencies have different distributions but are following the same trend. btc is clearly dominating the cryptocurrency market with an average price of almost 1000 USD over the last three years, 25 times larger than dash, the second cryptocurrency in value and over than 20 000 times larger than xrp, the cryptocurrency with the second largest market capitalization, see Table 4.1.

The cryptocurrency market experienced an incredible growth in the last year as we can see from the evolution of the CRIX price on Figure 4.1.

As a result, we can consider the price evolution of cryptocurrencies within two different periods, from July 2014 to the first months of 2017 where each price of the different cryptocurrencies was almost constant closed to its minimum, and from the beginning of the year 2017 to October 2017, where each price experienced a rapid growth until it reached a pick at the end of the summer and start to decline, see Figure A.1 in the appendix. The fact that the price range changes over time for each cryptocurrency can be a problem for its modelization, as we will see in the next section, we need to preprocess the data.

	btc	dash	xrp	xmr	ltc	doge	nxt	nmc
Min	172	1.06	0.00409	0.216	1.15	8.69e-05	0.00531	0.169
1st Quartile	296	2.75	0.00607	0.508	3.16	0.000155	0.00758	0.351
Median	452	5.79	0.00764	1.17	3.83	0.000219	0.0115	0.443
Mean	917	41.6	0.0405	12.7	9.97	0.000454	0.0257	0.703
3rd Quartile	894	12.8	0.0133	12	4.61	0.00027	0.0244	0.89
Max	6040	569	0.405	142	85.7	0.00383	0.195	3.66

Table 4.1: Cryptocurrencies prices statistics



Figure 4.1: CRIX price

4.1.2 Technical indicators

As we saw in Section 3.2.1, technical indicators can be useful variables for time series modeling. In the forecast of cryptocurrencies prices, we decided to use three major technical indicators: Bollinger bands at different time horizon, two weeks, one month and three months, so 14, 30 and 90 days which corresponds to 10, 20 and 60 days in the traditional market.

Let $S_{i,t}$ be the price of stock i at time t . We define the simple moving average of order q as:

$$\mu = \frac{1}{q} \sum_{t=1}^q S_{i,t}$$

and the rolling standard deviation of order q as:

$$\sigma = \sqrt{\frac{1}{q} \sum_{t=1}^q (S_{i,t} - \mu)^2}$$

The three components of Bollinger bands corresponds to μ , the central band, $\mu + \delta\sigma$, the upper band and $\mu - \delta\sigma$ the lower band, where δ is a fixed parameter historically equals to 2. The width of the band is a direct indicator for volatility of a stock. For example, on Figure 4.2 we can observe that Bitcoin price volatility increased as the price grew during the first months of 2017.

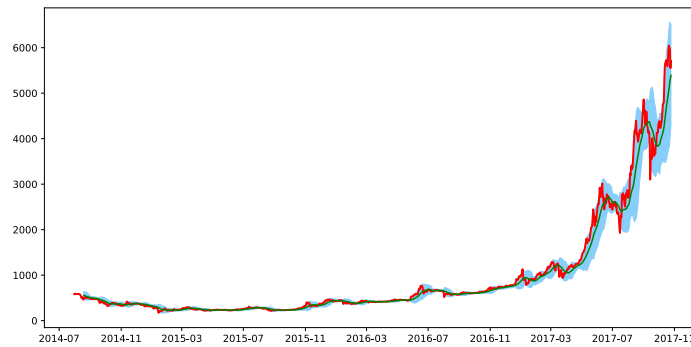



Figure 4.2: Bitcoin prices (red), its 20-days moving average (green) and its lower and upper Bollinger bands (light blue)  CRIXbtcBB

For each cryptocurrency model, we consider Bollinger bands of the cryptocurrency price on three different horizons to reflect the long-term trend directions of the price and its volatility.

4.1.3 Fundamental indicators

To reflect the intrinsic value of prices, we chose to include also some fundamental indicators as exogenous variables. As in Franke's study, we use international interest rates such as Euribor at different horizons and because our portfolio is made of cryptocurrencies, we use different exchange rates.

To be able to catch the inner relations of the cryptocurrency market, we also use the value of the other cryptocurrencies considered ($\{S_j\}, 1 \leq i \leq 8, j \neq i$) and the cryptocurrency index, CRIX.

In Table 4.2 we represent all the variables we use as input to modelize the cryptocurrency price S_i .

Technical indicators	Bollinger Bands 14 days, 30 days and 90 days of crypto i
Fundamental indicators	CRIX, Euribor 3 months, 6 months, 1 year Open, low, high and close price of exchanges UK/Euro, Euro/USD, JPY/USD $\{S_j\}, 1 \leq j \leq 8, j \neq i$

Table 4.2: Input variables for each model

4.2 Model architecture

Our goal is to build a portfolio with a maximum three months return, but forecast points of time series over such a large lag are notoriously unreliable, Franke (1999) and our goal is not to find a good forecast of the prices of the cryptocurrencies themselves, but to maximize the return of the cryptocurrency portfolio. That is why we do not build a direct forecast strategy with a NLARX(p,x,90) model, but we rather predict the prices movements than the actual prices. Thus, we try only to predict the trend correctly, that is if the cryptocurrency price will increase significantly (more than 5%), decrease significantly (less than 5%) or stay at the same level (does not change by more than 5%).

Let us consider the price of cryptocurrency i , $S_{i,t}$ at time t and let us assume that $S_{i,t}$ is an NLARX(p,q,x). We estimate the function f^* from Definition 3.1 with different neural network architectures, f .

$$(\hat{S}_{t+90}, \dots, \hat{S}_{t+1}) = f(S_t, \dots, S_{t-89}, X_t, \dots, X_{t-89}; \theta) + \varepsilon_t$$

4.2.1 Data preprocessing

- **Log-returns transformation:** Before feeding the inputs variables from 4.1 into the network we need to preprocess the data. Indeed, as we saw that cryptocurrencies prices are not easy to handle we uses log-returns. Moreover, because we are only interesting in the forward movement of the price we use k -returns (multiple step returns) rather than daily returns. We define the k -log-returns as follows:

$$R_t(k) = \ln \frac{S_{t+k}}{S_t}$$

- **Sequence learning:**

Because we are dealing with time series, we are going to use a sequence learning algorithm which learns the inner relations of sequences as well as relations between them. That is, we want to be able to predict if the price of a cryptocurrency will rise within a sequence but also if it will rise after a sequence as been fed to the network, thus we use a *many-to-many* sequence model.

Input variables: As inputs, we use sequences of past values of multiple log-returns of the cryptocurrencies and past values of the exogenous variables, which we can write $(I_{t-90}, I_{t-89}, \dots, I_{t-1})^\top$. It is a matrix of dimension $(90, 24)$:

$$\begin{pmatrix} R_{1,t}(-90) & \dots & R_{8,t}(-90) & X_{t-90} \\ R_{1,t}(-89) & \dots & R_{8,t}(-89) & X_{t-89} \\ \dots & \dots & \dots & \dots \\ R_{1,t}(-1) & \dots & R_{8,t}(-1) & X_{t-1} \end{pmatrix}$$

where $R_{i,t}(k)$ is the k -log-returns of cryptocurrency i at time t and X_t is the vector of exogenous variables (technical indicators of the cryptocurrency and fundamental indicators) at time t . For each cryptocurrency model, we only include the Bollinger bands of the cryptocurrency considered in order to reduce the complexity of the model and avoid overfitting problems. Thus, the height models share all input variables except for the Bollinger bands.

Output variables: The output variables depends both on the objective we fixed and the activation function at the output layer.

As outputs, we use the sequence of trend classifications in the future. We need to code the k -log-returns as three dimensional variables that reflects the price movements accordingly to our objective.

$$(T_i(1), \dots, T_i(90)) \in \{0, 1, 2\}^{90}$$

where, we have for $k \in 1, \dots, 90$:

$$T_i(k) = \begin{cases} 1, & \text{if } \frac{S_{t+k}-S_t}{S_t} \leq -0.05 \\ 2, & \text{if } 0.05 \leq \frac{S_{t+k}-S_t}{S_t} \\ 0 & \text{otherwise} \end{cases}$$

We can now write f as a sequence to sequence learning problem as it follows:

$$(T_i(1), \dots, T_i(90)) = f \left((I_{t-90}, I_{t-89}, \dots, I_{t-1})^\top \right) + \varepsilon_t$$

- **Scaling:** Finally, because neural networks uses activation functions that squash the input variables to their output interval, we need to scale the input data so the neurons learn faster, see LeCun et al. (1998). The scaler used depends on the activation function. Because we use the default LSTM activation functions, tanh, we scale the input variables to $[-1, 1]$.

4.2.2 Baseline model

Architecture

The general architecture of the model is constituted of an input layer of dimension (time steps \times number of features = 90×24), the inner structure and an output layer of dimension (time steps \times number of classes = 90×3). Indeed, we apply at each future time step a *softmax* layer with three neurons, one for each class of the output variables. We will change the inner structure of the model to test different architectures of neural networks.

Loss function

Since our problem is a classification task, we use the cross-entropy as loss function. But, as we saw on Figure 4.1, the market is quite stable at the beginning of the period and experienced a rapid growth at the end, so we can expect to have unbalanced classes in the dataset. While training we will have to balance the loss function by the class weights. From Table 4.3, we can clearly see that, except for ltc, all cryptocurrencies have unbalanced classes.

Cryptocurrency	0	1	2
btc	0.26	0.48	0.26
dash	0.18	0.51	0.31
xrp	0.19	0.33	0.48
xmr	0.13	0.53	0.35
ltc	0.32	0.34	0.34
doge	0.29	0.31	0.40
nxt	0.16	0.29	0.56
nmc	0.22	0.24	0.54

Table 4.3: Class repartition

Thus we need to use a modified version of the cross-entropy defined in Equation 2.5 and use the weighted cross-entropy:

$$Q(\theta) = -\frac{1}{n} \sum_{i=1}^n \sum_{k=0}^2 w_k y_{i,k} \ln \hat{y}_{i,k}$$

where $w_k = \frac{n}{K * \text{frequency of class } k}$ are the class weights.

Metric

As performance metric for model selection, we use the accuracy and the weighted F-measure introduce in Section 2.3.1.

Baseline models performance

Let us first present three different baseline models with two hidden layers with ten neurons each corresponding to three type of neurons, LSTM, simple RNN and perceptron. To reduce the training time, we used the well-known method of *early stopping* defined in Section 2.3.2. We apply early stopping on the validation accuracy and the validation loss with a patience of 10 epochs. That is, we stop the model training if the validation accuracy does not increase every 10 epochs or if the validation loss does not decrease every 10 epochs. We repeat the training of each model ten times in order to obtain a average performance which is more robust. We present the result in Table 4.4.

Cryptocurrency	Network	Epochs	Loss	Accuracy (%)	F Measure (%)
btc	LSTM	16	1.01	65	78
	RNN	21	0.94	56	72
	MLP	23	0.95	59	74
dash	LSTM	20	0.93	51	68
	RNN	14	0.89	81	89
	MLP	20	0.8	65	79
xrp	LSTM	11	1.15	27	12
	RNN	16	1.18	21	23
	MLP	19	1.14	65	67
xmr	LSTM	28	0.76	90	95
	RNN	18	1.06	57	74
	MLP	11	0.79	94	97
ltc	LSTM	11	1.13	9	11
	RNN	11	1.16	17	19
	MLP	13	1.42	7	1
doge	LSTM	11	1.12	46	56
	RNN	11	1.14	25	28
	MLP	11	1.25	6	6
nxt	LSTM	11	1.11	5	3
	RNN	11	1.62	2	4
	MLP	11	1.16	27	36
nmc	LSTM	11	1.28	11	10
	RNN	11	1.23	14	21
	MLP	11	1.37	21	20

Table 4.4: Performance of the baseline models on the test set

For btc, dash, ltc and xmr we can see that the baseline models have already a good prediction power on the validation set with a minimum F measure of 68% for LSTM model on dash and a maximum F-measure of 97% for MLP model on xmr.

For the other cryptocurrency, the results are mixed. MLP and LSTM models have a good F-measure for the returns of xrp and doge respectively, since their generalization power is better than a pure random forecast that would have a F-measure of 33%.

For the other models, it seems that each network experienced real difficulties during the training process. For example, for nmc, the best performance of the three models occurred after the first iteration, since the validation loss nor the accuracy did not improve after the eleventh epoch. They have a large loss with a minimum of 1.23 and really poor prediction performance with a maximum F-measure of 21%. The networks are incapable of extracting general information from the training set. We can see that the same phenomenon occurred with

ltc, doge, and nxt.

This problem is maybe due to overfitting problems, see Section 2.3.1. We can do this diagnosis of the training process by observing the loss and accuracy curves on the training and validation sets. We present this curves for the first 150 epochs of the training of nmc on Figure 4.3 and 4.4.

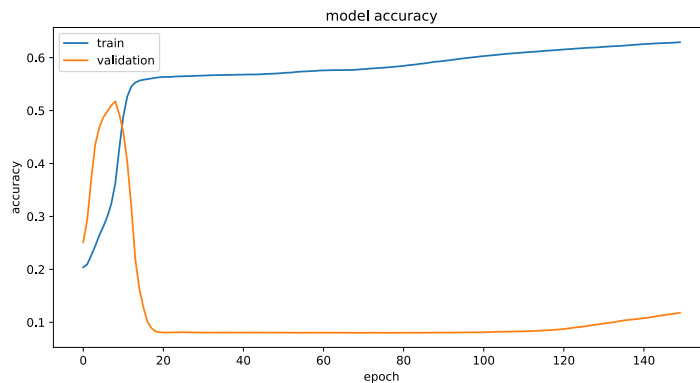


Figure 4.3: Training (blue curve) and validation (orange curve) accuracy

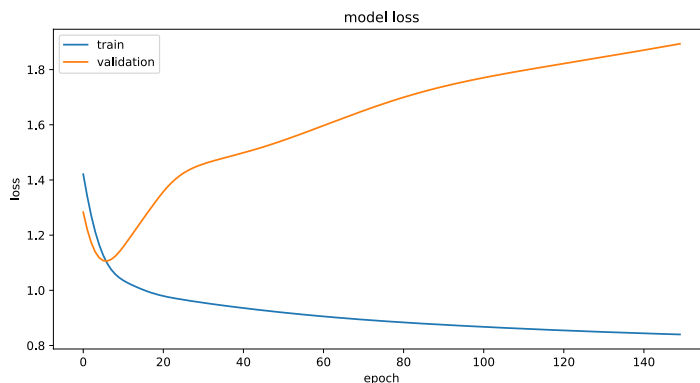


Figure 4.4: Training (blue curve) and validation (orange curve) loss

From the loss and accuracy curves, we can see that the model performs well on the train set. Nevertheless, the model overfits completely the train set. Indeed, the loss curve on the validation set increases while training and the model has no generalization power as the validation accuracy does not improve.

4.2.3 Best model selection: optimization methods and tuning of hyper-parameters

Regularization

As we saw in Section 2.3.2, regularization techniques are very useful to avoid overfitting problems. To improve our baseline models, we first need to apply one *dropout* and one *batch normalization* layer after each hidden layer to avoid overfitting and to make the training faster and easier.

Finally, to find the best architecture for each model, we have to tune the hyper-parameters of the models, that is the depth and the width of the models, but also the learning rate and the batch size.

The general architecture of the models are presented in Appendix B on Figure B.1, B.2 and B.3. The shape of the different layers is the same as the one from the baseline models except for the last dimension that depends on the number of neurons in the layer. We explain the tuning of this number and the number of hidden layers in the next paragraph.

Model tuning

As we saw in Section 3.3.2, model tuning is an important step of the model construction. We realize a tuning of the width and depth of each model as in the previous chapter, but we also tune the training parameters such as learning rate, batch size and epochs.

We first tuned the batch size with (32, 64, 128, 256) as grid search, to make computation easier for Keras. We chose 256 as batch size to reduce the variance on the evaluation metrics, which gives us more robust results. A large batch size also allow us to reduce considerably the training time. Finally, we tested (0.01, 0.001, 0.0001, 0.00001) as grid search for the learning rate and realized that small learning rates do not improve training and makes the complexity of the model higher with the need of more epochs. Thus, we selected 0.01, 0.001 and 0.0001 as potential learning rates.

To avoid a too large number of trainable parameters in the model, we prefer to use less neurons and more layers. For example, a MLP with one hidden layer with 100 neurons with 10 input variables has $(10 + 1) * 10 = 110$ trainable parameters, but a MLP with two hidden layers with 5 neurons each has $(5 + 1) * 10 + (5 + 1) * 5 = 85$ trainable parameters. Here, we prefer the second architecture. Nevertheless, we realize that adding a fourth layer did not improve the F-measure of the model.

Thus, we establish a grid search of parameters for the width (2, 5, 10, 15 and 24 neurons, 24 corresponding to the number of features in our model) and for the depth (2 or 3 layers). We select the parameters corresponding to the model with the highest F-measure.

We also could have tested much more values for the hyper-parameters of our models, but our computational resources were limited for this study.

4.2.4 Evaluation of the final models

In the Table 4.5, we present the evaluation metrics on the test set of the tuned models.

From Table 4.5, LSTM network has the best generalization power for btc, dash, ltc, nxt and nmc and MLP network has the best generalization power for xrp, xmr and doge. RNN always has a lower performance except for xmr which quarterly returns are perfectly predicted by the RNN and MLP networks with a F measure of 100%.

For each cryptocurrency, the results are different. We can see that the three different models for btc, dash and xmr achieve a very good performance, with a minimum F measure of 79% for xmr LSTM network. This result is important since this three cryptocurrencies are the most expensive. For the other cryptocurrencies, we must underline that RNN network seems to fail to extract the information necessary to perform a good generalization from the training set. This can be explainable by the overfitting problem and a fine tuning of the batch size and the learning rate is here necessary. We could also increase the complexity of the model by adding more training epochs.

To allow a reproducible method, we did not realize this fine tuning. We only compared the performance of the latter models with the performance of the baseline models and select the ones with the best F measure.

We finally obtain our final models that we will use for our trading strategy. We prefer the RNN baseline models for dash, xrp and doge, the LSTM baseline models for xmr and doge and finally the MLP baseline models for nxt. From the latter, we can see that it is easier to outperform the MLP baseline model, that indicate that the hyper-parameters of MLP network plays a more important role on the performance of the model than for LSTM and RNN networks.

Cryptocurrency	Model	Loss	Accuracy	F1 score
btc	LSTM	0.68	100	99
	RNN	1.10	60	75
	MLP	0.91	65	79
dash	LSTM	0.56	100	100
	RNN	1.00	77	87
	MLP	0.88	66	79
xrp	LSTM	1.27	55	54
	RNN	1.27	12	7
	MLP	1.04	64	67
xmr	LSTM	1.04	65	79
	RNN	0.74	100	100
	MLP	0.74	100	100
ltc	LSTM	1.06	64	71
	RNN	1.17	26	37
	MLP	1.17	16	22
doge	LSTM	1.02	66	52
	RNN	1.37	5	0
	MLP	1.08	74	73
nxt	LSTM	1.02	86	87
	RNN	2.37	19	6
	MLP	2.73	16	6
nmc	LSTM	1.06	71	75
	RNN	1.26	30	14
	MLP	0.95	81	86

Table 4.5: Tuned model: performance on the test set

In the Table 4.6, we present the architectures and the metrics of the final models selected.

On average, LSTM, RNN and MLP network achieve a performance of 78%, 43% and 68% F measure respectively which shows that LSTM is the best model in terms of prediction accuracy.

From Table 4.6, it is worth noticeable that MLP networks with two hidden layers achieve a better performance than MLP networks with three hidden layers. RNN and LSTM networks tend to be deeper, preferring three hidden layers for four and five cryptocurrencies out of eight respectively. The number of hidden units does not seem to have a major impact on the performance. Finally, LSTM network prefer a small learning rate of 0.001 or 0.0001 for xrp, whereas a high learning rate of 0.01 was selected for 5 RNN networks. Considering the low performance of RNN networks and the fact that small learning rate for RNN networks achieve a lower performance than high learning rate, we could maybe have adopted the technique of reducing the learning rate while training, which consists of beginning the training of the network with a high learning and reducing it by plateau monitoring the evolution of the metric considered.

Multi-training

The split into train/test sets and the amount of data in the train set may have an influence on the results. Indeed, the train set may include some special pattern in the data that are not useful for the generalizing purposes or may exclude general information that can improve the performance on the test set. That is why, we finally test the stability of the forecasting method when new information becomes available by retraining the models after each quarter.

We cut the test set in three sets Q_1 , Q_2 and Q_3 corresponding to three consecutive quarter

Cryptocurrency	Model	First layer	Second layer	Third layer	Learning rate	F1 score
btc	LSTM	15	5	15	0.001	99
	RNN	10	15	15	0.01	75
	MLP	5	15	0	0.001	79
dash	LSTM	15	5	15	0.001	100
	RNN	15	5	5	0.01	89
	MLP	5	15	0	0.001	79
xrp	LSTM	5	5	5	0.0001	54
	RNN	5	15	0	0.01	23
	MLP	24	15	0	0.001	67
xmr	LSTM	10	10	0	0.001	95
	RNN	24	24	24	0.01	100
	MLP	5	10	0	0.01	100
ltc	LSTM	5	5	5	0.001	71
	RNN	5	5	0	0.001	37
	MLP	5	15	0	0.001	22
doge	LSTM	10	10	0	0.001	56
	RNN	5	5	0	0.001	28
	MLP	5	24	0	0.001	73
nxt	LSTM	15	5	0	0.001	87
	RNN	10	5	5	0.001	6
	MLP	5	24	0	0.001	36
nmc	LSTM	2	2	2	0.001	75
	RNN	10	24	0	0.01	14
	MLP	5	15	0	0.001	86

Table 4.6: Final models: architecture

in the test sets. We do this experiment in three steps:

we predict $(Q_{1,pred})$ from the first quarter of the test set (Q_1) . We then add to the original train set (T_1) the first quarter of the test set. We then update the weights of the model by training the original model on the new train set $(T_2 = T_1 \cup Q_1)$ and we predict the second quarter $(Q_{2,pred})$ with the new model. We repeat this operation to get a new train set $T_3 = T_2 \cup Q_2$ and the last prediction $Q_{3,pred}$. Finally, we get predictions on the whole test set thanks to the three models $Pred = Q_{1,pred} \cup Q_{2,pred} \cup Q_{3,pred}$ and we compute the F-measure on this three-steps prediction on the whole test set to get the final performance.

We compare in Table 4.7 the performances between a one-shot and a three-steps prediction. If the generalization power improves, the last quarter added to the train set has crucial information for the next quarter, if not, either the model overfits on the new train set or the data itself has special patterns that are not useful for generalization.

Cryptocurrency	MLP		RNN		LSTM	
	One training	Multi training	One training	Multi training	One training	Multi training
btc	86	79	52	75	99	99
dash	91	79	24	87	100	100
xrp	41	67	23	23	54	57
xmr	85	100	63	100	95	95
ltc	34	22	42	37	71	80
doge	25	73	29	29	56	56
nxt	36	36	12	6	87	91
nmc	60	86	21	21	75	91
Average	57	68	33	47	78	84

Table 4.7: Comparison of F measure (in %) of one-shot and multi-training predictions

From Table 4.7, we see that for LSTM network, the multi-training experiment improved for each cryptocurrency the performance of the model, which underlines the ability of LSTM network to capture new patterns in the data and the necessity to retrain the models at each quarter for nmc, nxt and ltc. We obtain mixed results for MLP and RNN networks which indicates maybe a tendency to overfit new data when the F-measure declines from the one-shot and three-steps predictions. Nevertheless, on average we improve the generalization power by 11, 14 and 6 points for MLP, RNN and LSTM networks respectively.

4.3 Evaluation of different trading strategies

To evaluate our different models in terms of trading strategies, we need to look at another performance measure as the F-measure. Indeed, the F-measure gives us an idea of the prediction accuracy of each model, but not of the performance of a trading strategy based on each model prediction. To compare each model, we use in this section the financial returns of the trading strategies based on the prediction of each model.

Let us first define the return of a portfolio composed of cryptocurrencies.

Definition 4.1 (Portfolio return) *Let us consider a portfolio with N assets, P_t^i the price of asset i at time t . The T -days return of asset i is:*

$$r_{t+T}^i = \frac{P_{t+T}^i - P_t^i}{P_t^i}$$

The T -days return of the portfolio is then defined as:

$$R_{t+T} = w_1 r_{t+T}^1 + \dots + w_N r_{t+T}^N$$

We add a constraint on the weights, so the capital invested in the portfolio is divided between the assets included:

$$\sum_{i=1}^N w_i = 1$$

As we can see, w_i can be negative allowing for short position in the portfolio. Indeed, when we can take a short position on the cryptocurrency i , we sell on margin the cryptocurrency i , which implies that we borrow an amount M to a broker at the risk free interest rate, $r_{free} = 0.001$. The return of a long-short portfolio can be written:

$$R_{t+T} = \sum_{l \in L} w_l r_{t+T}^l + \sum_{s \in S} w_s r_{t+T}^s + 2 \left| \sum_{s \in L} w_s \right| r_{free}$$

where L and S contains the long and the short positions respectively.

Moreover, we need a benchmark to compare our strategies on the test set. We use three benchmarks, a portfolio replicating CRIX, that is CRIX quarterly returns, a portfolio based on the predictions of the baseline models for each type of neural networks and a portfolio based on perfect predictions or "true signals", that is the observed training signal on the test set. This benchmark reflects the accuracy of our predictive models.

Finally, we do not apply our strategy at the beginning of each quarter, but we apply it everyday, creating a new portfolio each day, as if we would have a new investor. In this way, we can measure not only the performance of a portfolio at the beginning of the test set, but the overall performance of our strategies on the whole test set. We can measure the overall performance of our different strategies with the cumulative quarterly returns, regardless of the date of investment, which gives us two ways to evaluate our strategies.

- We first use the quarterly returns everyday. If at time t , the quarterly return is higher than CRIX return at that date, we say that the portfolio beats the cryptocurrency market on that quarter. We can then say that on that quarter, it is more profitable to invest in our strategy, rather than in CRIX. We can use as indicator the number of days of investment which gives us higher quarterly returns than CRIX's.
- If the cumulative quarterly returns of our portfolio is higher than the cumulative returns of CRIX at the end of the test set, we can say that our portfolio beats the cryptocurrency market on the whole period considered. We can then conclude that it is always better to invest in our strategy, rather than in CRIX, on the test period. We use as indicator, the cumulative returns at the end of the test set.

4.3.1 Buy-and-hold strategy

Price weighted portfolio

First, we use a buy-and-hold strategy on the period considered, here a quarter of a year, that is we buy the cryptocurrency that returns has been predicted to increase significantly and we close all positions in our portfolio at the end of the quarter considered. We compare the returns of the portfolio with a portfolio replicating CRIX.

We suppose that we buy only one coin of each cryptocurrency and we compare the values of the portfolio at the beginning and at the end of the quarter considered. Let I be the ensemble of cryptocurrencies that we buy at time t and N_i , $i \in I$, the number of coins that we buy of the cryptocurrency i ; the quarterly return of this portfolio at time t is:

$$R_{t+90} = \frac{\sum_{i \in I} N_i P_{t+90}^i - \sum_{i \in I} N_i P_t^i}{\sum_{i \in I} N_i P_t^i}$$

Since in our case, we buy only one coin of each cryptocurrencies that we include in our portfolio, we have:

$$R_{t+90} = \frac{\sum_{i \in I} P_{t+90}^i - \sum_{i \in I} P_t^i}{\sum_{i \in I} P_t^i}$$

This portfolio correspond to a price weighted portfolio, which returns is influenced by the most expensive cryptocurrency. Indeed, cryptocurrencies with a higher price will be given more weight and will have a greater influence over the performance of the portfolio.

$$\begin{aligned} R_{t+89} &= \frac{\sum_{i \in I} P_{t+90}^i - \sum_{i \in I} P_t^i}{\sum_{i \in I} P_t^i} \\ &= \frac{\sum_{i \in I} (P_{t+90}^i - P_t^i)}{\sum_{i \in I} P_t^i} \\ &= \sum_{i \in I} \frac{P_t^i}{\sum_{i \in I} P_t^i} \frac{P_{t+90}^i - P_t^i}{P_t^i} \\ &= \sum_{i \in I} w_i R_{t+90}^i, \quad \text{where} \quad w_i = \frac{P_t^i}{\sum_{i \in I} P_t^i} \end{aligned}$$

This formula corresponds to Definition 4.1.

For example, if we build a portfolio with the 8 cryptocurrencies we consider in our thesis, we get on average on the test set the weights in Table 4.8.

We build three trading strategies based on the trading signals, that we obtain from the predictions of the different models. The result is then three price weighted portfolios based

Cryptocurrency	Weight
btc	0.93
dash	0.042
doge	4.8e-07
ltc	8.4e-03
nmc	5.5e-04
nxt	2.0e-05
xmr	1.4e-02
xrp	4.1e-05

Table 4.8: Weights of a price weighted portfolio, average on the test set

on the LSTM, RNN and MLP networks predictions respectively. The results are presented in Figure 4.5.

On Figure 4.5, we can see that LSTM portfolio clearly outperforms MLP portfolio from March 2017, but they are equivalent before this date. Indeed, we can see that MLP portfolio successively predicted wrong trading signals in March 2017 and Mai 2017, implying also that RNN portfolio outperforms MLP portfolio from April 2017.

Nevertheless, the three strategies and the perfect prediction portfolio do not beat CRIX before mid May, as we can see from the quarterly returns. Indeed, we know from Trimborn and Härdle (2016) that CRIX is based on market capitalization indexing, which gives higher weights to larger cryptocurrencies in terms of market capitalization (see next section for details on marketcap indexing). Thus, the structure of CRIX returns should be more influenced by the performance of the cryptocurrencies we consider. Nevertheless, we can see from Figure 4.5 that CRIX returns are two times larger than our portfolio at end of March 2017.

From this result, we can say that cryptocurrencies with a lower marketcap than the height cryptocurrencies we consider, had very large returns between February and June 2017, since they have a smaller weight in CRIX than our cryptocurrencies. We must also remind the reader that Ethereum, which value was multiplied by a factor of 100 during that period, is not included in our strategies but is in the computation of CRIX returns on the test set. Yet, Ethereum is the second cryptocurrency in terms of market capitalization, thus has the second most important influence on CRIX returns on the test set. From this, we can explain, as we can see from the cumulative returns, how CRIX portfolio outperforms all our strategies and the perfect prediction portfolio on the whole period.

Nevertheless, LSTM portfolio always beats CRIX from mid May, which indicates that the most expensive cryptocurrencies were the best investment solutions during the last quarter. Similarly, MLP portfolio beats CRIX from mid June.

Since the returns of a price weighted portfolio are influenced by the most expensive cryptocurrency, we can say that the latter strategies are highly influenced by btc changes. If one model is wrong at predicting btc returns, it can have a very low performance, but if it is wrong at predicting cheap cryptocurrencies, it can however have a good performance.

From this, we can clearly see that a price weighted portfolio does not profit from high returns in cheap cryptocurrencies. Yet, xrp, nxt and doge, which are the cheapest cryptocurrencies we study (see Table 4.1), experienced the highest returns on the test set as we can see on Figure A.2 in Appendix. That is why we also build a portfolio based on market capitalization weighting.

Market capitalization weighted portfolio

The market capitalization or "marketcap" of a cryptocurrency i is defined as $K_t^i = S_t^i \times P_t^i$ where P_t^i and S_t^i are respectively the price and the supply, that is the number of coins on the

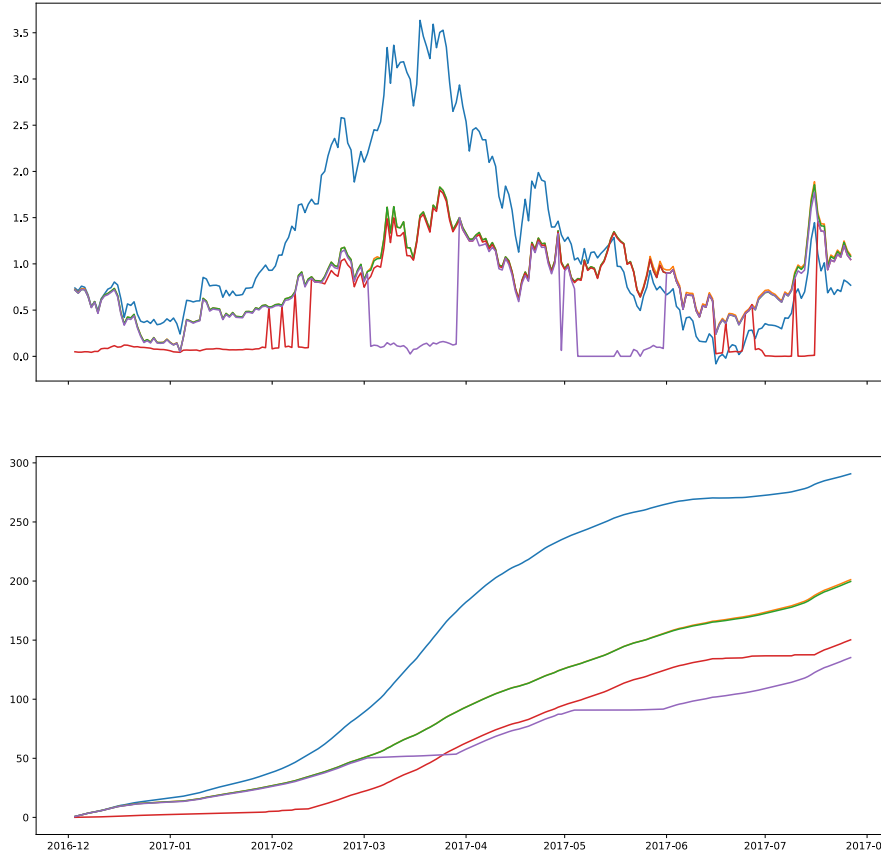



Figure 4.5: Price weighted quarterly returns (top) and cumulative quarterly returns (bottom) of CRIX (blue), perfect prediction (orange), LSTM (green), RNN (red) and MLP portfolios (purple)  CRIXportfolio

market, of cryptocurrency i at time t .

A marketcap weighted portfolio has its weights proportional to the market capitalization of its cryptocurrencies, that is cryptocurrencies with a higher marketcap will be given more weight. The return of such portfolio is defined as:

$$R_{t+90} = \sum_{i \in I} w_i r_{t+90}^i, \quad \text{where} \quad w_i = \frac{K_t^i}{\sum_{i \in I} K_t^i}$$

For example, if we build a portfolio with the height cryptocurrencies we consider in our thesis, we get on average on the test set the weights in Table 4.8. As we can expect, the portfolio is more diversified than a price weighted portfolio.

We present the performance of our strategies on Figure 4.6. RNN portfolio has the best performance and beats CRIX on the whole test set thanks to very large returns during the first quarter.

Nevertheless, the overall performance of the other strategies is similar to the price weighted

Cryptocurrency	Weight
btc	0.86
dash	0.017
doge	0.081
ltc	0.011
nmc	0.023
nxt	2.8e-03
xmr	1.0e-03
xrp	4.4e-04

Table 4.9: Weights of a marketcap weighted portfolio, average on the test set

portfolio and CRIX beats MLP and LSTM strategies. Indeed, since the market is growing, a marketcap weighted portfolio based on accurate predictions corresponds approximatively to a basket portfolio containing the height largest constituents of CRIX, since CRIX weights are defined by market capitalization (Trimborn and Härdle; 2016).

Finally, a marketcap strategy cannot profit from high returns in cryptocurrencies with lower marketcap. Yet, nxt and doge, which have a low marketcap (see Table 3.1), experienced very high returns as we can see from Figure A.2 in Appendix. In order to benefit from both high returns in cryptocurrencies with low price and market capitalization, we build a last trading strategies with an equally weighted portfolio, the simplest Beta strategy.

Equally weighted portfolio

An equally weighted portfolio weights each stock in the basket equally. As a result, the portfolio is highly diversified. As opposed to marketcap weighted portfolio, it does not overweight overpriced cryptocurrencies and underweight underpriced cryptocurrencies, which implies that it can overweight cryptocurrencies with a larger risk. Indeed, smaller cryptocurrency have a higher risk of failure.

Nevertheless, an equally weighted portfolio is really easy to construct, since its weights are inversely proportional to the number of cryptocurrencies in the basket. The quarterly return of such portfolio is defined as follows, where N_t is the number of stocks in the basket at time t :

$$R_{t+90} = \frac{1}{N_t} \sum_{i \in I} r_{t+90}^i$$

We present the performance of the different strategies based on such a portfolio on Figure 4.7. As we can see, the strategies based on our three predictive models and the perfect prediction portfolio beat the market on the whole test set, since the cumulative returns are much higher than CRIX portfolio at the end of the test period. Indeed, these portfolios benefit from very high returns in cheap cryptocurrencies at the beginning of the test set, as opposed to CRIX. However, at the end of the test period, the different strategies have a similar performance. Indeed, cheap currencies, for example xrp, nxt or doge, experienced very low returns, close to 0 or even negative returns, from end of May 2017. Since, these cryptocurrencies are overweighted comparing to CRIX weights, CRIX beats our equal weighted portfolios for some days in the last quarter.

Nevertheless, the overall performance of the strategies based on our predictive models is much higher than CRIX's (almost 4 times higher for MLP portfolio).

Finally, from Table 4.10, we can see that the strategies based on the predictions of the tuned model always beats the strategies based on the predictions of the baseline models for each type

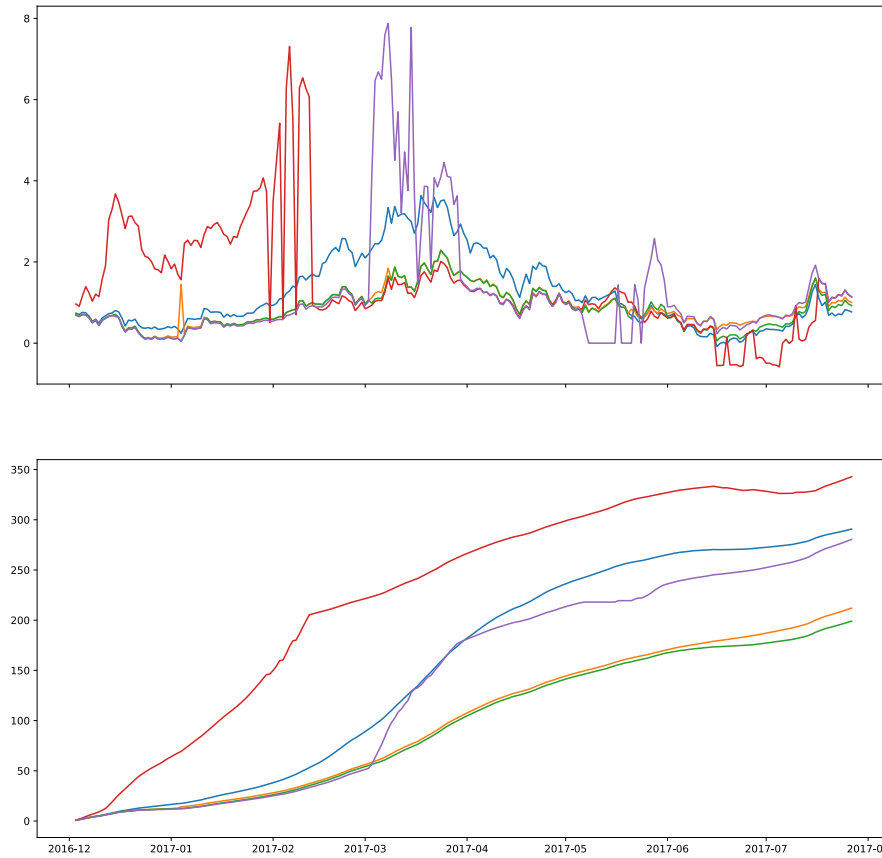



Figure 4.6: Marketcap weighted quarterly returns (top) and cumulative quarterly returns (bottom) of CRIX (blue), perfect prediction (orange), LSTM (green), RNN (red) and MLP portfolios (purple)  CRIXportfolio

of neural networks. The best strategy is an equally weighted portfolio based on the predictions from MLP model.

4.3.2 Long short strategy

In the latter subsection, we exposed some trading strategies based on a 3 months buy-and-hold strategy that forbid short positions, but, since we trained our neural networks so they can predict negative returns, we can profit from these predictions by short selling cryptocurrencies that are predicted to decrease significantly.

In this section, we build our portfolio in the following manner. We buy the cryptocurrencies that are predicted to increase significantly (long position) and sell them after three months as in a buy-and-hold strategy. Moreover, if we predict that a cryptocurrency will decrease significantly, we borrow the cryptocurrency, or fiat money, to a broker to sell it on the same day. At the end of the quarter, we close the short position by purchasing the cryptocurrency borrowed in order to give it back to the broker. If the price actually decreased, we made a profit.

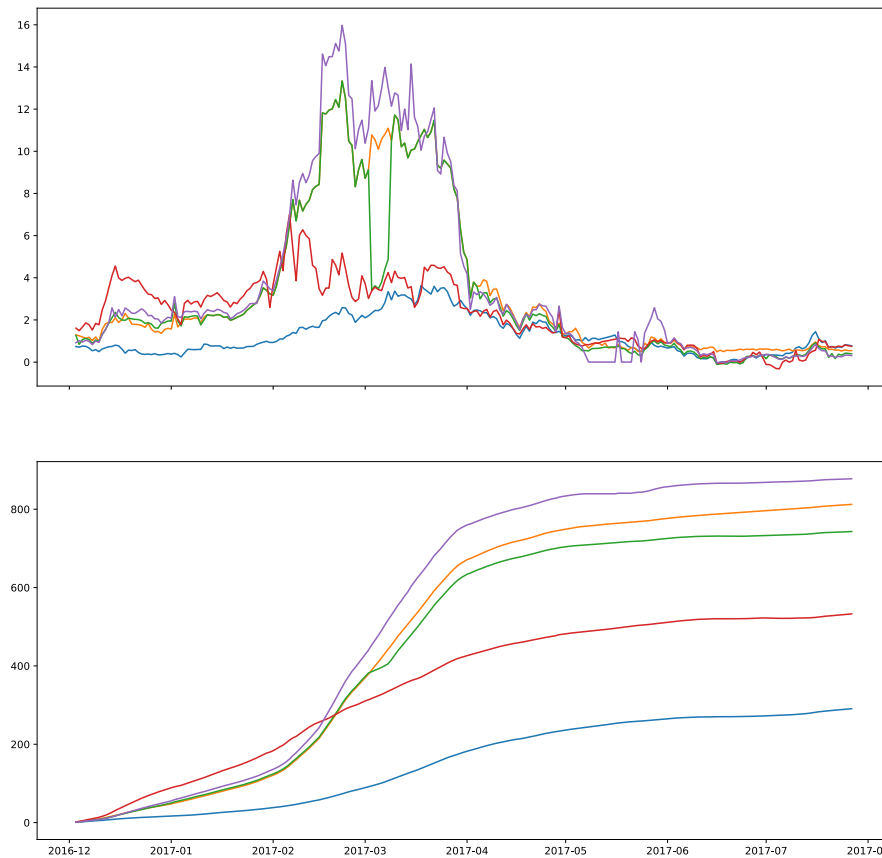



Figure 4.7: Equally weighted quarterly returns and cumulative quarterly returns of CRIX (blue), perfect prediction (orange), LSTM (green), RNN (red) and MLP portfolios (purple)  CRIXportfolio

As we can see from Table 4.11, we observe first that the strategies based on the final models have a larger cumulative return at the end of the test period than the strategies based on the baseline models, except for the strategies based on marketcap portfolio with MLP predictions and equally weighted portfolio with RNN predictions.

Again, price weighted and marketcap final portfolios do not beat CRIX and have quite similar performances on the whole test set (see Figure C.1 and Figure C.2 in Appendix). Only the equal weighted portfolios based on MLP and LSTM predictions beat CRIX (see Figure C.3 in Appendix), LSTM equal weighted portfolio performing more than two times better than CRIX for a final return of 627.

LSTM portfolio always beats RNN or MLP portfolios on the long run, which means that wrong predictions by RNN or MLP networks cost a lot in terms of returns to the trading strategies. Indeed, if we look at the returns of the equally weighted portfolio based on RNN final predictions, we can see that the strategy has a negative performance. This is caused by wrong predictions of long instead of short trading signals, and conversely. Indeed, LSTM

Portfolio	Baseline model			Final model		
	Price	Marketcap	Equal	Price	Marketcap	Equal
CRIX	291	291	291	291	291	291
LSTM	126	152	468	200	199	743
RNN	110	147	630	150	343	533
MLP	116	241	626	135	280	878
Maximum	CRIX	CRIX	RNN	CRIX	CRIX	MLP
Best NN	LSTM	MLP	RNN	LSTM	RNN	MLP

Table 4.10: Long strategy cumulative return at the end of the test period

Portfolio	Baseline model			Final model		
	Price	Marketcap	Equal	Price	Marketcap	Equal
CRIX	291	291	291	291	291	291
LSTM	122	136	-299	198	195	627
RNN	104	102	-193	129	133	-353
MLP	111	216	59	130	175	458
Maximum	CRIX	CRIX	CRIX	CRIX	CRIX	LSTM
Best NN	LSTM	MLP	MLP	LSTM	LSTM	LSTM

Table 4.11: Long/short strategies cumulative returns at the end of the test period

strategy has a better performance, because LSTM model has a better F-measure.

Comparison of buy-and-hold and long/short strategies

If our models would have a very low misclassification rate between "long" and "short" trading signals, we can benefit from a long/short strategy. In Table 4.12, we compare the overall performance of a buy-and-hold and a long/short strategy. We can observe that the long strategies always beat the long/short strategy. Indeed, in a long/short strategy, it is more costly to predict a "long" trading signal instead of a "short", and conversely, than to predict a "do nothing" instead of a "short" or a "long" trading signal.

We can also see that the final performances of the price weighted portfolios based on long or long/short strategies are quite similar, but they differ when we compare the marketcap or equally weighted portfolios, especially for the portfolios based on MLP and RNN predictions.

Portfolio	Long			Long/short		
	Price	Marketcap	Equal	Price	Marketcap	Equal
CRIX	291	291	291	291	291	291
LSTM	200	199	743	198	195	627
RNN	150	343	533	129	133	-353
MLP	135	280	878	130	175	458
Maximum	CRIX	CRIX	MLP	CRIX	CRIX	LSTM
Best NN	LSTM	RNN	MLP	LSTM	LSTM	LSTM

Table 4.12: Long and Long/short strategies cumulative returns at the end of the test period

In general, from these results, we prefer to apply a buy-and-hold strategy that does not allow short position, because our models predict too many trading signals of short, instead of long

positions and conversely. However, we know that it is hard to beat an index when the market is growing and, during the year 2017, the cryptocurrency market has known a exponential growth. From Table 4.13, we can see that on average on all strategies, our portfolio beats CRIX on 38% of the test set, which is quite remarkable. Moreover, the equally weighted portfolios always beats CRIX, except for RNN long/short portfolio, with a maximum cumulative return of 878 for MLP long portfolio, that is 3 times higher than CRIX.

Portfolio	Long			Long/short			Average	Best strategy
	Price	Marketcap	Equal	Price	Marketcap	Equal		
MLP	24	38	79	24	32	44	40	Equal
LSTM	32	26	72	32	25	49	39	Equal
RNN	22	45	73	22	24	15	34	Equal
Average	26	36	75	26	27	36	38	Equal
Best strategy	LSTM	RNN	MLP	LSTM	MLP	LSTM	MLP	MLP Equal

Table 4.13: Number of days (in percentage) with higher returns than CRIX portfolio

Chapter 5

Discussion and future works

5.1 Discussion on the method

5.1.1 Data

Data quality

In this thesis, we used Deep Learning methods that require a large amount of data in order to estimate the parameters of the model correctly. Especially, the first solution to overfitting problems is to collect more data. Since the cryptocurrency market is quite new, we could not get more daily data for the cryptocurrencies we considered without introducing missing values. A follow-up to this thesis would be to work on data with higher frequencies (for example every five minutes) and reduce the holding period of the portfolio.

Generalization

The cryptocurrency market is a very dynamic market. Its structure has changed in comparison with January 2017 when we started our study. Some cryptocurrencies we have considered are not the part of largest in market capitalization, one year later at the time of writing. Major cryptocurrencies have emerged such as Ethereum, BitcoinCash, Iota, and many more. That is why it would be necessary to always update the initial basket in relation with the evolution of the market structure.

5.1.2 Model architecture

First, we could have used a random grid search for the tuning of the width and depth of the different neural networks. Indeed, it showed very good performance in the literature. We could also have gone further in the tuning of the hyper-parameters of the models that have a low performance on the test set. Indeed, model selection involves multiple trials and errors, which need important computation power that we didn't have for this study. We could also have tested other types of models such as convolutional neural network, but we wanted to underline the ability of LSTM to outperform former simple RNN and MLP.

5.2 Discussion on the results

The question of predictions on the financial market is very problematic, especially on the cryptocurrency market where the problems of liquidity, whales or scams have raised too many times. Moreover, 2017 has been a new example of gigantic bubble on the cryptocurrency market where Bitcoin price reached 20 000 dollars. That is why our results have to be taken with great care, since the question of prediction during a bubble is quite problematic. At least, we can expect

our models to perform very well since it is much easier to predict long positions in a highly growing market. Thus, even if the deep learning methodology always tries to avoid overfitting, it would be interesting to see how our models perform in a bearish market.

5.3 Towards a real portfolio management strategy

5.3.1 Model, metric and loss functions

We chose to build height different models to predict independently the price trend of each cryptocurrencies as in Franke's study. It would have been interesting to predict the weight of each cryptocurrency inside a weighted portfolio and maximize its returns. To this model we could have used a custom loss function so the model learns to penalize misclassifications that cost more to the trading strategy. Indeed, we satisfied ourselves with the built-in loss and metric functions of Keras library, that is *sparse_categorical_crossentropy()* and *accuracy()*. For example, in order to find the strategy with the highest return, we could have chosen the return of the trading strategy based on predictions on the test data, as a selection criterion for our final model. Nevertheless, if this strategy can maybe work in a short term manner, in the long run, without taking into account any risk measure, it would eventually fails.

5.3.2 Risk management

Indeed, our strategy, which based its quarterly investment decisions only on returns, is not a portfolio management system because it is not realistic. Indeed, we have to take into consideration the volatility of the portfolio in order to reflect a real investor behavior. Economic agents are rational and take risk into consideration. We can state some portfolio management strategies such as Markowitz portfolio, Sharpe ratio, equirisk contribution, MaxDrawDown, Optimal diversification, see Franke (1999) for an early application of neural networks in portfolio management strategy. Nevertheless, these portfolio management solutions have very strong assumptions on the distribution of returns, assumptions that are very hard to meet in real life, in particular on the cryptocurrency market which is highly volatile.

5.4 Reinforcement learning

In our studies, we predict future price movements and consider them as trading signal, converting directly the trend into an action for the investor. However, price predictions are not market actions. Moreover, our models do not predict the amount we should buy or sell for each cryptocurrency. To build a real portfolio management strategy, we need to consider another type of machine learning, Reinforcement learning, that simulates a real artificial intelligence. Reinforcement learning studies the reaction of an agent towards its environment thanks to a policy and a reward function. Jiang et al. (2017) present a financial-model-free Reinforcement Learning framework to provide a deep machine learning solution to the portfolio management problem applied to cryptocurrencies.

Chapter 6

Conclusion

In this thesis, we presented a general introduction on the deep neural networks theory. We applied it on financial time series that is why we focused our analysis on recurrent neural networks as a nonlinear method for sequence learning. We explained the basics of MLP, RNN and LSTM architectures and the deep learning methodology in order to open what some researchers or practitioners called the “black box” of neural networks. The reader should have a basics theoretical knowledge on how to choose the neural network model corresponding to its particular problem and how to train it in order to build prediction on unseen data.

We also showed on two examples how we can apply this methodology to a real practical problem: price prevision in order to take financial decisions. With this thesis, we add to the literature by providing a first cryptocurrencies portfolio based on deep learning asset selection strategies. By tuning the different hyper-parameters of the model with the trial-and-error methodology that we presented, the reader should be able to find a model with an acceptable generalization power. We showed how this methodology is necessary since hyper-parameter tuning always improves the prediction accuracy of the model.

If our performance results on the cryptocurrency market should be taken with great care, since the market was experiencing abnormal positive returns, typical of a bubble situation, we showed how neural networks, especially LSTM, are useful tool for trend predictions by achieving high prediction accuracy. Our strategy succeed to beat CRIX index in terms of financial returns showing how index funding can be outperformed with an AI based buy-and-hold strategy, even in a highly growing market.

Nevertheless, predicting the market is very risky and a realistic investment system should be implemented by taking into account the active environment where it is evolving. A neural network for trend prediction is not able to understand the financial cost of misclassifications. That is why it would be interesting to study how such strategy would perform by adding a financial policy and risk measure to the learning process in a Reinforcement Learning manner.

Appendix A

Cryptocurrencies

A.1 Cryptocurrencies prices

A.2 Cryptocurrencies quarterly returns

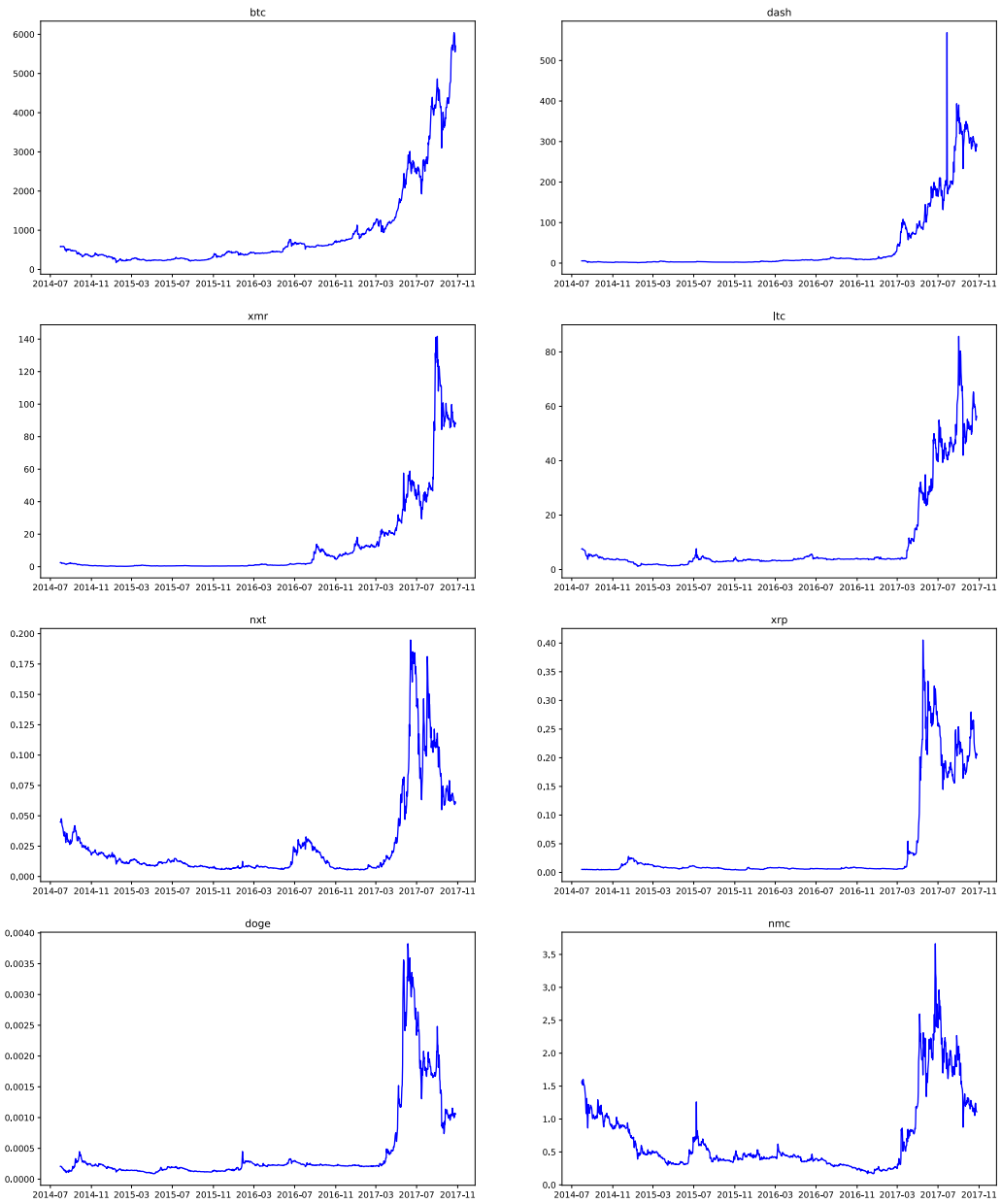


Figure A.1: Cryptocurrencies prices in dollars

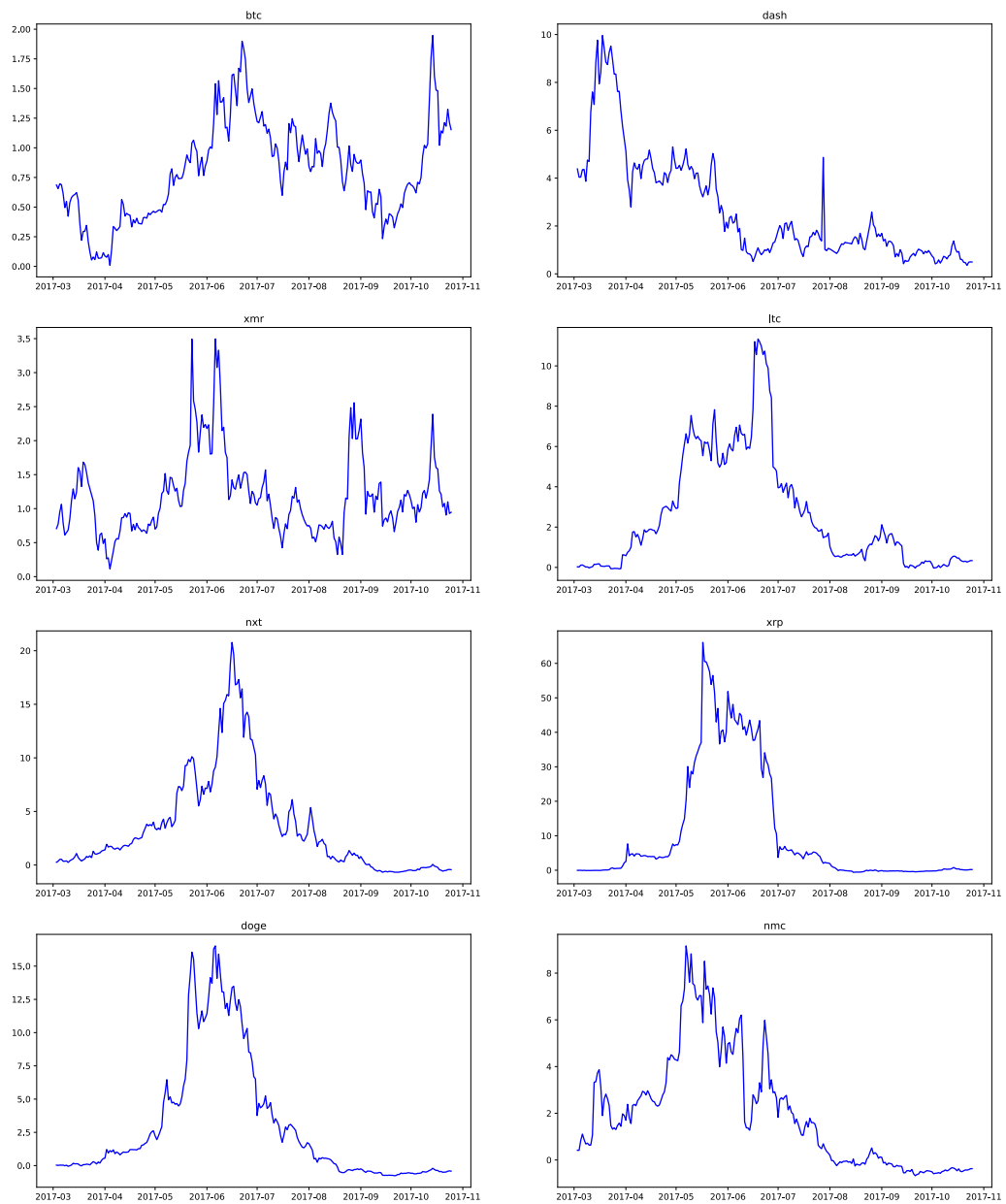


Figure A.2: Cryptocurrency quarterly returns on the test set

Appendix B

Models architectures

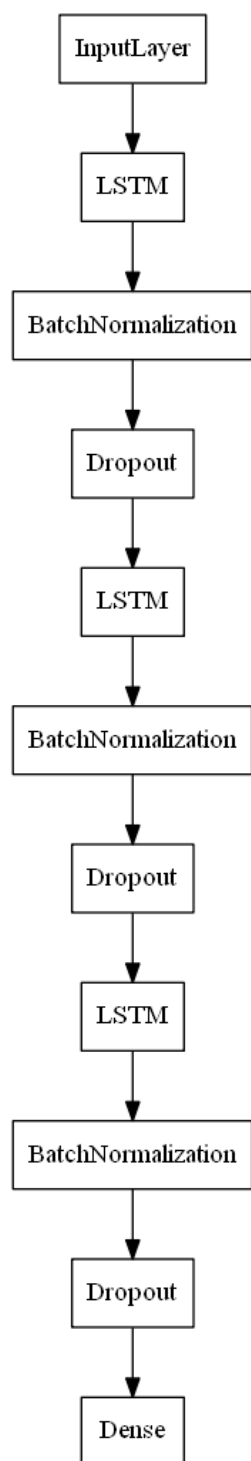


Figure B.1: LSTM model

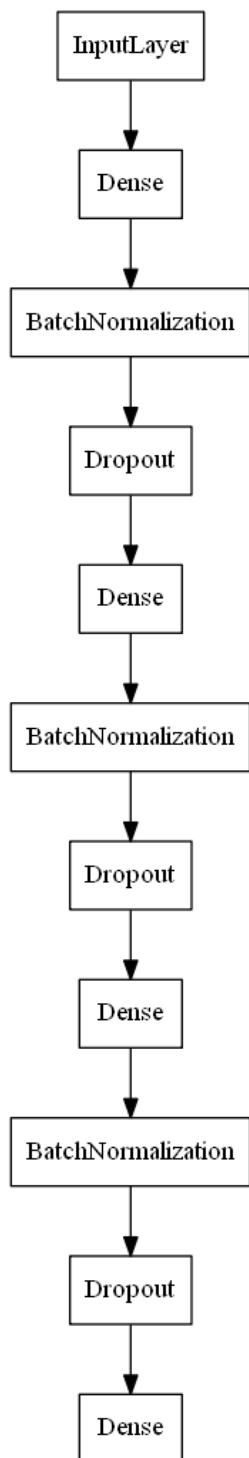


Figure B.2: MLP model

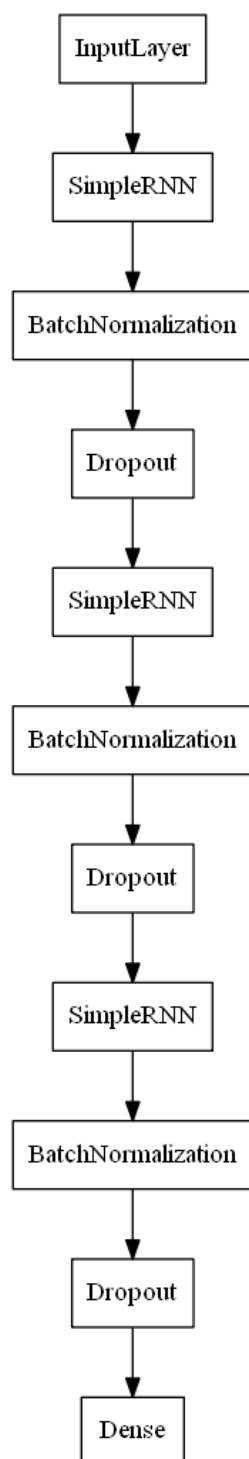


Figure B.3: RNN model

Appendix C

Long short portfolio

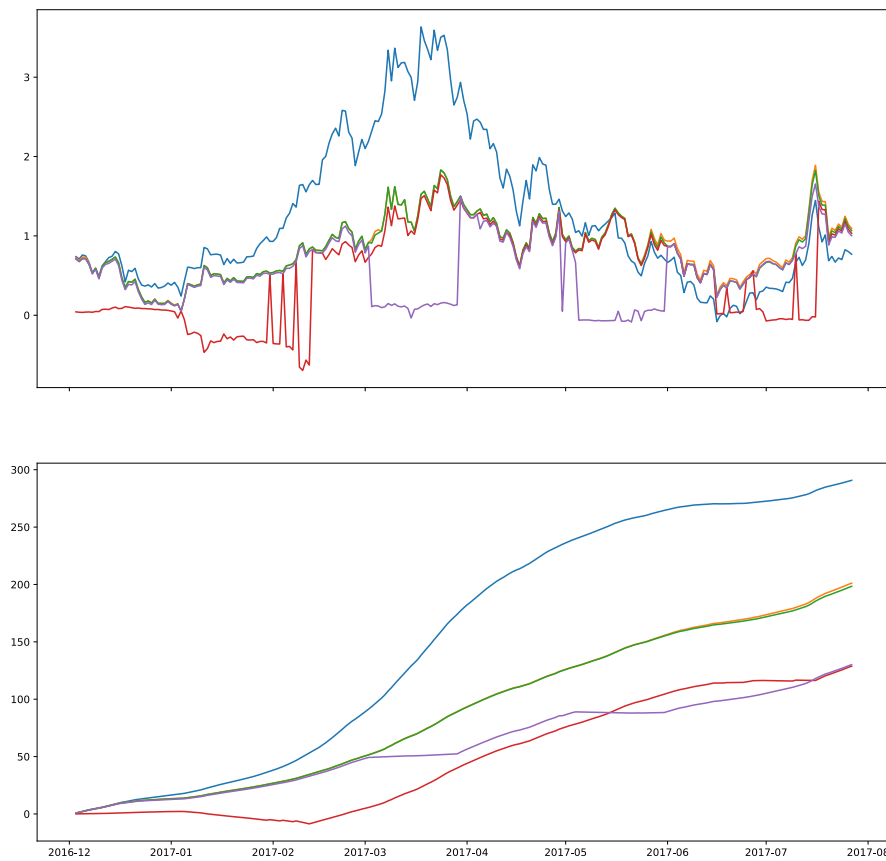



Figure C.1: Long short price weighted quarterly returns and cumulative quarterly returns of CRIX (blue), perfect prediction (orange), LSTM (green), RNN (red) and MLP portfolios (purple)  CRIXportfolio

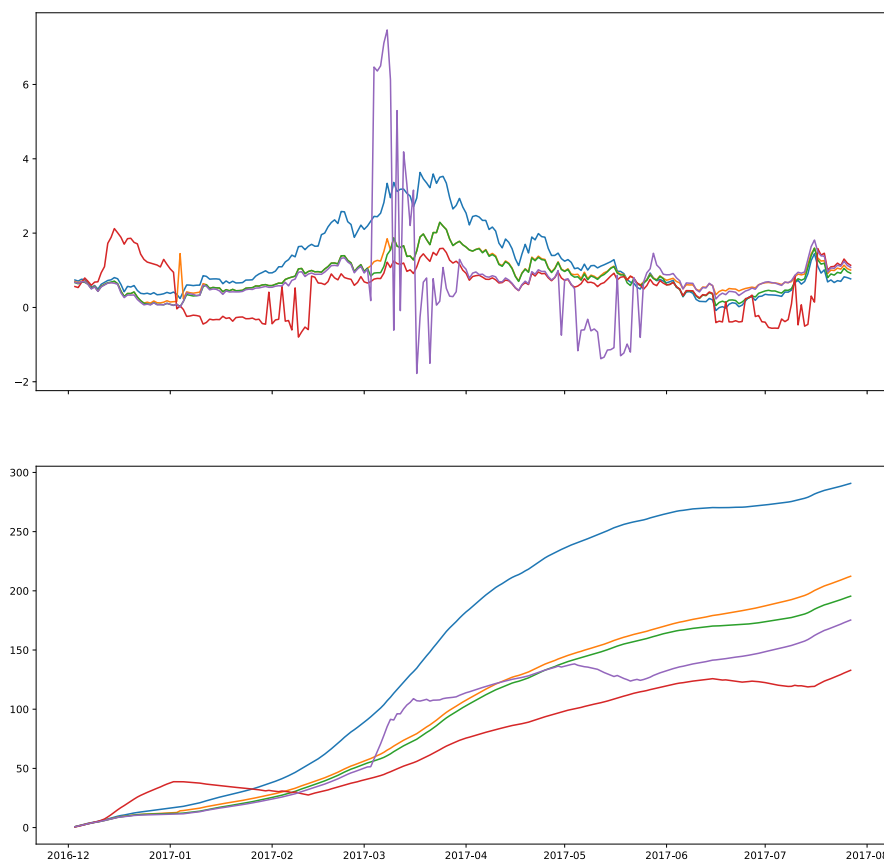



Figure C.2: Long short marketcap weighted quarterly returns and cumulative quarterly returns of CRIX (blue), perfect prediction (orange), LSTM (green), RNN (red) and MLP portfolios (purple)  CRIXportfolio

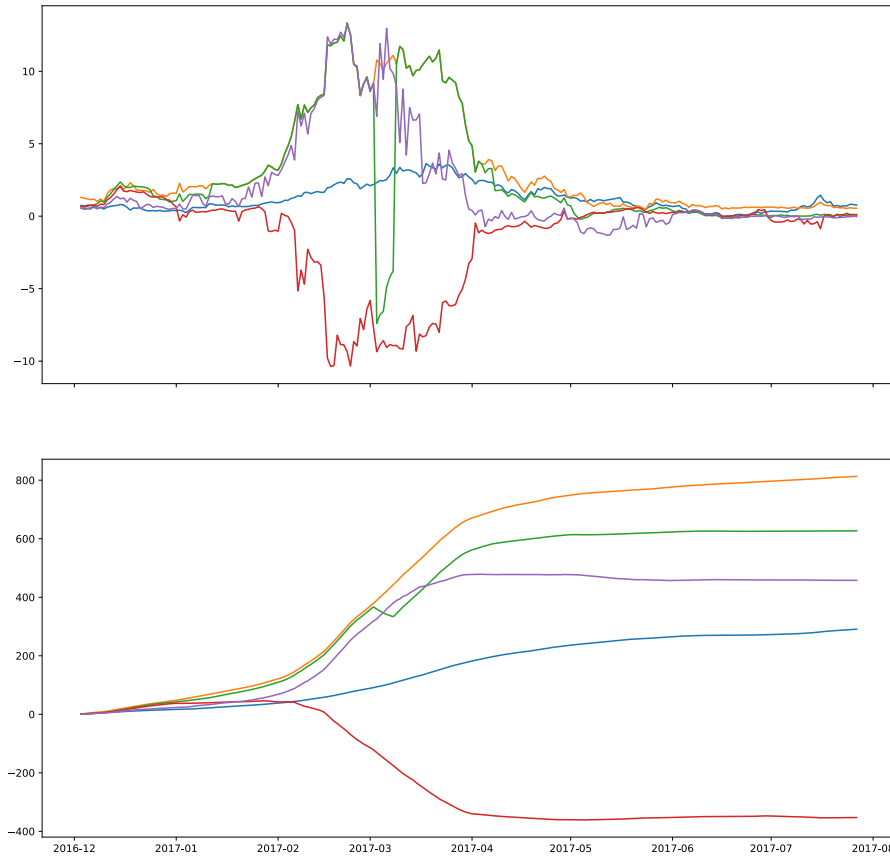



Figure C.3: Long short equally weighted quarterly returns and cumulative quarterly returns of CRIX (blue), perfect prediction (orange), LSTM (green), RNN (red) and MLP portfolios (purple)  CRIXportfolio

Bibliography

- Back, A. and Tsoi, A. (1991). Fir and iir synapses, a new neural network architecture for time series modeling, *Neural Computation* pp. 375–385.
- Bottou, L. (1998). On-line learning and stochastic approximations, in D. Saad (ed.), *On-line Learning in Neural Networks*, Cambridge University Press, New York, NY, USA, pp. 9–42.
- Domingos, P. (1999). The role of occam’s razor in knowledge discovery, *Data Mining and Knowledge Discovery* **3**(4): 409–425.
- Duda, R. O., Hart, P. E. and Stork, D. G. (2000). *Pattern Classification (2Nd Edition)*, Wiley-Interscience.
- Eisl, A., Gasser, S. M. and Weinmayer, K. (2015). Caveat emptor: Does bitcoin improve portfolio diversification?, *SSRN Scholarly Paper ID 2408997*. Rochester, NY: Social Science Research Network .
- Elendner, H., Trimborn, S., Ong, B. and Lee, T. M. (2017). The cross-section of cryptocurrencies as financial assets, in D. Lee Kuo Chen and R. Deng (eds), *Handbook of Digital Finance and Financial Inclusion: Cryptocurrency, FinTech, InsurTech, Regulation, ChinaTech, Mobile Security, and Distributed Ledger. 1st Edition*.
- Elman, J. L. (1990). Finding structure in time, *Cognitive Science* **14**(2): 178–211.
- Franke, J. (1999). Nonlinear and nonparametric methods for analyzing financial time series, in P. Kall and H.-J. Lüthi (eds), *Operations Research Proceedings 1998: Selected Papers of the International Conference on Operations Research Zurich, August 31 – September 3, 1998*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 271–282.
- Franke, J., Härdle, W. K. and Hafner, C. M. (2015). *Statistics of Financial Markets*, Springer.
- Gers, F. A., Schmidhuber, J. and Cummins, F. (2000). Learning to forget: Continual prediction with lstm, *Neural Computation* **12**(10): 2451–2471.
- Glorot, X., Bordes, A. and Bengio, Y. (2011). Deep sparse rectifier neural networks, in G. Gordon, D. Dunson and M. Dudík (eds), *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics*, Vol. 15 of *Proceedings of Machine Learning Research*, PMLR, Fort Lauderdale, FL, USA, pp. 315–323.
- Goodfellow, I., Bengio, Y. and Courville, A. (2016). *Deep Learning*, MIT Press, Cambridge, MA, USA. <http://www.deeplearningbook.org>.
- Graves, A. (2012). *Supervised Sequence Labelling with Recurrent Neural Networks*, Springer.
- Graves, A. (2013). Generating sequences with recurrent neural networks.
- Hinton, G. (2012). Overview of mini-batchngradient descent. unpublished lecture.

- Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory, *Neural Computation* **9**(8): 1735–1780.
- Hornik, K., Stinchcombe, M. and White, H. (1989). Multilayer feedforward networks are universal approximators, *Neural Networks* **2**: 359–366.
- Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift, *CoRR* **abs/1502.03167**.
- Jarrett, K., Kavukcuoglu, K., Ranzato, M. and LeCun, Y. (2009). What is the best multi-stage architecture for object recognition?, *2009 IEEE 12th International Conference on Computer Vision* pp. 2146–2153.
- Jiang, Z., Xu, D. and Lian, J. (2017). A deep reinforcement learning framework for the financial portfolio management problem, *ArXiv e-prints*.
- Jordan, M. I. (1986). Serial order: A parallel distributed processing approach, *Technical report*, Institute for Cognitive Science, University of California, San Diego.
- Kingma, D. and Ba, J. (2015). Adam: A method for stochastic optimization, *International Conference on Learning Representations*.
- LeCun, Y., Bottou, L., Orr, G. B. and Müller, K.-R. (1998). Efficient backprop, in G. B. Orr and K.-R. Müller (eds), *Neural Networks: Tricks of the Trade*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 9–50.
- Leshno, M., Ya. Lin, V., Pinkus, A. and Schocken, S. (1993). Multilayer feedforward networks with a nonpolynomial activation function can approximate any function, *Neural Networks* **6**: 861–867.
- Özgür, A., Özgür, L. and Güngör, T. (2005). Text categorization with class-based and corpus-based keyword selection, in P. Yolum, T. Güngör, F. Gürgen and C. Özturan (eds), *Computer and Information Sciences - ISCIS 2005: 20th International Symposium, Istanbul, Turkey, October 26-28, 2005. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 606–615.
- Pascanu, R., Mikolov, T. and Bengio, Y. (2013). On the difficulty of training recurrent neural networks, in S. Dasgupta and D. McAllester (eds), *Proceedings of the 30th International Conference on Machine Learning*, Vol. 28 (3) of *Proceedings of Machine Learning Research*, PMLR, Atlanta, Georgia, USA, pp. 1310–1318.
- Qian, N. (1999). On the momentum term in gradient descent learning algorithms, *Neural Networks*.
- Rojas, R. (1996). *Neural Networks: A Systematic Introduction*, Springer.
- Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain, *Psychological review* **65**(6): 386–408.
- Rumelheart, D. E., Hinton, G. E. and Williams, R. J. (1986). Learning internal representations by error propagation, in D. E. Rumelheart and J. L. McClelland (eds), *Parallel distributed processing: explorations in the microstructure of cognition*, MIT press, Cambridge, MA, USA, chapter 8, pp. 318–362. vol. 1.
- Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting, *Journal of Machine Learning Research* **15**(1): 1929–1958.

- Trimborn, S. and Härdle, W. K. (2016). Crix an index for blockchain based currencies, *SFB 649 Econmic Risk, revise and resubmit Journal for Empirical Finances* **2016-021**.
- Trimborn, S., Li, M. and Härdle, W. (2017). Investing with cryptocurrencies - a liquidity constrained investment approach, *SFB 649 Discussion Paper* (2017-014).
- Williams, R. J. and Zipser, D. (1995). Gradient-based learning algorithms for recurrent networks and their computational complexity, in D. E. Rumelheart and J. L. McClelland (eds), *Back-propagation: Theory, Architectures and Applications*, Lawrence Erlbaum Publishers, Hillsdale, N.J., chapter 13, pp. 433–486.
- Yao, Y., Rosasco, L. and Caponnetto, A. (2007). On early stopping in gradient descent learning, *Constructive Approximation* **26**(2): 289–315.
- Zimmermann, M., Chappelier, J.-C. and Bunke, H. (2006). Offline grammar-based recognition of handwritten sentences, *IEEE Transactions on Pattern Analysis and Machine Intelligence* pp. 818–821.