UNIVERSITÀ
DEGLI STUDI
FIRENZE

Scuola di Economia e Management
MSc in Finance and Risk Management

Thesis

DEEP LEARNING MODELS FOR HIGH-FREQUENCY CRYPTOCURRENCY FORECASTING

ANDREA VENUTA

Advisors: *Lucio Geronazzo, Pierluigi Zezza*

Academic Year 2017-2018

CONTENTS

LIST OF FIGURES

"Unlike a well-defined, precise game like Russian roulette, where the risks are visible to anyone capable of multiplying and dividing by six, one does not observe the barrel of reality."
— Nassim Nicholas Taleb

1

INTRODUCTION

Cryptocurrencies are loosely defined as currencies whose transactions are secured by means of cryptography. Most, but not necessarily all, cryptocurrencies are based on the principle of the *blockchain*: a distributed ledger of cryptographically signed transactions whose integrity is built upon the consensus of all participants, resulting in a decentralized mechanism lacking both a single point of failure and the need of trusted intermediaries for its correct functioning.

By now nearly two decades old, the concept of *cryptocurrency* has gained incredible momentum and popularity in the very recent past. The sudden surges in prices have been attracting retail and professional investors alike; the underlying *blockchain* technology is the focus of a growing body of industrial and academic research in the field of computer science, all the while the banks and governments attempt to tackle the issues arising from the way cryptocurrencies interact with fiat money and existing financial instruments.

The entire ecosystem is still poorly understood by all the parties involved, with the bulk of general opinion polarized on either side of the bull-bear spectrum. As a result, the markets where cryptocurrencies are traded exhibit extraordinary uncertainty not only with respect to pricing, but also in terms of volatility and liquidity.

Barring the possibilities of studying the fundamentals of the phenomenon of *cryptocurrencies* via traditional models – and moreso with a long term outlook, due both to the peculiar nature of a cryptocurrency as an asset, and the debatable forecasts about future relevance or lack thereof – we shift the focus towards modern *machine learning* approaches and attempt to build a short term price prediction framework geared towards high frequency trading strategies.

Our model employs *long short-term memory* cells in a *recurrent neural network* trained over historical high frequency data to classify one-period-

ahead forecasts of the market returns. Two classifiers are built, one designed to label the *directionality* of future returns, the other focused on capturing their *magnitude*. The data is sourced from the GDAX cryptocurrency exchange and the analysis is carried on the historical exchange rates of *Bitcoin*, *Ethereum* and *Litecoin* against both *EUR* and *USD*, for a total of six pairs. Statistical analysis over the forecasted time series shows promising significance in the predictive power of the model in terms of volatility.

Chapter 2 will introduce cryptocurrencies and cryptocurrency markets, providing the context upon which the modeling ideas have been formulated.

Chapter 3 will introduce the general principles of *deep learning*, details on the techniques and fundamental ideas employed in our work; namely, the usage of *long short-term memory recurrent neural networks* to discover patterns in sequential information.

Chapter 4 will present the forecasting framework, the hypotheses it is built upon, and elaborate on the choice of model hyperparameters and formats and time horizons of data.

Chapter 5 will discuss the results obtained in terms of predictive power by each of the classifiers built, and draw some conclusions upon the appropriateness of the model and features of the data.

2

CRYPTOCURRENCIES

2.1 HISTORY

Literature introducing the idea of applying cryptographical algorithms to payments dates as far back as the early eighties. Chaum (1982) raised the issue of conflict between personal privacy and criminal activity in the context of electronical banking payments that were becoming popular [6] and proposed, along with Fiat and Naor, a model for untraceable payments [7]. Ever since, a number of attempts were made towards the development of the concept, which mostly remained in the form of prototypes or academic papers until the publication of the paper *Bitcoin: A Peer-to-Peer Electronic Cash System* (Satoshi Nakamoto, 2008) [23]. *Bitcoin* had the innovative feature of being a *decentralized* form of digital currency: based on a *peer to peer* computer network ran by the users of the currency themselves, transactions' proof of legitimacy was sharded across every node of the network, simultaneously rendering unilateral attacks infeasible, and central, trusted financial intermediaries superfluous. This appealed to those more critical of the established financial system, which was heavily centralized and suffering the immediate aftermath of the subprime crisis in terms of credibility and trustworthiness.

Around 2013, possibly as a disgruntled response to the European banks' bailouts, *Bitcoin* started appearing under the spotlight of the media, and its prices began to soar, moving from a value of as little as $10 to as much as $1'000 over the course of the year. Over time, as *Bitcoin* piqued the interests of retail investors and amateurs first, and governments and financial institutions next, a large number of cryptocurrencies began to surface, ranging from shady Ponzi schemes to genuine technical innovations improving upon the state of the art. Among those, particularly noteworthy is *Ether*, the cryptocurrency of the distributed *smart contract* platform *Ethereum*.

As of early 2018, *Bitcoin* is by and large the cryptocurrency with highest market capitalization, estimated around $150 billion, its price exceeding the $17'000 mark in late 2017. *Ethereum* follows suit, nearing a $100 billion market cap for a price of around $1'000 and, after that, the *Ripple* cryptocurrency holds a market cap of around $40 billion on a price of around one US dollar per unit[2].

2.2 BACKGROUND AND DEFINITION OF CRYPTOCURRENCY

To illustrate how a cryptocurrency works, it is useful to take a step back and consider how an exchange of value normally happens in society. Barter, the most primitive form of it, is entirely transparent to the parties involved, as the exchange of a good for another requires nothing more than the physical possession of the good exchanged by each of the parties, and their mutual agreement on the transaction. A significant, intrinsic issue with barter is the fact that it may be hard to match supply to demand, as not all parties involved may own some goods that all the other parties are willing to trade with their own goods: if Alice was willing to trade their cattle for grains, and Bob was willing to trade his grains for salt, no transaction would happen unless there was a third party Carlos willing to trade both grains for cattle and salt for grains.

In order to make up for the absence of a third party enabling the transaction to happen, the practice of *credit* can take place: the exchange of goods may be replaced, on one side, by contracting *debt*, that is, promising that the value exchanged will be provided at a later point in time. As beneficial to transactional efficiency as they are, credit-based systems require either party to endure the risk that the counterparty may default on their debt.

Eventually, as human societies grow in size, the concept of *cash* tends to emerge. Cash represent, essentially, a form of certificate of ownership of value, in whatever shape and form may it be, and, *provided the general consensus upon its validity* (as the paper cash is printed upon doesn't have any intrinsic worth), allows transactions to happen without a bidirectional exchange of goods or services. In the previous example, Alice would just have to own cash (obtainable by selling their cattle to any third party willing to buy it) in order to buy Bob's grains. In order to come into shape, a cash based economy needs some form of *bootstrapping*: the emission, at inception, of a certain amount of cash, necessary for the subsequent trading activity to occur. Another fundamental feature of cash is *anonymity*: once cash exchanges hands, it is impossible to unequivocally

trace its provenance. There is also no necessity of intermediaries to carry the trades, as all that is involved is to actually physically exchange the cash for the goods or services provided.

As a further evolution, our current economic system relies, by and large, upon the intervention of *banks* and *central authorities*, who take on the role of trusted intermediaries in the fulfillment of financial matters and agreements, which is valuable whenever transactions are not as trivial as exchanging cattle for grains. Fiat money is printed, distributed and regulated by said central authorities.

Although no formal, agreed-upon definition exists, it is helpful to describe a cryptocurrency as a system which meets the following criteria[17]:

- *Distributedness of consensus*. The state of the system is not determined by a single party: all parties must agree on the current state of the system at any given time, either by trusting other parties or by actively contributing to the construction of new state.
- *Tracking of ownership*. The system must account, at any given time, for the ownership of units of currency – not necessarily by linking units of currency to their physical owners, but by at least defining some actor in the system to which currency units can be linked, e.g. a "*wallet*".
- *Currency minting rules*. The system must specify and enforce conditions, rules, amounts and initial ownership for the process of minting new coin.
- *Cryptographical proof of ownership*. As public-key cryptography algorithms are robust and reliable, it is possible to employ them as a necessary and sufficient form of proof of ownership by verifying digital signatures via public key.
- *Transaction issuance by owners*. Only the owners of currency have the right to issue transactions with their own currency, and they are able to do so by digitally signing the transactions with their private key.
- *Uniqueness of transactions*. If more than one transaction refers to the same unit of currency, all but one of those transactions must be considered invalid; i.e., *double spending* of the same unit of currency is prohibited.

Each of these features may be compared to the state of our current financial landscape to account for similarities and differences and understand the innovations brought by cryptocurrencies.

Requiring a *distributed consensus* to function is a characteristic cryptocurrencies have in common with paper money, and a robust one: if a few individuals reject the notion of paper money, they will simply be excluded from participating in the economy without posing any risk to its correct functioning, but can and will be allowed back in as soon as they decide to accept paper money again. For a paper money economy to be overthrown, the largest amount of its current participants must agree to some new course of action. This intrinsic robustness falls apart in larger societies, whenever banks and central authorities attain any form of discretionary power over the money: the trust is no longer spread over the whole community, but has now to be largely focused towards a small number of actors, thus creating a single point of failure. It could be argued that bank bailouts, ill-received by the general society, may have promoted interest in cryptocurrencies' decentralized trust features, although little in the way of research offers insights on the matter[19].

Tracking the ownership of every unit of currency is a fundamental feature of any and all economic systems, without which the system cannot function at all, and it is a feature for which the system must provide adequate guarantee. Paper money and barter have it embedded, as the physical possession of either cash or goods is sufficient to prove ownership. Credit relies instead on the good faith of the debtor or on some form of intermediation, thus introducing a form of risk for the individual participants in the transaction. Banks constitute a different configuration altogether: to rely on them means to rely on the accuracy of their bookkeeping operations and on the robustness of their databases. As much as excellent technology and processes may be deployed, centralization introduces again a source of fragility[26]: a single bank will always face some technological or operational risk that may jeopardize a large number of transactions and destroy customers' value. Cryptocurrencies solve this issue by maintaining a *distributed ledger* of transactions whose integrity relies on cryptography and, again, the decentralization of consensus.

Currency minting rules again draw a line between the centralized approach and the cryptocurrency approach. While the ultimate responsibility in terms of the former lies with the authorities, in cryptocurrencies the rules are established by the design of their software, are known beforehand and can't be circumvented as the software itself enforces them – it is not possible to issue a fraudulent currency-minting transaction as the

network would simply reject it, and this constitutes an advantage over paper money which is exposed to forgery risk. Generally, those rules for a cryptocurrency contemplate some form of *scarcity by design*, such that the presence of hard limitations allows the coin to gain value. *Bitcoin*, for instance, has an asymptotic ceiling of 21 million units.

Cryptographical proof of ownership and *limitation of transaction rights to currency owners* can, on principle, be attained both by banking based systems and by cryptocurrencies. As it is the worldwide industrial standard, public-key cryptography is also heavily employed in banking for these (and other) security purposes, and while decentralization might provide a further means of tamper prevention, cracking public-key cryptography is already widely regarded as an infeasible endeavor.

Uniqueness of transactions is the context where a centralized system definitely presents an advantage over a distributed system. The *double spending* issue is hard (but not impossible) to tackle computationally, whereas centralizing the responsibility of validating transactions intrinsically solves it. If the bank facilitating the transaction declares some transaction to be valid, that declaration is authoritative, provided accurate bookkeeping, and can be trusted. On the other hand, if a node of the distributed network receives only one of a couple of transactions spending the same amount of currency, and another node receives the other one, the two nodes must find some algorithm to decide for only one of them to be valid. In computer science, this is termed the *Byzantine Generals problem*[16].

2.3 THE BLOCKCHAIN

To implement the aforementioned features, cryptocurrencies rely on a data structure called *blockchain*. On practical terms, the *blockchain* is a public ledger of transactions whose cryptographical properties, coupled with the distributedness of consensus, concur in guaranteeing its validity. Technically, a *blockchain* is a *linked list* (i.e. a "chain") of arbitrary blocks of information, inherently sequential in nature – just like financial transactions are. Starting from an authoritative block, called the *genesis block*, another block can be created by first crafting the operational information it must contain (i.e. a list of transactions), then including a reference to the previous block, and finally signing the whole block with an expensive computational operation called *proof of work*. As the new block is accepted by the network, the next block will have to reference it for the *blockchain*

to keep growing, and so the ones after that[1]. The procedure of validating a block is referred to as *mining*.

The reference to a previous block is done by computing a *cryptographic hash* of said block and including it in the new block. Cryptographic hashes are easy to verify, but nearly impossible to forge: finding a *collision*, that is, a set of different data with the same hash, is computationally infeasible for all intents and purposes, so a hash guarantees correct reference to a block.

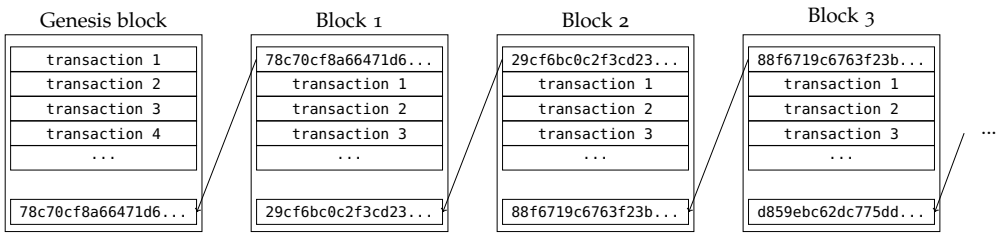

Figure 1: Simplified representation of a *blockchain*. Each block contains a list of transactions and refers to its parent block via its *hash*.

The *proof of work*[11] is a very important requirement for the reliability of the cryptocurrency, as it prevents *Sybil attacks* from happening. A *Sybil attack*[9] entails the forgery of many identities in a distributed network in order to gain control of it. If the signing of a block was instantaneous, one could use a large amount of fake identities to *mine* an equally large amount of blocks and, by impersonating more than half of the whole network, gain implicit authority and control over the *blockchain*. The *proof of work* is a computationally expensive algorithm which is feasible to execute as one instance, but which would be impossible to execute in a number of parallel instances high enough to cover more than half of the network. This way, genuine, well-intentioned individual nodes can perform their work by expending a relatively modest amount of computing power, while attackers have no means of subverting the network. Furthermore, its output has to be easily verifiable, and again, hashing functions are an appropriate candidate. As an example, *Bitcoin*'s *proof of work* algorithm[23] requires a *nonce* (i.e. a number) to be computed and added to the block, such that the numerical representation of the hash of the block is smaller than a given threshold – the smaller the threshold,

1 If two blocks reference the same block as predecessor, a *fork* happens, the *blockchain* splits in two and the network has to take a decision as for which of the two branches to accept or reject. The way this happens is implementation-specific and outside the scope of this work.

the harder it is to find the proper *nonce*. Furthermore, considering that no information can be extracted from computing a hash, it is also not possible to speed up the retrieval of the *nonce* in any way, requiring it to be sought by brute force alone.

To incentivize validation of transactions and creation of blocks, *miners* are rewarded with the emission of new currency; at the time of writing, 12.5 BTC are awarded to whomever "discovers" a new block, and the chance to do so largely depends on how much computing power is available to the single miner[2].

Blockchains need not be restricted to encapsulating financial information; any kind of information that may require timestamping and approval by general consensus may constitute a good use case for the blockchain. The *Ethereum* platform, for instance, relies on the blockchain to store and enforce *smart contracts*, which are essentially software protocols describing some form of negotiation or contract. Interestingly, both *Ethereum* and *Bitcoin* are Turing complete, hence virtually any form of *smart contract* may be deployed.

2.4 CRYPTOCURRENCY MARKETS

One of the fundamental practical issues concerning the adoption of cryptocurrency for real world transactions is value stability. As of the time of writing, alongside by participating in the mining process, cryptocurrency can be acquired in exchange for fiat money via trading in digital exchanges; from this trading activity, a valuation may be inferred for each cryptocurrency, both in relation to various fiat currencies and in relation to each other. What is apparent to active market participants and external observers alike are the exceptional swings in value that brought *Bitcoin* from a value of around $10 up to $10'000 over the course of five years. It is debatable whether cryptocurrencies may gain traction as a means of payment in the foreseeable future, when extreme variations in the daily prices may discourage both sellers to part from something that may be much more valuable tomorrow, and buyers to acquire something that may be worth much less tomorrow.

As it stands, perhaps the most profitable endeavor towards cryptocurrencies could lie in speculation; both as a means of attaining financial returns for the individual, and as a means of aiding market efficiency with

2 *Mining pools* can be organized, i.e. groups of nodes working together towards mining a block to then split the reward according to each node's individual contribution in terms of computing power.

the outlook of collectively discovering a long term fair price, thus promoting adoption of cryptocurrencies for real world use cases. It is hence of interest to make a few observations on the features of cryptocurrency markets.

Typically, the daily returns of a cryptocurrency are much more volatile than more liquid, traditional assets, and are very non-normal[5].

It has been observed that cryptocurrency prices correlate with sentiment, when measured via Google Trends data[21].

When considering pairs with a single fiat currency, different cryptocurrencies' returns tend to be highly correlated with each other, possibly owing to the imbalance in market capitalization – as *Bitcoin* is arguably the most popular cryptocurrency, when volatility events happen in consequence to sentiment, other cryptocurrencies tend to follow suit.

In terms of market microstructure, peculiar behaviours are observed as well. While the traded volume is generally high, many orders tend to be matched fairly frequently, and the bid-ask spread stays at the one-cent level (both for EUR and USD markets) most of the time, the distribution of liquidity is unusual: order volume tends to cluster at prices relatively distant from the best bid and ask, and market-taking orders of all but the smallest volumes inevitably incur in heavy slippage. When medium to large market orders are fulfilled, the bid-ask spread widens to tens, when not hundreds, of the paired fiat currency – apparently for *Bitcoin*, less so for the other, less valuable cryptocurrencies, but still high enough to disrupt efficiency. The likely cause of these peculiarities lies in the fact that the bulk of participants to these markets are non-professional retail investors endowed with very small capitals, trading on sentiment, their afflux promoted by the very low barriers to entry and mainstream popularity of the practice.

As a consequence of observing the uncommon conditions of these markets, we speculate that some patterns may emerge from the intraday pricing data that could allow for some form of forecasting. As linearity may prove too restrictive a function space, we seek for nonlinear relationships instead, by means of *deep learning* techniques.

3

DEEP LEARNING AND RECURRENT NEURAL NETWORKS

This chapter will provide an overview of the fundamental concepts of deep learning, and details on the specific models and optimization techniques employed in the work described in *Chapter 4*.

3.1 SUPERVISED LEARNING

Supervised learning is the class of techniques geared towards the usage of known inputs and outputs in order to find a function, along with its parameters, that approximates correct, unknown outputs given new, unknown inputs. Perhaps the simplest example of this is constituted by linear regression: given a matrix $\mathbf{X} \in \mathbb{R}^{n \times p}$ of n observations of p different variables (called *independent*) and a vector $y \in \mathbb{R}^n$ of observations of yet another variable (called *dependent*), the objective is to find a *parameter vector* $\beta \in \mathbb{R}^p$ which minimizes an unobservable *error vector* $\varepsilon \in \mathbb{R}^n$ in the equation

$$y = \mathbf{X}\beta + \varepsilon \tag{3.1}$$

so that *predictions* could be made in the form of

$$\hat{y} = \hat{\mathbf{X}}\beta \tag{3.2}$$

where $\hat{\mathbf{X}}$ is a different matrix of independent variables observations and $\hat{y}$ is yet unknown.

More generally, any supervised learning algorithm is composed of a function, a parameter space for it, a source of data (the *training set*) made of **(input, output)** couples, and some target function to be optimized over the parameter space.

Intuitively, given one input and the associated expected output, the parameters are "learned" for a function which will yield values as close

as possible to the expected output, and it is supposed, when exposed to unknown input, to yield a value as close as possible to the correct, but yet unobserved, output.

The forecasting problem studied here has been framed in such a way as to be fit for a supervised learning algorithm: given past price data up to one period before the present as input, and current price data as output, learn the parameters which will allow a reasonably significant one-period-ahead forecast to be made over new price data; the focus on *deep learning* techniques is based on the hypothesis that nonlinear models are more adept at obtaining significant performances over linear models.

3.2 NEURAL NETWORKS FUNDAMENTALS

Neural networks are borne out of the seminal idea, by McCulloch and Pitts, of a mathematical computation framework imitating the behavior of biological neurons [22]. A neuron fires a current along its *axon* through its *dendrite* after receiving solicitation from its *synapses*; what determines whether the dendrite will fire is some "strength" attribute of the synapses connected to it and the amount of current each synapse, coming out of some other axon, is carrying. The way a neuron "learns" is by altering the "strength" attribute of each synapse, reinforcing the connections that yield desirable results and weakening those that give negative contribution. Mathematically, this results in a model of the form

$$y_k = \psi\left(\sum_{i=0}^{m} w_{ki}x_i + \beta_k\right) \tag{3.3}$$

Where:

- y_k is the axon's "current level", i.e. the neuron's *output* value
- ψ is the *transfer* or *activation function*, i.e. a function exhibiting a number of properties that allow it to act as a *threshold* over its input – that is, intuitively, make it so its output is some continuous form of a Boolean state of truthfulness or falsity, similarly to a logic gate
- The x_is, for $i \in \{0, 1, \dots m\}$, are the "current levels", i.e. numerical values coming from either other synapses, or some input source
- The w_{ki}s are termed the neuron's *weights* and represent the aforementioned "strength" attribute for each synapse entering the neuron; lower values will mean penalty in the contribution coming

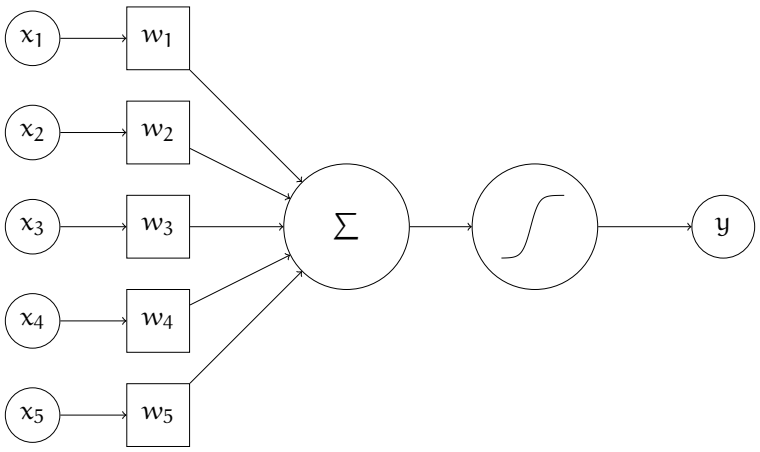

Figure 2: A *perceptron* with a 5-dimensional input and a *sigmoid* activation.

from a synapse to the input of the activation function, while higher values will reward previous activations coming from that synapse with a higher output value.

- β_k is a *bias* term

So, synthetically, a neuron's output is the activation value of the sum of its inputs weighted by that specific neuron's parameters – also called *propagation function*. One such unit of computation is also called a *perceptron*.

On a larger scale, each perceptron's axon is connected to the synapse of one or more other neurons, forming a layered network. Perceptrons in a neural network can be subdivided into three categories, according to their position in the network: they can either be on the *input* layer, on a *hidden* layer or on the *output* layer. The first of these three classes indicates the set of perceptrons whose input values are provided straight out of the training set; the last one indicates the set of perceptrons that emit output; finally, the second class comprises perceptrons that "communicate" to and from some other perceptrons. There may be zero, one or more than one *hidden* layers, and perceptrons may even form feedback loops or connect to earlier layers via *recursion* – this will be key for our model implementation. The attribute *deep* in "*deep learning*" refers to the presence of many hidden layers, as this fact can yield arbitrarily complex models capable of impressive feats, such as learning to play the game of Go from scratch well enough to beat some of the best human players [25].

A recursive traversal of the network from the input to the output layer yields our external model, that is, a function that takes as inputs the same

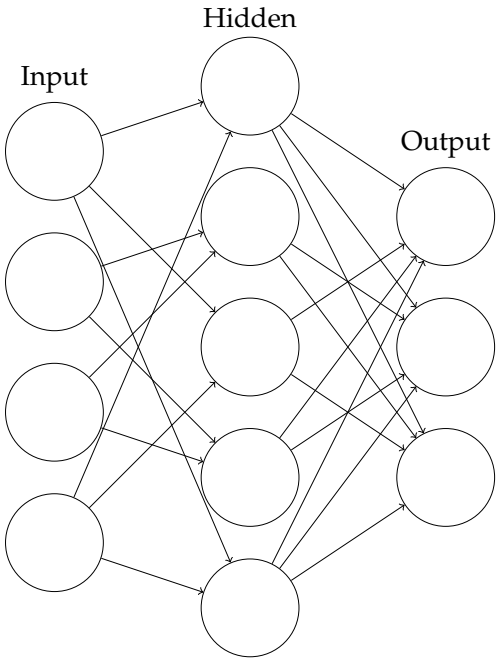

Figure 3: A neural network, endowed with a 4-dimensional *input* layer, a 5-dimensional *hidden layer* and a 3-dimensional *output layer*.

inputs as the input layer and yields an output in the same space as the output layer's.

3.2.1 *Optimizing the learning problem*

Let us define a generic neural network as a function of the form

$$f : X \to Y, \qquad f(x; \omega) = \hat{y} \tag{3.4}$$

where $\omega \in \Omega$ is a set of parameters, namely, the weights of each unit in the neural network's layer. We want the network to be able to predict the value $y \in Y$ for some given $x \in X$; in other words, we want f to be a function with parameters ω such that, if applied to x, yields a value $\hat{y}$ "as close as possible" to y, in some sense. A measure of the "closeness" of $\hat{y}$ and y is given by some *cost* or *loss function*. Without loss of generality, we may give the following definition, along with the desired properties, of *loss function*:

$$\mathcal{L} : Y \times Y \to \mathbb{R}, \quad \mathcal{L}(\hat{y}, y) = l \tag{3.5}$$

$$\mathcal{L}(\hat{y}, y) \geqslant 0 \quad \forall \hat{y}, y \in Y \tag{3.6}$$

$$\mathcal{L}(y, y) = 0 \tag{3.7}$$

that is, it should be zero only for the *exact* prediction, and greater than zero for any other value of $\hat{y}$. The definition may be relaxed by requiring a unique global minimum in (y, y), not necessarily taking value 0.

The loss function can be chosen according to the task at hand. For a regressive model, some function as intuitive as the *mean squared error* could be used:

$$\mathcal{L}(\hat{y}, y) \equiv \mathrm{MSE}(\hat{y}, y) = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2, \quad n = \dim Y \tag{3.8}$$

For classification tasks, other loss function are more appropriate, such as the *cross entropy* or the *Kullback-Leibler divergence*, and will be discussed in detail in the next chapter, as they pertain to the implementation of the model.

At this point, we are endowed with all the mathematical objects necessary to formulate the *learning optimization problem*

$$\min_{\hat{\omega} \in \Omega} \mathcal{L}(f(x; \hat{\omega}), y) \tag{3.9}$$

Solving this problem is not much different, in principle, than solving any convex optimization problem: if the cost function is continuous and differentiable, its minima or maxima may be found in correspondence of the roots of its first derivative. As x and y stay constant throughout one *learning* cycle, the loss function can be rearranged to highlight the fact that x and y are its parameters and ω is its domain, and thus rewritten in the form

$$\mathcal{L}(\hat{\omega}; x, y) \tag{3.10}$$

An important intuition arises from the fact that, if $\mathcal{L}$ is differentiable, then the quickest way of reaching a (local) minimum is by "walking" along the steepest descent direction; that is, by moving from the current point in the optimization cycle in the direction opposite of the gradient computed at that point.

This intuition leads to the *gradient descent* algorithm. Let $\nabla\mathcal{L}(\hat{\omega}_t; x, y)$ be the gradient of the loss function at the point $\hat{\omega}_t$. Then, the iterative step

$$\hat{\omega}_{t+1} = \hat{\omega}_t - \lambda \nabla \mathcal{L}(\hat{\omega}_t; x, y) \tag{3.11}$$

makes $\hat{\omega}$ converge towards the minimum for appropriate λ, and it holds

$$\mathcal{L}(\hat{\omega}_{t+1}; x, y) \leqslant \mathcal{L}(\hat{\omega}_t; x, y). \tag{3.12}$$

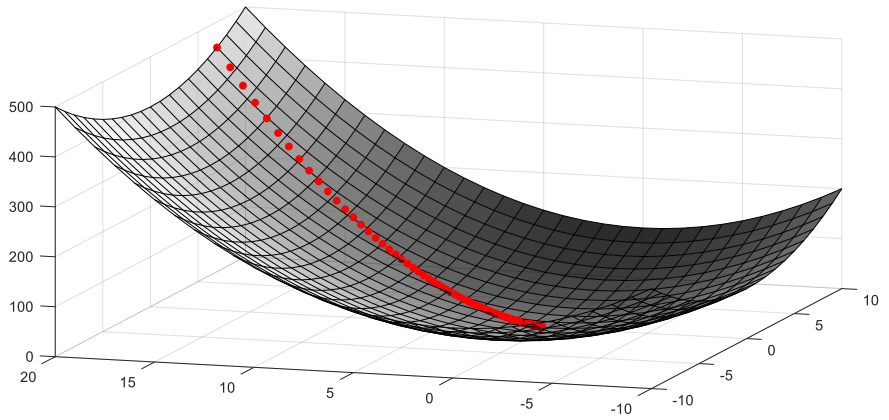

Figure 4: Gradient descent. Taking repeated steps against the gradient eventually leads to a local minimum.

The *hyperparameter* λ, called the *learning rate*, is one of the most important to evaluate for a neural network: it establishes a compromise between speed of convergence and accuracy. It is, fundamentally, a measure of "how long a step" will be taken in the gradient's opposite direction: too short, and the convergence will take a prohibitively long time; too long, and a minimum might be overshot by a "jump" on an opposing side of the slope and convergence may never be achieved.

A fundamental issue of "vanilla" gradient descent is that the gradient is evaluated for the entire training set, and this can be extremely computationally intensive and may render the optimization too slow to be of any use. A number of variants are available that progressively solve this and other issues that appear in the context of solving of the optimization problem. [24]

Stochastic gradient descent, for instance, computes the gradient, at each step, only for a single training example; this will lead to a much more frequent overshooting problem, especially around *ravines* (i.e. points where the surface has partial derivatives of very different magnitudes along different dimensions), but one that can be solved with appropriate choice of learning rate.

While generally the learning rate is set as a constant before starting the learning process, a better approach is to let it vary according to the context at each step. *Adagrad* [10] alters the learning rate for each of the model's parameters at each time step. Let $G_T \in \mathbb{R}^n$ be a vector of the

sum of the squares of the gradients of each parameter for the first T time steps; that is,

$$G_{T,i} := \sum_{t=1}^{T} \left(\frac{\partial \mathcal{L}(\hat{\omega}_t; x, y)}{\partial \hat{\omega}_{t,i}} \right)^2, \quad i \in 1, \dots n \tag{3.13}$$

where n is the dimension of the parameter space. Then, one *Adagrad* step is defined as

$$\hat{\omega}_{t+1} = \hat{\omega}_t - \frac{\lambda}{\sqrt{G_t + \epsilon}} \cdot \nabla \mathcal{L}(\hat{\omega}_t; x, y) \tag{3.14}$$

where ϵ is a positive term of negligible magnitude (perhaps in the order of 10^{-8}) that has the practical usefulness of avoiding divisions by zero. While this greatly increases the robustness of the optimization, there is a sharp monotonical decrease in the learning rate that may push it to too low a value, thus preventing convergence; this decrease derives from the fact that the gradient dampening rate is growing monotonically, as it is the sum of squared values.

To solve this issue, *RMSprop* and *Adadelta* were developed and focused on employing an exponentially weighted moving average of the square of the gradients as dampening rate, letting the power of earlier iterations decay over time.

$$G_{t,i} := \beta G_{t-1,i} + (1 - \beta) \left(\frac{\partial \mathcal{L}(\hat{\omega}_t; x, y)}{\partial \hat{\omega}_{t,i}} \right)^2, \quad i \in 1, \dots n \tag{3.15}$$

As a further improvement, *Adam* [15] includes an exponentially weighted moving average of the non-squared partial derivatives, introducing two terms

$$G^m_{t,i} := \beta_1 G^m_{t-1,i} + (1 - \beta_1) \left(\frac{\partial \mathcal{L}(\hat{\omega}_t; x, y)}{\partial \hat{\omega}_{t,i}} \right), \quad i \in 1, \dots n \tag{3.16}$$

$$G^v_{t,i} := \beta_2 G^v_{t-1,i} + (1 - \beta_2) \left(\frac{\partial \mathcal{L}(\hat{\omega}_t; x, y)}{\partial \hat{\omega}_{t,i}} \right)^2, \quad i \in 1, \dots n \tag{3.17}$$

and the corresponding *update rule*

$$\hat{\omega}_{t+1} = \hat{\omega}_t - \frac{\lambda}{\sqrt{G^v_{t-1} + \epsilon}} G^m_{t-1} \tag{3.18}$$

We found *Adam* to be the best fit for our work as it had better performance on all the data sets considered.

3.2.2 *Backpropagation*

In order to implement gradient descent, or any of its variants, it is necessary to have a way of efficiently computing gradients. Consider the formalism of *computation graphs*: one can represent *operations*, that is, functions $f : \mathbb{R}^n \to \mathbb{R}^m$ as directed edges, and *values* (e.g. the input $x \in \mathbb{R}^n$ and the output $y \in \mathbb{R}^m$ obtained by applying $f(x)$) as nodes. This way, it is easy to represent elaborate webs of function compositions, as neural networks are.

With this representation in mind, let us consider the chain rule for derivatives:

$$\frac{df \circ g(x)}{dx} = \frac{df(y)}{dy}\frac{dg(x)}{dx} \tag{3.19}$$

where $f : Y \to Z$, $g : X \to Y$ are differentiable functions and $X, Y, Z \subseteq \mathbb{R}$. Generalizing to higher dimensions, it holds

$$J_{f\circ g} = (J_f \circ g)J_g \tag{3.20}$$

where J_* is the Jacobian matrix. Intuitively, the chain rule allows us to simplify the computation of the derivative of a mathematical object as complex as a neural network, provided it can be factorized as composition of simple functions.

At this stage, the graph representation becomes useful as it is the base for the differentiation algorithm known as *backpropagation*. The algorithm requires a first step, the *feedforward*, to happen; this consists of evaluating the objective function, namely, the loss function $\mathcal{L}(f(x;\hat{\omega}), y)$ endowed at time t with the set of parameters $\hat{\omega}_t$, over some training data (x, y). This happens via a *breadth-first traversal* of the computation graph, i.e. computing all the innermost composed functions first. Once the scalar value of the final function $\varepsilon = \mathcal{L}(\hat{y}, y)$ has been computed, it is possible to compute the Jacobian of the outermost function (e.g. $J_{\mathcal{L}}$) and, following the reverse order of the breadth-first traversal (hence the term *backpropagation*), recursively multiply it by the Jacobian of every other operation. From the chain rule follows that the gradient of the web of functions has been found.

It is worth noting that employing a differentiable activation function at the nodes is what makes it possible to analytically compute partial derivatives beforehand from which a Jacobian matrix can be formed.

Linearity is a sufficient, but not necessary, condition for this to happen, and generally undesirable at that: as linear functions map to matrices,

and linear function composition maps to matrix multiplication, a neural network endowed with only linear operations would reduce to a single matrix and, hence, to a linear operation, drastically reducing its usefulness. Employing non-linear differentiable activation functions at the nodes, such as the *hyperbolic tangent*, the *logistic function* or the *rectified linear unit*, allows for a much broader mapping from neural networks to functions spaces.

The backpropagation process begins with an initial, randomly chosen set of weights, i.e. choosing a random point in the parameter space from where to start the gradient descent. The gradient is then evaluated for the loss function computed over every *training example*, and a step proportional to the *learning rate* is taken against it. Once the process is iterated over all the training examples, an *epoch* is over. Another *epoch* can begin, starting from the point in the parameter space the preceding epoch ended at. The number of epochs is a *hyperparameter*: more epochs mean slower training, but potentially more accurate results.

3.2.3 *Overfitting: regularization and cross-validation*

The overarching objective of a machine learning algorithm is to provide an *estimator* to be used with unknown data after having learnt its parameters via known data. In other words, the desired outcome is *generalization*. It may happen that this is not the case, and a very low loss achieved on the training data does not correspond to a low loss on predicting unforeseen data. This situation is termed *overfitting*. As an example, consider an algorithm that tries to fit a very high degree polynomial through a few points lying on a simple parabola; it may well be that the high degree polynomial passes through all of the training points, but the prediction outcome is not the one desired. On the other end of the spectrum, there is the phenomenon of *underfitting* which happens when the model barely fits the provided data at all – consider the problem of trying to fit a straight line through the same points on the parabola. The *hyperparameter* of *capacity* may be introduced here: it may intuitively be defined as the "range" of functions that the algorithm is allowed to learn. For example, restricting the grades of the polynomial in the problem just mentioned to a range greater than one and less than five is bound to yield much better and more accurate results.

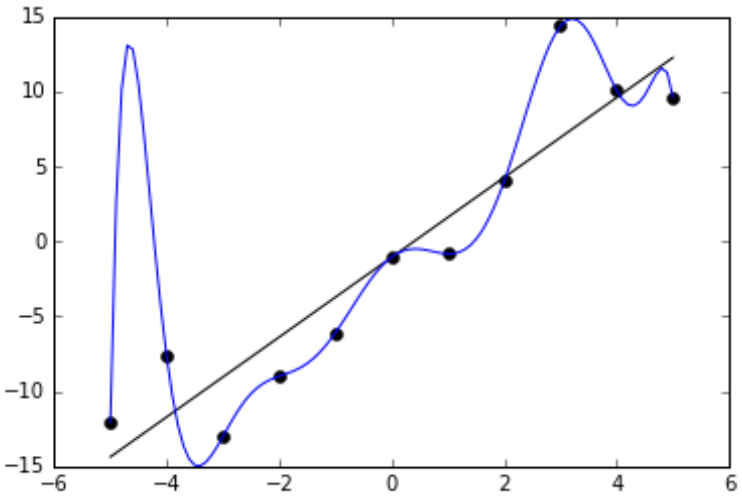

Figure 5: An example of *overfitting*. Wikimedia Commons, CC-BY-SA-4.0

Norm regularization

The problem of restricting the function search space to a subspace of simpler functions may be formalized by introducing a penalty term on some measure of the "complexity" of the function, and adding it to the *loss function* before performing gradient descent. In practice, as there is strong dependence between the neural network function f and its weights ω, the L^1 and L^2 norms over Ω are employed as regularization terms.

$$\|\omega\|_1 := \sum_{n=1}^{\dim \Omega} |\omega_n| \tag{3.21}$$

$$\|\omega\|_2 := \left(\sum_{n=1}^{\dim \Omega} |\omega_n|^2 \right)^{\frac{1}{2}} \tag{3.22}$$

Given some discretionary hyperparameter $\delta \in \mathbb{R}$ representing some desired "strength" of the regularization, we can reformulate the *learning problem* with (e.g.) L^2 regularization as

$$\min_{\hat{\omega} \in \Omega} \mathcal{L}(\hat{\omega}; x, y) + \delta \|\hat{\omega}\|_2 \tag{3.23}$$

Generally, the L^2 norm is chosen as it has the advantage of being differentiable, hence possessing analytical partial derivatives that allow for a faster numerical computation of the gradient. As the neural network is akin to a function composition graph, the regularization may also be applied to a subset of the weights pertaining to a specific layer or unit.

Dropout regularization

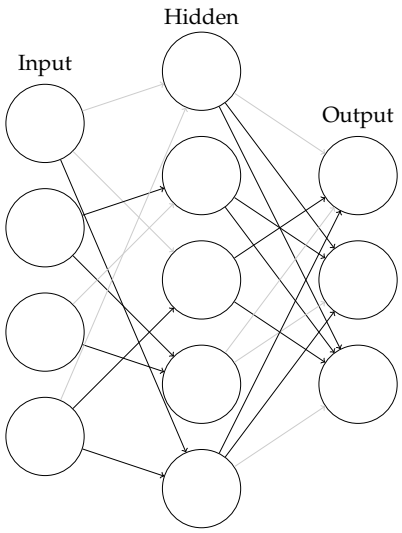

Figure 6: An example of *dropout regularization*. In gray, edges which are being temporarily disabled over the course of a training operation.

As a further means of reducing overfitting, the technique of *dropout* is implemented. A dropout of magnitude p, $p \in (0, 1)$ indicates that every edge between nodes in the network has a probability p of being severed during any gradient descent step. This makes it so not all network nodes are trained against all data, leading to an increased robustness in the features learned by each node, and a consequent better generalization.

Cross-validation

In order to provide a means of quantitatively evaluating underfitting or overfitting issues, it is useful to split the data set initially into two subsets, the *training* and the *test* set. Running the network against the test set, one can check whether the magnitude of the error is satisfactory. Generally, the error evaluated on the training set constitutes a lower bound for the error obtained on the test set.

As a means of providing an indicator for automatically tuning capacity (among other hyperparameters), it is useful to further subdivide the data set and reserving a portion of it as *validation set*; no training is done over validation data as well, but errors and other metrics may be computed on it in order to "steer" the learning process and reducing the occurrence of both overfitting and underfitting. This process is called *cross-validation*.

3.3 RECURRENT NEURAL NETWORKS

While vanilla feed-forward neural networks constitute an instrument adept at solving a broad class of problems, the task of price prediction via past observations is perhaps best addressed by a specialization of the model in the form of *recurrent neural networks*[14]. Recurrent networks are, in their structure, geared towards the processing of *sequential* data; that is, data where a substantial amount of information is delivered not only by the values in and of themselves, but also by their relative ordering and positioning. Without loss of generality, we could define sequential data as data carrying information in the form of positional dependency with past data.

Consider a discrete *dynamical system* of the (general) form

$$p_{t+1} = f(p_t, x_t; \omega) \tag{3.24}$$

Such a system, parameterized by a parameter vector ω, yields the next value of a sequence (in the mathematical sense) as function of both some endogenous *context*, i.e. the point in the sequence's space the system is at at time t, and some exogenous vector of factors x_t (likely, but not necessarily, itself time-dependent). It is in this form that most phenomena linked to variation over time can be modeled.

The definition, quite apparently, shows *recursion* over the endogenous input: that is, by repeatedly applying the definition we can explicitly express dependence between the current state and some endogenous state farther back in time.

$$\begin{aligned}
\mathbf{p_{t+1}} &= f(\mathbf{p_t}, x_t; \omega) \\
&= f(f(\mathbf{p_{t-1}}, x_{t-1}; \omega), x_t; \omega) \\
&= f(f(f(\mathbf{p_{t-2}}, x_{t-2}; \omega), x_{t-1}; \omega), x_t; \omega) \\
\cdots &= (f \circ f \circ \cdots \circ f)(\mathbf{p_0}, x_t \dots x_0; \omega) \\
&=: f^{(t)}(\mathbf{p_0}, x_t, \dots x_0; \omega)
\end{aligned}$$

It is upon this "unrolling of dynamics" principle that recurrent neural networks operate. One immediate, practical simplification with respect to the above abstract model is that while a dynamical system can exist since a "negative infinity" time and until a "positive infinity" time, computing resources are necessarily finite and a recurrent network won't be able to handle more than a finite number of iterations, thus driving some lossiness in the modeling process.

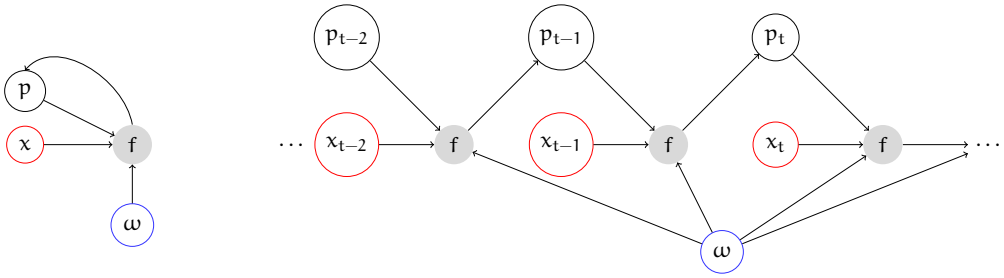

Figure 7: Graph of a recurrent neural network, in the implicit form (on the left) and in the explicit, unrolled form (on the right).

On the other hand, "unrolling" a finite, appropriate number of recursive steps allows us to represent the recurrent network as a directed acyclic graph, and capture the following properties of the abstract model (see figure 7):

- The sequentiality of the exogenous inputs
- The sharing across nodes of the parameter vector
- The temporal dependence on past values

Interestingly, by virtue of the presence of recurrence, the unlimited form of the model is *Turing complete*, that is, according to the *Church-Turing thesis*, it can simulate a Turing machine and consequently compute any computable function of the naturals. Notably, this is a property intrinsically deriving from recursive functions, not strictly from the fact that said functions are organized in the form of a neural network.

3.3.1 *Vanishing and exploding gradients*

One substantial drawback in terms of learning speed for a recurrent neural network lies in the sequentiality of the computation graph: one single recurrent layer maps to a high number of hidden layers when unrolled, and the intrinsic sequential dependence from one iteration to the next makes it so the breadth-first traversal behaves more like a *depth-first* traversal, on the grounds that the network is deeper than it is broad and there is limited potential for parallelizing the calculation of one step of gradient descent; formally, the backpropagation algorithm can't get any less complex than $\mathcal{O}(n)$ in the number of recurrent units.

Recurrent networks are affected by another significant issue arising from their structure, and related to convergence. Recall that, for the

network to fit functions beyond linearity, it is necessary to apply some non-linear differentiable function at the nodes, i.e. the *activation* function. Characteristic to most activation functions is the *saturation* property: informally, an activation function tends, for input values far from 0 at either side, to asymptotically converge at some value. Consider the example of the *logistic function* and its derivative:

$$S(x) := \frac{e^x}{e^x + 1} \tag{3.25}$$

$$\frac{dS(x)}{dx} = \frac{e^x}{(1 + e^x)^2} \tag{3.26}$$

$S(x)$ is monotonically increasing, takes values (as a gross approximation) in $(0.9, 1)$ for $x \gtrsim 2.5$ and in $(0, 0.1)$ for $x \lesssim -2.5$; straying too far from these small intervals of values for x would "flatten" the node's response. Most notably, the derivative exhibits a very limited codomain, $\left(0, \frac{1}{4}\right]$, taking maximum at $\dot{S}(0) = \frac{1}{4}$ and asymptotically converging to 0 for $x \to \infty$ and $x \to -\infty$.

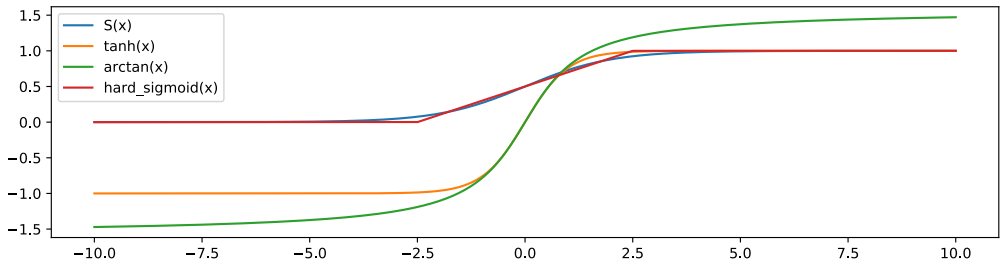

Figure 8: Some activation functions: sigmoid, hyperbolic tangent, arctangent and hard sigmoid. All of them exhibit some form of asymptotic *saturation* far from the origin.

Now, backpropagation through time in the case of an unfolded recurrent layer consists in repeated application of a gradient of the same form. In the case of the sigmoid, this would mean "constraining" each successive gradient's argument in a progressively narrower interval, such that at the end of the chain we'd be left with a gradient that has a value of approximately zero and which would not push the optimization any further at later backpropagation steps. This issue takes the name of *vanishing gradients problem*.

The vanishing gradients problem is very apparent in the presented example of the logistic function, but it applies generally to most other activation functions wherever there is a critical topology, i.e. a path in the

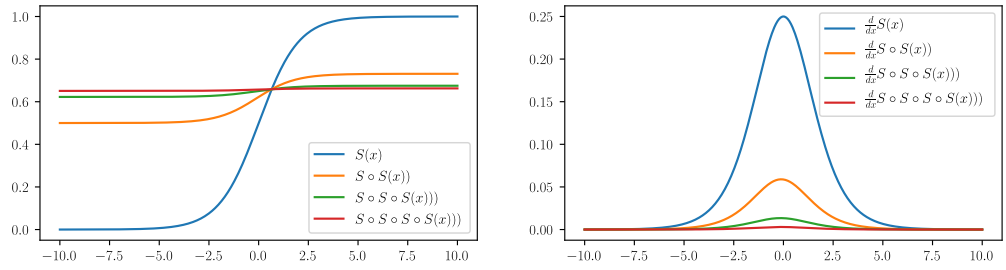

Figure 9: Vanishing gradients. On the left, the logistic function composed with itself up to four times; on the right, the respective four derivatives. As the function flattens to smaller codomains, the gradient quickly goes to zero.

recursive application of the gradient operator that leads to values near zero.

Albeit infrequent, the symmetrical *exploding gradients problem* may also happen; it would consist of gradient composition paths insisting on larger and larger values that would lead to extreme movements in the gradient descent thus completely disrupting the optimization.

3.3.2 *Long short-term memory*

In practice, what vanishing gradients would entail in the sequential context is that the weights would be optimized with much smaller gradients in response to the training examples farther back in the sequence and with larger gradients in response to the elements near the end of the sequence; this is because the backpropagation steps apply a large number of compositions for the former and a smaller number of composition for the latter. In the end, this means that what is modeled by the recurrent network is a much stronger dependence on recent elements of the sequence and a disregard for the older ones, i.e. long term dependencies aren't properly captured. Furthermore, this means that a recurrent network doesn't have the ability to detect relationships at larger time scales, but that is precisely what would be desirable in a nonlinear modeling framework for pricing time series, where a nonlinear longer term trend may bring as much information as the last few observations.

The vanishing gradient problem can be mitigated, for example, by severing a number of connections along the backpropagation path or by trying to process and alter such path. For example, if we had weaker recurrent connections to the immediately nearby units but added con-

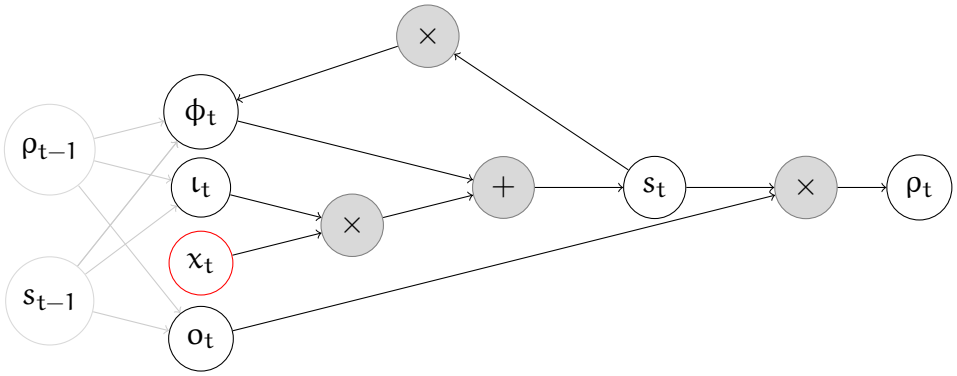

Figure 10: Graph of a LSTM cell at time t, representing the flow between input, output, forget gates (resp. ι_t, o_t, ϕ_t), proper inputs (x_t), internal state (s_t) and output (ρ_t).

nections to units farther back, we could overweight the importance of long term dependencies; this fundamental idea was introduced in the form of *leaky recurrent units* [12] [18] and developed as the concept of *gated recurrent units*; that is, recurrent units that govern the error flow via *hidden gates*, i.e. activations of weighted sums not much different from a vanilla artificial neuron, but executing special functions by virtue of their connections in the single recurrent unit. Of particular interest to our application is the *long short-term memory*, or *LSTM*, unit [13].

A long short-term memory cell (see figure 10) is endowed with an *input gate*, an *output gate*, a *forget gate*, a hidden unit and, most importantly, an *internal state* (denoted as s_t). The internal state spans the whole unfolded chain of LSTM units, net of some linear operations, thus allowing information to be stored in it, and flow through it, which doesn't strictly depend on a point in the sequence. So, each recurrent unit has three input sources, that is, the internal state, the proper input (denoted as x_t) and the previous cell's hidden output (ρ_{t-1}), and emits two outputs, that is, the (next) internal state and the cell's hidden output (ρ_t).

The *forget gate* can be intuitively defined as an activated sum of both the inputs and the hidden unit, each weighted by a different set of weights – the former set (ω) is conceptually the same discussed earlier, the latter (θ) is specific to this hidden unit, and both of them are specific to the forget gate. The output to this gate is in the range $(0, 1)$ where the left bound means "forget everything" and the right bound means "remember

everything"; an appropriate activation function for this task is the *logistic function* S.

$$\phi_t(x_t, \rho_{t-1}; \omega^{\phi_t}, \theta^{\phi_t}) := S\left(\sum_i \omega_i^{\phi_t} x_{t,i} + \sum_j \theta_j^{\phi_t} \rho_{t-1,j} + \beta_t^{\phi_t}\right) \tag{3.27}$$

The "forget" operation is performed by multiplying the internal state times the output of the forget gate. At the recurrence step $t+1$, the information coming from the internal state will be carried over intact from the state at time t if the forget gate outputs a "high" (i.e. close to 1) value and will be diminished if the gate outputs a "low" value.

The *input gate* has different semantics, but the form of its function is identical to that of the forget gate, and its parameters are its own:

$$\iota_t(x_t, \rho_{t-1}; \omega^{\iota_t}, \theta^{\iota_t}) := S\left(\sum_i \omega_i^{\iota_t} x_{t,i} + \sum_j \theta_j^{\iota_t} \rho_{t-1,j} + \beta_t^{\iota_t}\right) \tag{3.28}$$

Again, its output is in the interval $(0, 1)$ and it provides a weight for the activation of the weighted sum of the input values at time t to the network. Activation on the input needs not, in fact, necessarily be a sigmoid, but a different nonlinear operation can be chosen in its stead; a good alternative is the *hyperbolic tangent* tanh. The input activation as well has its own set of parameters $(\omega^{A_t}, \theta^{A_t})$:

$$A_t(x_t, \rho_{t-1}; \omega^{A_t}, \theta^{A_t}) := \tanh\left(\sum_i \omega_i^{A_t} x_{t,i} + \sum_j \theta_j^{A_t} \rho_{t-1,j} + \beta_t^{A_t}\right) \tag{3.29}$$

Up to here, all the objects have been defined which are necessary for the computation of one step of *internal state*, which can be expressed in the following form (arguments omitted for the sake of simplicity):

$$s_t = \phi_t \cdot s_{t-1} + \iota_t \cdot A_t \tag{3.30}$$

Informally, we can say that the next step of the internal state is computed as the sum of the gated previous state and the gated activated input, that is, as a "mix" of past, persistent information and new, punctual information.

The *output gate*'s form is no different from the other gates, and it is, similarly, used to filter its hidden unit's value, in order to compute the cell's output ρ_t:

$$o_t(x_t, \rho_{t-1}; \omega^{o_t}, \theta^{o_t}) := S\left(\sum_i \omega_i^{o_t} x_{t,i} + \sum_j \theta_j^{o_t} \rho_{t-1,j} + \beta_t^{o_t}\right) \tag{3.31}$$

$$\rho_t = \tanh(s_t) o_t \tag{3.32}$$

Overall, equations 3.30 and 3.32 represent the gist of the LSTM unit's functioning. As it can be expressed in the form of a graph, the backpropagation formula applies as previously discussed, the error propagating backwards from the final output ρ_t (for a LSTM endowed of t units) towards the first layer.

4

THE FORECASTING FRAMEWORK

The empirical work hereby presented is focused upon the employment of a *long short-term memory* deep neural network with the objective of evaluating its predictive power on high frequency cryptocurrency time series. The overarching idea is to test whether cryptocurrencies' extreme volatility drives a signal of inefficiencies which may be captured by leveraging the pattern recognition capabilities of deep learning models.

The methodology employed in order to test our hypothesis is as follows. First, the network is trained on a chosen data set, then one-period-ahead forecasts are generated from an out-of-sample data set. Statistical tests and measurements are then computed to assess the resulting model's predictive power.

4.1 HYPOTHESIS

The hypothesis will be articulated over two experiments:

- Predicting the *directionality* of prices, i.e. the *sign* of future returns
- Predicting the *volatility* of future returns, i.e. some point-wise measure of their *magnitude*

The testing will entail running an *out-of-sample* forecast and evaluating statistical classification measures (such as *precision*, *recall*, *specificity* and *accuracy*) in order to draw conclusions on the model's power for the task at hand.

4.1.1 *Directionality of prices*

We will frame our experiment as a *classification problem*. We want to use recent historical price data to forecast the probability of the following events:

- The price will rise by at least 0.2% over the next time interval ("*price increase*" outcome)
- The price will fall by at least 0.2% over the next time interval ("*price decrease*" outcome)
- The price will move by less than 0.2% in either direction over the next time interval ("*neutral*" outcome)

The motivation behind the choice of a three-label classification in lieu of a more intuitive binomial classification of positive versus negative returns is twofold. First, the employment of high frequency data is bound to show a high degree of noise around the origin: due to factors mostly arising from market microstructure, it is likely for the bulk of returns to have magnitude in the order of 10^{-3}, 10^{-4}, and as these kinds of magnitudes are relatively inconsequential in operational terms it makes little sense to separate the positive from the negative ones. Furthermore, such small returns prevent realizing a profit when accounting for transaction costs.

The threshold of 0.2% has been chosen as a reasonable compromise between the rarity of the event and the noisiness of the magnitude: returns in the order of 0.2% are rare enough at our chosen time horizon as to filter out noisy observations, but frequent enough not to produce largely unbalanced classes. Consider this extreme example: if we chose 50% returns as threshold, any network would achieve maximum accuracy, as the predicted label would be always be *neutral* due to the fact that all observed returns would fall within the (-50%, 50%) interval and all the network would have to learn would be to always output the *neutral* label regardless of the input.

We chose to conduct our analysis on a 5 minutes time horizon. This, too, is the outcome of a compromise between availability and noisiness of the data: shorter time intervals would carry almost only noise, while for longer time intervals the volume of data might be insufficient for the task of training the deep network.

Depending on the instrument considered, the 0.2% threshold allows us to label between 40% to 60% of the sample data as *neutral*, while the rest of the data is approximately evenly split between the *price increase* and the *price decrease* labels; very roughly, we have a $(25\%, 50\%, 25\%)$ split. The distribution of data yielded from this threshold is not overly balanced, but still reasonable.

We want to test whether *long short-term memory* based deep neural networks are able to correctly predict the direction of returns larger

than the threshold over short intraday time intervals. We will refer to variables and mathematical objects related to this experiment with the term "*directionality experiment*", and by labeling with a "d" superscript or subscript.

4.1.2 *Volatility of returns*

This experiment is framed similarly to the previous one. We want to forecast the probability of the following events:

- The price will move by at least 0.2% in either direction over the next time interval ("*high volatility*" outcome)
- The price will move by less than 0.2% in either direction over the next time interval ("*low volatility*" outcome)

The reasons for the choice of the 0.2% threshold are identical; the classes, instead, have been reduced to two on the grounds that adding a third one would either produce wildly unbalanced classes (possibly extremely skewed towards the *neutral* at high thresholds) or balanced but arguably useless classes at smaller thresholds – it is debatable whether any value could be extracted from separately identifying 0.1% and 0.2% returns.

We want to test whether *long short-term memory* based deep neural networks are able to correctly predict the occurrence of large returns over short intraday time intervals. We will refer to variables and mathematical objects related to this experiment with the term "*volatility experiment*", and by labeling with a "v" superscript or subscript.

4.2 DATA PREPROCESSING AND NETWORK STRUCTURE

The deep network used as a forecasting means implements a supervised learning algorithm based upon *long short-term memory* (*LSTM*) cells. The network accepts as input a three-dimensional tensor. Each element of the first dimension of said tensor is computed as a rolling window of past observations; namely, for the t-th forecast it is a matrix $M_t \in \mathbb{R}^{n \times m}$ where n is the length of the window and m is the number of parameters. The parameters chosen are five, that is, the simple returns computed on the *open, high, low* and *closing prices,* and finally the *trading volume* out of an observed 5-minutes candle. The window of those observations

defining the matrix M_t of the input for the prediction of the return on the closing price at time t is chosen as $(t-n, t-1)$, such that the matrix takes the following form:

$$M_t := \begin{bmatrix} O_{t-n} & L_{t-n} & H_{t-n} & C_{t-n} & V_{t-n} \\ O_{t-n+1} & L_{t-n+1} & H_{t-n+1} & C_{t-n+1} & V_{t-n+1} \\ \vdots & \vdots & \vdots & & \\ O_{t-2} & L_{t-2} & H_{t-2} & C_{t-2} & V_{t-2} \\ O_{t-1} & L_{t-1} & H_{t-1} & C_{t-1} & V_{t-1} \end{bmatrix} \tag{4.1}$$

$$M_t \in \mathcal{M}^{n\times m} \subseteq \mathbb{R}^{n\times m} \tag{4.2}$$

Each one of such matrices is called a *sequence*, as it is a set of elements ordered over time of which the network is supposed to learn to predict the successor.

The *training example* y_t is the *one-hot encoding* of the label of the return computed on the closing price at time t, that is, a probabilistic representation of the certainty that the (known) return C_t belongs to one of the three labels (*price decrease*, *neutral*, *price increase*) or one of the two labels (*low volatility*, *high volatility*), respectively, for the directionality and for the volatility experiment.

Let us define the *probability space* associated with our problem

$$(\Omega, \mathcal{F}, \mathbb{P}) \tag{4.3}$$

where Ω is the continuous *sample space* describing the possible values of the closing returns at time t, $\mathcal{F}$ is the σ-algebra of information available at every time t, and $\mathbb{P}$ is the probability function. We then define the *random variables*

$$\bar{y}^d : \Omega \to Y^d, \quad Y^d \equiv \{\textit{price decrease}, \textit{neutral}, \textit{price increase}\} \tag{4.4}$$

$$\bar{y}^v : \Omega \to Y^v, \quad Y^v \equiv \{\textit{low volatility}, \textit{high volatility}\} \tag{4.5}$$

as functions mapping from the *sample space* to the set of the events pertaining to each experiment.

We represent the classes in the general form of a vector describing a partition of the probability space. We have, respectively in the case of the

directionality experiment and in the case of the volatility experiment, the following events:

$$y_t^{d,T} := \begin{bmatrix} \mathbb{P}(C_t \leqslant 0.002|\mathcal{F}_T) \\ \mathbb{P}(-0.002 < C_t < 0.002|\mathcal{F}_T) \\ \mathbb{P}(C_t \geqslant -0.002|\mathcal{F}_T) \end{bmatrix} \tag{4.6}$$

$$y_t^{v,T} := \begin{bmatrix} \mathbb{P}(|C_t| < 0.002|\mathcal{F}_T) \\ \mathbb{P}(|C_t| \geqslant 0.002|\mathcal{F}_T) \end{bmatrix} \tag{4.7}$$

where $\mathcal{F}_T$ is the σ-algebra of information available in the *training* set, i.e. *after* the outcome has been observed; that is, with this process we label known past information. As an example, if the return C_t was below -0.2%, its label would be *price decrease* in the directionality experiment and *high volatility* in the volatility experiment, and the training examples would be equal to

$$y_t^{d,T} := \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad y_t^{v,T} := \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{4.8}$$

Do note that the event space for the volatility experiment can be derived from the event space for the directionality experiment simply by merging the classes *price increase* and *price decrease* into the single *high volatility* class, and the *low volatility* and *neutral* classes are equivalent.

The final network is akin to a function of the form

$$f : \mathcal{M}^{n\times m} \to \mathbb{R} \qquad f(M_t) = \hat{y}_t \tag{4.9}$$

where $\mathcal{M}^{n\times m}$ is the space of the matrices defined as of (4.2), and $\hat{y}_t$ is a partition of the probability space representing the forecast at time t, conditional on the preceding time steps; that is, of the forms

$$\hat{y}_t^{d} := \begin{bmatrix} \mathbb{P}(C_t \leqslant -0.002|\mathcal{F}_{t-1}) \\ \mathbb{P}(-0.002 < C_t < 0.002|\mathcal{F}_{t-1}) \\ \mathbb{P}(C_t \geqslant 0.002|\mathcal{F}_{t-1}) \end{bmatrix} \tag{4.10}$$

$$\hat{y}_t^{v} := \begin{bmatrix} \mathbb{P}(|C_t| < 0.002|\mathcal{F}_{t-1}) \\ \mathbb{P}(|C_t| \geqslant 0.002|\mathcal{F}_{t-1}) \end{bmatrix} \tag{4.11}$$

The predicted label, i.e. the *random variable* $\bar{y}_t^*$, is extracted from such partitions by choosing the index with the highest probability, i.e.

$$\bar{y}_t^d = \begin{cases} \textit{price decrease} & \hat{y}_{t,1}^d \geqslant \frac{1}{3} \\ \textit{neutral} & \hat{y}_{t,2}^d \geqslant \frac{1}{3} \\ \textit{price increase} & \hat{y}_{t,3}^d \geqslant \frac{1}{3} \end{cases} \tag{4.12}$$

$$\bar{y}_t^v = \begin{cases} \textit{low volatility} & \hat{y}_{t,1}^v \geqslant \frac{1}{2} \\ \textit{high volatility} & \hat{y}_{t,2}^v \geqslant \frac{1}{2} \end{cases} \tag{4.13}$$

From the presence of bounded activation functions in the layers arises the need to further transform the data to a suitable format. The *sigmoid* function, for instance, is bounded in the interval $(0, 1)$ and takes meaningful values in some neighborhood of the point 0; too far out in either direction, and it *saturates* quickly to 0 or 1, too close to zero and it yields little to no information.

The data needs to be distributed in such a way as to spread the available information throughout some useful range of the domain of the activation function. The untransformed 5 minutes returns time series falls into the second case, that is, it generally exhibits values in the order of magnitude of 10^{-2} to 10^{-4} which are unoptimal for the reasons explained above.

It is thus necessary to map the input domain to a wider interval near the origin so that the activation function may show desirable behaviour.

To do this, we need to define a *normalization* operation which should perform the mapping without applying any distortion to the information.

A very simple function which is suitable to the aforementioned needs is a form of the *feature scaling* function, defined this way:

$$G_x : \mathbb{R} \to \mathbb{R}, \quad G_x(x) = \frac{x}{\max |x|} \tag{4.14}$$

It is a linear transformation, parameterized by the maximum amplitude of the x series, that magnifies the magnitude properties of the input data to a scale which is better suited for the neural network's activation, but does not distort important information such as sign or distributional properties. Do note that the labels are to be computed *before* the feature scaling, as this doesn't preserve the magnitude of returns we are interested in. Also, the same normalization operation with the same width parameter is to be applied to out-of-sample data in order to provide consistency of the inputs.

The normalization and denormalization functions, along with their parameters, are computed separately once for each of the *open, high, low*

and *close* over the whole data set for a given pair; the *volume* column is left unaltered as it already exhibits desirable properties.

Once labels are computed and normalization is applied, the next step of data preparation involves building the actual data set to be used by the neural network, that is, computing and storing all the (M_t, y_t^*) couples for both experiments:

$$I^d := \left\{ (M_t, y_t^d) : t \in \{n, n+1, \dots N\} \right\} \tag{4.15}$$

$$I^v := \{ (M_t, y_t^v) : t \in \{n, n+1, \dots N\} \} \tag{4.16}$$

where N is the total number of entries in the data set; the first index in the set is taken at n as M_t is built with data going back n periods.

The data set will further be split into two chunks: the *training set*, that is, the set that will be used by the network for training its weights, and the *test set*, that is, a set that will be used for assessing the performance of the model after it has been trained. This split is fundamental as to avoid or at least mitigating the *overfitting* problem. Data up to midnight of December 31st, 2017 has been allocated for the training set, the remainder up to January 27th, 2018 for the test set. Training sets may show radically different sizes as some instruments haven't been available for trading over the whole time horizon considered, and some were illiquid enough as to have periods of inactivity for which no data was recorded.

A further split is performed on the training set, the last 20% of which is reserved as *validation set*. Internally, the neural network performs the *validation* operation at the end of each *epoch*: that is, it runs a prediction cycle on the validation set and evaluates the loss function. This is useful in evaluating the training progress and in establishing a condition for the early stopping of the process, as the training shouldn't be carried further if the network has already converged near a local minimum, since continuing may cause overfitting; this can be assessed by observing the value of the loss function on the validation data at the end of each epoch and checking whether a significant decrease, a stagnation or even an increase has taken place.

The neural network has a very simple structure:

- A *LSTM* layer endowed with 24 hidden layers, a 32-dimensional output space, the *hard sigmoid* [1] function as its recurrent activation and the *hyperbolic tangent* function as its input activation, L^2 regularized
- A *dropout* layer
- A *densely connected* simple perceptron layer with a 32-dimensional input space and a 3-dimensional (for the directionality experiment) or 2-dimensional (for the volatility experiment) output space, activated with the *softmax* function (resp. $\mathcal{S}_3$ and $\mathcal{S}_2$), also L^2 regularized.

The *softmax* function is a generalization to n dimensions of the *sigmoid* function. Intuitively, it is the normalization of a vector to which an elementwise *sigmoid* function has been applied. It is defined as

$$\mathcal{S}_n : \mathbb{R}^n \to \mathbb{R}^n_1, \quad \mathcal{S}_n(x)_j := \frac{e^{x_j}}{\sum_{i=1}^{n} e^{x_i}} \tag{4.18}$$

where $\mathbb{R}^n_1$ is the subspace of vectors in $\mathbb{R}^n$ having unit Euclidean norm, and exhibits the property

$$\sum_{i=1}^{n} \mathcal{S}_n(x)_i = 1 \quad \forall x \in \mathbb{R}^n \tag{4.19}$$

which matches the property of the probability space partitioning

$$\bigcup_{i=1}^{n} B_i = \Omega \implies \sum_{i=1}^{n} \mathbb{P}(\hat{y}_t \in B_i) = \mathbb{P}(\Omega) = 1 \tag{4.20}$$

Algebraically, the neural network can be represented as the composition of two transformations, say, $\text{LSTM} : \mathbb{R}^{n \times m} \to \mathbb{R}^{32}$ and, respectively for each experiment, $D^d : \mathbb{R}^{32} \to \mathbb{R}^3_1$ and $D^v : \mathbb{R}^{32} \to \mathbb{R}^2_1$:

1 *Hard sigmoid* is a piecewise linear function, defined as

$$H(x) := \begin{cases} 0 & x \leqslant -2.5 \\ \frac{x+2.5}{5} & x \in (-2.5, 2.5) \\ 1 & x \geqslant 2.5 \end{cases} \tag{4.17}$$

$$N^d : \mathbb{R}^{24\times 5} \to \mathbb{R}^3_1 \qquad N(x) := \mathrm{LSTM} \circ D^d(x) \tag{4.21}$$

$$N^v : \mathbb{R}^{24\times 5} \to \mathbb{R}^2_1 \qquad N(x) := \mathrm{LSTM} \circ D^v(x) \tag{4.22}$$

Thus, once trained, the network will transform a *sequence* into the one-period-ahead forecast.

The 32-dimension figure was found by empirical means as it yields desirable results in terms of both minimum loss achieved and speed of convergence. Along with it, the dimension $m = 5$ was identified as the number of distinct features in a sequence (*open, high, low, close, volume*) and the dimension $n = 24$ (corresponding to 2 hours windows) was empirically found to yield acceptable results.

4.3 LOSS FUNCTION

As the experiments involve the construction of a *classifier*, it is necessary to properly choose the *loss function*. A metric is needed that is able to capture some sort of "distance" between the actual probability distribution of the inputs, i.e. y^*_t, and the predicted probability distribution, i.e. $\hat{y}^*_t$. We will employ instruments from the field of *information theory* [8].

As we are dealing with probability distributions, we may give the definition of *entropy*:

$$H(\bar{y}) = -\sum_{y\in Y} \mathbb{P}(\bar{y} = y) \log_2 \mathbb{P}(\bar{y} = y) \tag{4.23}$$

(for ease of notation, we will write $p_y \equiv \mathbb{P}(\bar{y} = y)$). Intuitively, *entropy* measures the uncertainty of a discrete random variable in *bits*. Let us develop the concept as a simple example. Consider the toss of a fair coin, and the associated probability density function

$$X_c = \begin{cases} \textit{heads} & p = \frac{1}{2} \\ \textit{tails} & p = \frac{1}{2} \end{cases} \tag{4.24}$$

Its entropy is computable as

$$H(p) = -p_h \log_2 p_h - p_t \log_2 p_t = -\frac{1}{2}\log_2 \frac{1}{2} - \frac{1}{2}\log_2 \frac{1}{2} \tag{4.25}$$

$$= \frac{1}{2} + \frac{1}{2} = 1 \tag{4.26}$$

that is, one *bit* of entropy. This is intuitive as one could map the binary value 0 in a bit to the *tails* event and the binary value 1 to the *heads* event. Consider the analogous case of a fair die:

$$X_d = \begin{cases} 1 & p^d = \frac{1}{6} \\ 2 & p^d = \frac{1}{6} \\ \vdots & \\ 6 & p^d = \frac{1}{6} \end{cases} \tag{4.27}$$

Its entropy is computable as

$$H(p^d) = -\sum_{i=1}^{6} p_i^d \log_2 p_i^d = -\sum_{i=1}^{6} \frac{1}{6} \log_2 \frac{1}{6} \tag{4.28}$$

$$= -\log_2 \frac{1}{6} \approx 2.58 \tag{4.29}$$

The intuitive difference with the fair coin example is that representing the information conveyed from the possible states of the die variable *using the* X_d *representation* is "more expensive" in terms of bits due to the higher complexity of the random variable – trivially, one can represent 2 states with 1 bit (2^1), and 8 states with 3 bits (2^3), hence 6 states of equal probability can be represented with slightly less than 3 bits, that is, with $H(X_d)$ bits.

Consider now an unfair coin, that is, the variable X_c endowed with a different probability distribution q:

$$X_c = \begin{cases} \textit{heads} & q = \frac{1}{4} \\ \textit{tails} & q = \frac{3}{4} \end{cases}, \quad H(q) = -\frac{1}{4} \log_2 \frac{1}{4} - \frac{3}{4} \log_2 \frac{3}{4} \approx 0.623 \tag{4.30}$$

The entropy is now lower than 1, owing to the fact that most of the information is carried by the *tails* event, which is more frequent.

Consider now the case of a suspicious agent betting on X_c who decides, lacking information on the coin's true probability distribution p, to assume it is distributed unfairly instead, with probability q: the agent will not be able to take decisions in the most efficient way as it is assuming a wrong probability distribution. The "cost" of assuming this wrong distribution is measurable via the *Kullback-Leibler divergence*:

$$D(p\|q) = \sum_{x \in X_c} p(x) \log_2 \frac{p(x)}{q(x)} \tag{4.31}$$

This measure satisfies *Gibbs' inequality*[20]:

$$D(p\|q) \geqslant 0, \quad D(p\|q) = 0 \iff p \equiv q \tag{4.32}$$

and fits the criteria required for a *loss function* as defined in 3.5, 3.6, 3.7. Do note that the divergence is not an actual measure, as it is not symmetrical: $D(p\|q) \neq D(q\|p)$ in general. This can be explained informally by noting that the *Kullback-Leibler divergence* measures the *cost of assuming distribution* q *over the actual distribution* p, and the opposite, i.e. *assuming distribution* p *over the distribution* q, may not be as costly (or as cheap). In information terms, the divergence measures the *number of extra bits* required to encode information pertaining to a variable distributed like p if the distribution q is used in its stead.

It is appropriate to also give the definition of *cross entropy*, both in absolute terms and in relation to *entropy* and *Kullback-Leibler divergence*:

$$H(p, q) = -\sum_{x \in X_c} p(x) \log_2 q(x) = H(p) + D(p\|q) \tag{4.33}$$

As the entropy depends strictly on the probability distribution, for fixed p the two measures are equivalent up to the additive constant $H(p)$.

In the context of our forecasting framework, we can use either the *cross entropy* or the *Kullback-Leibler divergence* to measure the cost of using the distribution emerging from the prediction, $\hat{y}$, when the actual distribution is y. As no assumptions are made over the underlying distribution of the data, a numerical estimation is carried in the form of averaging the measures computed over the sample and the forecast:

$$\hat{D}(y\|\hat{y}) = \frac{1}{N} \sum_{t=1}^{N} D(y_t\|\hat{y}_t) \tag{4.34}$$

$$\hat{H}(y, \hat{y}) = \frac{1}{N} \sum_{t=1}^{N} H(y_t, \hat{y}_t) \tag{4.35}$$

Either of the two functions above can be used interchangeably as loss function in the deep network, as both fit the relaxed requirement of a unique global minimum on two identical arguments.

4.4 SUMMARY OF THE MODEL

The model implemented is structurally identical for both the directionality experiment and the volatility experiment, with the exception of the

number of neurons in the output layer – three for the directionality case, two for the volatility case, one neuron for each class. Summing up, the structure of the module is as follows:

- LSTM input layer: $f_{LSTM} : \mathbb{R}^{24\times5} \to \mathbb{R}^{32}$
  - 24 recurrent units
  - *hyperbolic tangent* input activation
  - *hard sigmoid* gates activation
  - L^2 regularization
- Dropout layer with $p = 0.2$: $f_{dropout} : \mathbb{R}^{32} \to \mathbb{R}^{32}$
- Dense output layer: $f_{dense} : \mathbb{R}^{32} \to \mathbb{R}^{q}_{1}$
  - $q = 3$ (*directionality*) or $q = 2$ (*volatility*) neurons
  - *softmax* activation
  - L^2 regularization

The number of trainable parameters in the LSTM layer arises from the cells' internal structure and is computable as

$$4((m+1)q + q^2) \tag{4.36}$$

where $m = 5$ is the number of features and $q = 32$ is the dimensionality of the output space, which yields 4864 trainable parameters; the number $n = 24$ does not enter the computation as each timestep is fed sequentially and bears no effect on the parameters, and the number 1 summed to m indicates the bias term. The dropout layer has no trainable parameters, as it is simply a random static operation on the edges independent of the parameter space. The number of parameters for the dense layer is simply

$$(q+1) * o \tag{4.37}$$

where $q = 32$ is the dimensionality of the input space, $o = 3$ or $o = 2$ is the dimensionality of the output space and, again, the number 1 indicates the bias. This leads to 99 parameters for the *dimensionality* case and 66 parameters for the *volatility* case. Overall, we have:

- $f_d : \mathbb{R}^{24\times5} \to \mathbb{R}^{3}_{1}$ with 4963 parameters (*dimensionality network*)
- $f_v : \mathbb{R}^{24\times5} \to \mathbb{R}^{2}_{1}$ with 4930 parameters (*volatility network*)

These models, while exhibiting a staggering amount of parameters compared to classical econometrics models – consider that models as parsimonious as GARCH(2,2) or ARMA(2,2) are few and far between and already considered too complex – are parsimonious enough given both the nature of deep networks, and the fact that the number of samples used for the learning process is much higher, spanning from a few tens of thousands up to hundreds of thousands of sequences.

Other hyperparameters have been chosen as follows:

- *Cross entropy* has been chosen as loss function
- The optimization algorithm is *Adam* with a learning rate of 10^{-4}
- We chose to allocate the observations from January 1st, 2018 to January 27th, 2018 (roughly 7500 observations) as *test set*
- The rest of the samples have been split as 80% training set, 20% validation set
- The training happens in *mini-batches* of 64 sequences at a time; this allows the computation to be parallelized and executed on a GPU to some degree and is a good trade-off between speed and results, as the LSTM's internal state is only reset from one batch to the next
- The training happens over 100 epochs; although some sort of convergence is achieved early on in the training, we observed more robust results by avoiding early stopping

Thusly configured, training the model takes roughly between one and three hours of time, where higher sample count translates to longer training times.

5

RESULTS

5.1 TIME SERIES SOURCING AND STRUCTURE

We hereby present the outcome of our analysis. The time series data was sourced from Coinbase[1]'s GDAX exchange[3]'s historical data *application programming interface*. Data was requested in the form of 5 minutes OHLCV candles in a timespan ranging from January 1st, 2016 to January 27th, 2018, for the following currency pairs:

- BTC-EUR (*Bitcoin* to Euro)
- BTC-USD (*Bitcoin* to US Dollar)
- ETH-EUR (*Ethereum* to Euro)
- ETH-USD (*Ethereum* to US Dollar)
- LTC-EUR (*Litecoin* to Euro)
- LTC-USD (*Litecoin* to US Dollar)

The time series were downloaded via a Python script and subsequently processed in order to obtain suitable sequences in the form specified in 4.1. To obtain as many samples as possible, sequences were computed in a sliding window fashion, i.e. all matrices M_t were evaluated for $t = n, n+1, \dots N$ which would comprise data points respectively indexed $(1 \dots n-1)$, $(2 \dots n)$, $\dots (N-n \dots N-1)$. The overlapping of sequences arising from this formatting would introduce a bias that could cause overfitting; to correct for it, it was sufficient to shuffle the sample ordering, i.e. presenting sequences to the network in a random order, so that overlapping sequences would not appear close to one another and the gradient could move in more significant directions. Furthermore, sequences including missing data were discarded.

Table 1: Time series sample sequences count

Symbol	Training set	Validation set	Test set	Total
BTC-EUR	153854	38463	7474	199791
BTC-USD	167234	41808	7474	216516
ETH-EUR	37768	9441	7441	54650
ETH-USD	125332	31332	7474	164138
LTC-EUR	30873	7718	7319	45910
LTC-USD	76739	19184	7474	103397

The number of sequences for the training, validation and testing of each pair is reported in Table 1. Noteworthy is the fact that the EUR-based pairs have a much smaller sample count, arising from the fact that some pairs have been available for trading only later on in the time period considered, and EUR markets are generally slightly less liquid than the USD counterparts and tend to have more missing data.

As we have around 5000 parameters in each of the deep models, we roughly obtain 6 to 33 observations per parameter. While there are neither hard-and-fast rules, nor loose heuristics related to such ratio, it is tenable that the model could benefit from a much larger amount of data – as is the case with the largest part of deep learning models, that have the general tendency to be much less parsimonious than any other class of models.

In the figures 11 and 12 is reported the evaluation of the *cross entropy* and *Kullback-Leibler divergence* for all the pairs considered, respectively for the *directionality* and for the *volatility* experiment.

It is interesting to make some observations upon the convergence speed. In the *directionality* experiment, the pairs with the largest amount of training samples (BTC-EUR, BTC-USD and ETH-EUR) reached convergence as early as the tenth epoch, which is much sooner than the other pairs; the most blatant case being LTC-EUR, which kept making significant progress well into the fiftieth epoch. This heterogeneity is similarly reflected in the volatility experiment, although convergence happens earlier for all pairs (e.g. LTC-EUR has good enough convergence before the fortieth epoch). This is explainable by noting the simpler structure of the experiment, and the better balance of the classes (see tables 2 and 3): while the cardinality of the *neutral* class in the *directionality* experiment is equal to the cardinality of the *low volatility* class in the *volatility* experi-

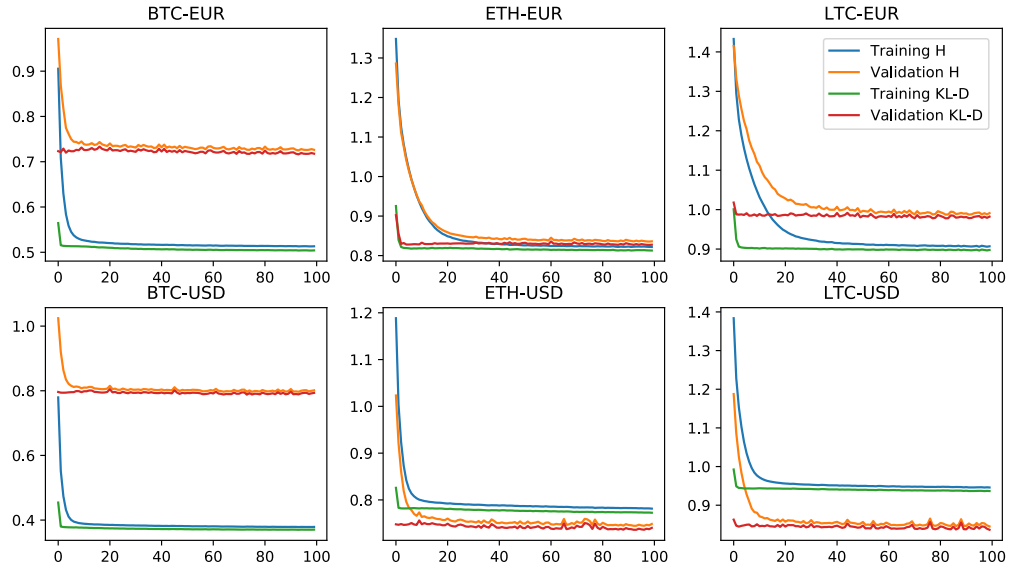

Figure 11: *Directionality* experiment. Cross entropy and Kullback-Leibler divergence evaluated at the end of each training epoch for each pair, both for the training set and the validation set

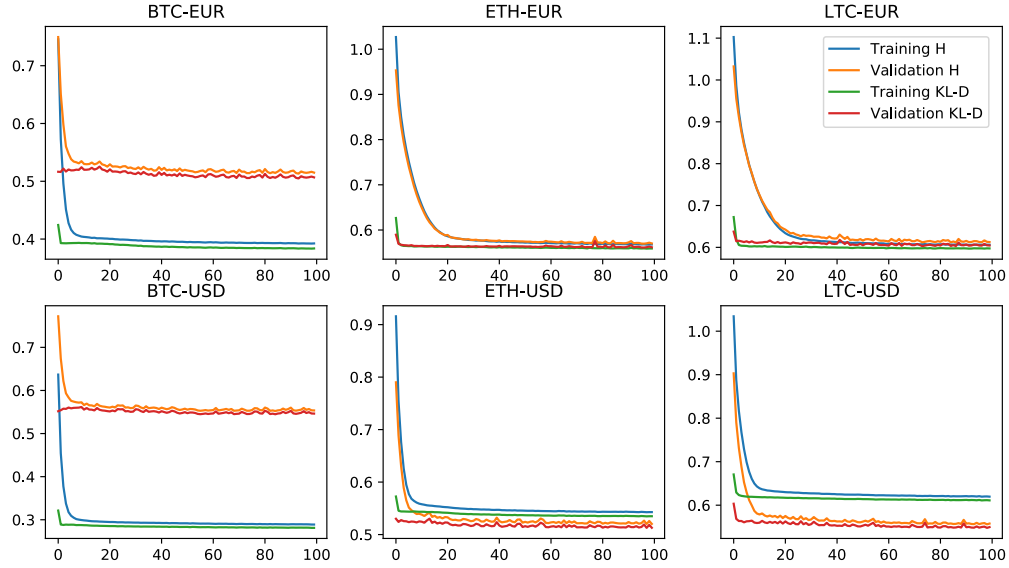

Figure 12: *Volatility* experiment. Cross entropy and Kullback-Leibler divergence evaluated at the end of each training epoch for each pair, both for the training set and the validation set

Table 2: *Directionality* data set labels distribution

	Training set			Test set		
Symbol	*Decrease*	*Neutral*	*Increase*	*Decrease*	*Neutral*	*Increase*
BTC-EUR	9.73%	80.05%	10.22%	24.73%	50.94%	24.34%
BTC-USD	8.54%	82.75%	8.71%	27.12%	45.89%	26.99%
ETH-EUR	18.19%	63.04%	18.77%	22.85%	52.16%	25.00%
ETH-USD	16.66%	66.30%	17.05%	26.13%	45.21%	28.66%
LTC-EUR	22.60%	54.65%	22.76%	24.07%	53.03%	22.90%
LTC-USD	22.71%	54.34%	22.95%	27.01%	46.21%	26.77%

ment, ranging from 54% up to 80%, the *high volatility* class has roughly double the size of each of the *price decrease* and *price increase* classes; in fact, the former is just a merged version of the latter two, and while in the *volatility* case a very balanced distribution of 54%–46% can be attained (in the LTC-USD case), the best case scenario in the *directionality* case attains a 23%– 54%–23% split, which is somewhat unbalanced already, and much worse imbalances in the other cases.

While generally balanced classes are desirable, one must not forget the practical nature of a forecasting framework: the target of balanced classes could be simply achieved by choosing a different threshold, but if it was set too low the classification might not have made sense in real world applications. The chosen threshold – 0.2% – on the other hand, might be of use e.g. in choosing to perform high frequency trades only when there's a high enough expectation of a short term variation of size sufficient to cover transaction costs.

Another apparent peculiarity is the remarked difference in balance between the training sets and the test sets. Test sets, spanning roughly four weeks of data, appear much more balanced than their training set counterparts, spanning up to two years, for the same pair, and there is much less variation between pairs as well; all pairs have roughly a 25%–50%–25% balance in the directionality case, and a 50%–50% balance in the volatility case. This fact probably embodies some volatility trend affecting the last month of data which wasn't always present in the preceding period.

Table 3: *Volatility* data set labels distribution

	Training set		Test set	
Symbol	*Low volatility*	*High volatility*	*Low volatility*	*High volatility*
BTC-EUR	80.05%	19.95%	50.94%	49.06%
BTC-USD	82.75%	17.25%	45.21%	54.11%
ETH-EUR	63.04%	36.96%	52.16%	47.84%
ETH-USD	66.30%	33.70%	45.21%	54.79%
LTC-EUR	54.65%	45.35%	53.03%	46.97%
LTC-USD	54.34%	45.66%	46.21%	53.79%

5.2 STATISTICS

To ascertain the effectiveness of our model, we evaluated some metrics derived from the *confusion matrix* on the results yielded from running the model on the *test set*. We compared the predicted classes to the actual classes in order to identify:

- *True positives*: the instances where the classifier predicted a "positive" outcome when the actual outcome was "positive" (*Statistical power*)
- *True negatives*: the instances where the classifier predicted a "negative" outcome when the actual outcome was "negative"
- *False positives*: the instances where the classifier predicted a "positive" outcome when the actual outcome was "negative" (*Type I error*)
- *False negatives*: the instances where the classifier predicted a "negative" outcome when the actual outcome was "positive" (*Type II error*)

In the *volatility* experiment, we arbitrarily chose to equate the *high volatility* class with the *positive* outcome as that is arguably the more interesting event to forecast. In the *directionality* experiment, we computed statistics in three groupings:

- Considering "positive" the *price increase* class and "negative" the aggregate of the *neutral* and *price decrease* classes
- Considering "positive" the *neutral* class and "negative" the aggregate of the *price increase* and *price decrease* classes

- Considering "positive" the *price decrease* class and "negative" the aggregate of the *neutral* and *price increase* classes

This setup allows us to evaluate the predictive power of the model individually for each class.

Via the numerosity of the elements in the *confusion matrix* for each of the cases above, we computed the following statistics:

- *Precision*: the probability that the actual outcome is "positive" conditional on the classifier predicting "positive". The higher the precision, the more confident is the classifier when emitting a "positive" classification.
- *Recall*: the probability that the classifier predicts "positive" conditional on the actual outcome being "positive". The higher the recall, the fewer "positive" outcomes are mislabeled by the classifier.
- *Specificity*: the probability that the actual outcome is "negative" conditional on the classifier predicting "negative". The higher the specificity, the lower the chance that a "negative" prediction is actually a "positive" outcome.
- *Accuracy*: the probability that a given prediction is correct, both accounting for true positive and true negative.
- F_1 *Score*: it is computed as the harmonic average of precision and recall; it has no direct probabilistic meaning, but a classifier may be considered of better quality the closer the F_1 is to 1, and of worse quality the closer the F_1 is to 0.

The statistics are computed as follows, where tp, tn, fp, fn respectively indicate the number of true positives, true negatives, false positives and false negatives:

$$\text{Precision} = \frac{tp}{tp + fp} \tag{5.1}$$

$$\text{Recall} = \frac{tp}{tp + fn} \tag{5.2}$$

$$\text{Specificity} = \frac{tn}{tn + fp} \tag{5.3}$$

$$\text{Accuracy} = \frac{tp + tn}{tp + tn + fp + fn} \tag{5.4}$$

Table 4: Statistics for the *directionality* experiment

Symbol	Class	Precision	Recall	Specificity	Accuracy	F1 Score
BTC-EUR	↑	0.42935	0.08199	0.96393	0.74461	0.13769
	↔	0.51402	0.98265	0.07742	0.52846	0.67497
	↓	–	–	–	–	–
BTC-USD	↑	0.41798	0.21289	0.88450	0.69620	0.28210
	↔	0.48943	0.92642	0.23448	0.54031	0.64048
	↓	0.37576	0.02783	0.98223	0.71726	0.05182
ETH-EUR	↑	0.41030	0.20577	0.88962	0.70376	0.27408
	↔	0.57872	0.90125	0.39767	0.63870	0.70484
	↓	0.41345	0.19597	0.90755	0.72996	0.26590
ETH-USD	↑	0.41093	0.26554	0.82892	0.65422	0.32261
	↔	0.51758	0.80080	0.48003	0.61174	0.62877
	↓	0.36842	0.21678	0.85597	0.67743	0.27295
LTC-EUR	↑	0.42027	0.17094	0.92320	0.73839	0.24303
	↔	0.57665	0.93036	0.32135	0.62488	0.71200
	↓	0.40927	0.15347	0.92380	0.72666	0.22324
LTC-USD	↑	0.40151	0.23159	0.86035	0.67925	0.29375
	↔	0.51779	0.86122	0.40974	0.60114	0.64674
	↓	0.39858	0.17815	0.89126	0.68586	0.24624

$$F_1 = \frac{2}{\frac{1}{\text{Precision}} + \frac{1}{\text{Recall}}} \tag{5.5}$$

Tables 4 and 5 report the computed metrics for all of the cases considered, which may be used to infer the quality of the model.

Results of the directionality experiment

It is immediately apparent that our model in the *directionality* experiment performed very poorly. Both *increase* and *decrease* labels for all of the considered pairs achieved precisions in the range 35%–43%, which means that the model is highly unreliable whenever predicting a direction.

Table 5: Statistics for the *volatility* experiment

Symbol	Precision	Recall	Specificity	Accuracy	F1 Score
BTC-EUR	0.675115	0.479411	0.777778	0.631389	0.560676
BTC-USD	0.711503	0.478734	0.771137	0.612925	0.572358
ETH-EUR	0.690157	0.519944	0.785880	0.658648	0.593079
ETH-USD	0.688474	0.647619	0.644865	0.646374	0.667422
LTC-EUR	0.704282	0.459279	0.829168	0.655417	0.555986
LTC-USD	0.674961	0.647761	0.636943	0.642762	0.661081

Particularly noteworthy is the case of the BTC-EUR pair, for which the classifier emitted no "decrease" predictions at all. The *neutral* case is slightly better, with values ranging between 48%–58% which is barely better than emitting the signal via the flip of a coin. Recall is just as bad for the *increase* and *decrease* classes: anywhere between 75% and 90% of the "positive" events are missed by the classifier. The *neutral* case, on the other hand, exhibits a disproportionately high recall. An interpretation can be derived from these results: all the model learned to do was to emit the *neutral* label most of the time. If the model always predicts the same label, it will certainly capture every instance of the corresponding event happening! This can further be confirmed by noticing the high specificity of the *increase* and *decrease* classes: the model is right most of the time when predicting something other than *increase* and *decrease* respectively, and this emerges from the large number of *neutral* predictions. The overall high accuracy is misleading, again as a consequence of classifying most observations as *neutral*. The F_1 score reveals the inadequacy, by presenting generally unsatisfactory values. We conclude that LSTM based deep neural network are not a suitable tool for directionality prediction in the domain of cryptocurrencies and under the simple formulation proposed, and at the very least more articulate approaches should be considered[4].

Results of the volatility experiment

The model responds much better under the *volatility* formulation of the problem. Precision is between 67% and 72% for all the six analyzed time series, which means that in general a "positive" signal is much more reliable than random guessing; how important an error the 30%

of wrong "positive" classifications constitutes will generally depend on the strategy it is being deployed in. The recall is lower than the precision for all of the pairs; this means that the model emits fewer "positive" signals than the number of actual volatility events, but is reasonably confident on the prediction when it does emit said signals. Specificity exhibits more variability across the pairs, while being generally satisfactory – with values above 77% on all pairs but ETH-USD and LTC-USD, in both of which cases it lies around 64%. Accuracy is consistent between 61% and 65% which indicates a generally good performance when accompanied by the previously commented measures. Lastly, F_1 highlights the penalization occurring on the recalls, symptom of the fact that the model is affected by the same issue as the *directionality* model: namely, a bias towards the *neutral* class.

5.3 CONCLUDING REMARKS AND FUTURE DEVELOPMENTS

We designed a simple *long short-term memory* deep learning model for classifying cryptocurrency time series. The results in the *directionality* experiment were unsatisfactory, while the *volatility* experiment showed promising results. As absence of autocorrelation for returns and presence of autocorrelation for squared returns are commonly known stylized facts for most other asset classes, these results suggest that, despite their overly erratic behaviour, cryptocurrencies may not necessarily exhibit peculiar short term patterns with time dependencies which may be captured by means of nonlinear classification.

Further study should be carried upon the state of the art in terms of model optimization in machine learning: as the choice of model structure and hyperparameters was conducted heuristically, it is plausible that algorithmic optimization over the (vast) hyperparameter space may lead to significant improvements.

BIBLIOGRAPHY

[1] Coinbase. https://www.coinbase.com/.

[2] Coinmarketcap.com. https://coinmarketcap.com/.

[3] The gdax exchange. https://www.gdax.com/.

[4] Wei Bao, Jun Yue, and Yulei Rao. A deep learning framework for financial time series using stacked autoencoders and long-short term memory. *PLOS ONE*, 12(7):1–24, 07 2017.

[5] Stephen Chan, Jeffrey Chu, Saralees Nadarajah, and Joerg Osterrieder. A statistical analysis of cryptocurrencies. *Journal of Risk and Financial Management*, 10(2), 2017.

[6] David Chaum. Blind signatures for untraceable payments. In David Chaum, Ronald L. Rivest, and Alan T. Sherman, editors, *Advances in Cryptology*, pages 199–203, Boston, MA, 1983. Springer US.

[7] David Chaum, Amos Fiat, and Moni Naor. Untraceable electronic cash. In Shafi Goldwasser, editor, *Advances in Cryptology — CRYPTO' 88*, pages 319–327, New York, NY, 1990. Springer New York.

[8] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley-Interscience, New York, NY, USA, 1991.

[9] John (JD) Douceur. The sybil attack. In *Proceedings of 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, January 2002.

[10] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12:2121–2159, July 2011.

[11] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In Ernest F. Brickell, editor, *Advances in Cryptology — CRYPTO' 92*, pages 139–147, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[12] Salah El Hihi and Yoshua Bengio. Hierarchical recurrent neural networks for long-term dependencies. In *Proceedings of the 8th International Conference on Neural Information Processing Systems*, NIPS'95, pages 493–499, Cambridge, MA, USA, 1995. MIT Press.

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

[14] John Hopfield. Neural networks and physical systems with emergent collective computational abilities. 79, 04 1982.

[15] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[16] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4/3:382–401, July 1982.

[17] Jan Lansky. Possible state approaches to cryptocurrencies. 8, 01 2018.

[18] Tsungnan Lin, B. G. Horne, P. Tino, and C. L. Giles. Learning long-term dependencies in narx recurrent neural networks. *Trans. Neur. Netw.*, 7(6):1329–1338, November 1996.

[19] William J. Luther and Alexander W. Salter. Bitcoin and the bailout. *The Quarterly Review of Economics and Finance*, 66:50 – 56, 2017.

[20] David J. C. MacKay. *Information Theory, Inference & Learning Algorithms*. Cambridge University Press, New York, NY, USA, 2002.

[21] Martina Matta, Ilaria Lunesu, and Michele Marchesi. Bitcoin spread prediction using social and web search media. 06 2015.

[22] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, Dec 1943.

[23] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, http://bitcoin.org/bitcoin.pdf.

[24] Sebastian Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016.

[25] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–359, 10 2017.

[26] Nassim Nicholas Taleb. *Antifragile: Things that Gain from Disorder*. Allen Lane, London, UK, 2012.