UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Centre de Formació Interdisciplinària Superior

THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Bachelor's Thesis

A deep learning approach to portfolio optimization

Pau Cartanyà Caro

Advisor (HKUST):
Daniel P. Palomar

Tutor (UPC):
Argimiro Arratia

February 2022

In partial fulfilment of the requirements for the

Bachelor's degree in Mathematics

Bachelor's degree in Informatics Engineering

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat de Matemàtiques i Estadística

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

ABSTRACT

Since Markowitz's work on portfolio theory in 1952, numerous developments have been carried out to improve the original techniques of portfolio optimization, to the point that the current research on the topic focuses on building deep learning models that design the whole portfolio in only one step. Even though deep learning has helped to solve a wide range of problems in different areas, it has not proven successful enough yet in some branches of finance such as portfolio design.

After verifying that heuristic or more simple methods do not work as active trading strategies, the goal of this thesis becomes to explore how the portfolio optimization problem can be approached via deep learning. We analyze whether the classical portfolio theory ideas should be either completely replaced by end-to-end neural networks or incorporated into the architectures, and we discuss the impact that the frequency of the data has on the performance of the models.

For this reason, multiple deep learning architectures that take as input only asset returns have been implemented and extensively backtested, mainly on high-frequency cryptocurrency data. All backtests have been carried out under the same trading framework and their performances have been compared to some standard benchmarks in order to extract significant conclusions. Our results indicate that, on high-frequency data, deep learning models are capable of detecting features that are invisible to the classical financial methods. Nevertheless, this predictability can not be exploited in real-life trading as the portfolios need to be largely reoptimized often and therefore the transaction costs imposed by the exchanges cancel out all the profits.

Keywords: Quantitative finance, portfolio optimization, deep learning, time series analysis

AMS Code: 91G05

RESUM

Des del desenvolupament de la teoria moderna de carteres de Markowitz l'any 1952, s'han dut a terme numerosos avenços per a millorar-ne les tècniques originals, fins al punt que la recerca actual en aquest àmbit es centra en construir models d'aprenentatge profund que distribueixin el capital de manera òptima entre els actius considerats en un sol pas. Malgrat que l'aprenentatge profund ha ajudat a resoldre un gran nombre de problemes en altres àrees, encara no s'ha demostrat que sigui prou exitós en algunes branques del camp de les finances com és el cas del disseny de carteres.

Després de comprovar que mètodes més heurístics no funcionen com a estratègies d'inversió activa, l'objectiu d'aquesta tesi esdevé explorar com l'aprenentatge profund pot servir per abordar el problema de l'optimització de carteres. S'analitza si les idees clàssiques de la teoria de carteres han d'ésser reemplaçades completament per xarxes neuronals que integrin tot el procés, o si bé és més adequat incorporar-les dins de l'arquitectura dels models. Es discuteix també l'impacte que té la freqüència de les dades amb què treballen els models en el seu rendiment.

Per aquests motius, s'han implementat i posat a prova diversos models d'aprenentatge profund que treballen únicament amb els guanys històrics dels actius, entrenats principalment en dades d'alta freqüència de criptomonedes. Totes les proves s'han dut a terme sota el mateix marc d'inversió i els seus rendiments s'han comparat amb el d'altres carteres estàndard per tal de poder extreure conclusions significatives. Els results obtinguts indiquen els models d'aprenentatge profund implementats són capaços de detectar patrons en les dades que són invisibles pels models financers clàssics. No obstant, aquesta predictibilitat no es pot aprofitar completament en les inversions a temps real ja que les carteres han de ser reoptimizades sovint i això provoca que els costs que les plataformes imposen a cada transacció neutralitzin tots els guanys.

Paraules clau: Matemàtica financera, optimització de carteres, aprenentatge profund, anàlisi de sèries temporals

Codi AMS: 91G05

RESUMEN

Desde el desarrollo de la teoría moderna de carteras de Markowitz en el año 1952, se han llevado a cabo numerosas mejoras sobre las técnicas originales, hasta el punto que la investigación actual en este ámbito se centra en construir modelos de aprendizaje profundo que distribuyan el capital de manera óptima entre los activos considerados en un solo paso. Aunque el aprendizaje profundo ha ayudado a resolver un gran número de problemas en otras áreas, aún no se ha demostrado que sea suficientemente exitoso en algunas ramas del campo de las finanzas como es el caso del diseño de carteras.

Después de verificar que métodos más heurísticos no funcionan como estrategias de inversión activa, el objetivo de esta tesi es explorar cómo el aprendizaje profundo puede usarse para abordar el problema de la optimización de carteras. Se analiza si las ideas clásicas de la teoría de carteras deben ser reemplazadas completamente por redes neuronales que integren todo el proceso, o si conviene incorporarlas dentro de la arquitectura de los modelos. Se discute también el impacto que tiene la frecuencia de los datos con que trabajan los modelos en su rendimiento.

Por estos motivos, se han implementado y puesto a prueba varios modelos de aprendizaje profundo que trabajan únicamente con las ganancias históricas de los activos, entrenados principalmente con datos de alta frecuencia de criptomonedas. Todas las pruebas se han llevado a cabo bajo el mismo marco de inversión y sus rendimientos se han comparado con el de otras carteras estándar para poder extraer conclusiones significativas. Los resultados obtenidos indican que los modelos de aprendizaje profundo implementados son capaces de detectar patrones en los datos que son invisibles para los modelos financieros clásicos. Sin embargo, esta predictibilidad no se puede aprovechar completamente en las inversiones a tiempo real ya que las carteras deben ser reoptimizadas a menudo y esto provoca que los costes que las plataformas imponen a cada transacción neutralicen todos los beneficios.

Palabras clave: Matemática financiera, optimización de carteras, aprendizaje profundo, análisis de series temporales

Código AMS: 91G05

ACKNOWLEDGEMENTS

First and foremost, I would like to sincerely thank Prof. Daniel P. Palomar. Not only for his time and guidance during these months in Hong Kong, but also for many meaningful conversations besides the usual work topics. I would also like to extend my gratitude to the rest of his research group for letting me know first-hand what working on research is like.

I am grateful to CFIS for giving me this opportunity to culminate my undergraduate studies with an international experience of this caliber. Moreover, this adventure would not have been possible with the help of HKUST, Fundació Ciutat de Valls and Fundació Privada Cellex. A heartfelt acknowledgement to the latter for the trust they placed in me and my education for the past 7 years, which is above and beyond the financial support.

Lastly, huge thanks to my family for their support and understanding throughout all these years.

Pau Cartanyà Caro
Hong Kong, 4 February 2022

Contents

1 Introduction **1**

2 Primer on financial data **3**

2.1 Asset prices and returns . . . 3

2.2 Portfolio basics . . . 5

2.3 Performance measures . . . 7

3 Literature review **9**

4 Preliminary experiments **11**

4.1 Univariate sizing . . . 11

4.2 Experiments on daily data . . . 12

4.2.1 Dataset . . . 12

4.2.2 Moving average-based strategies . . . 12

4.2.3 Volatility targeting . . . 14

4.2.4 Machine learning models . . . 16

4.2.5 Direct reinforcement learning . . . 19

4.3 Experiments on minute-by-minute data . . . 22

4.3.1 Dataset . . . 22

4.3.2 Direct reinforcement learning . . . 23

4.4 Overall assessment . . . 25

5 Main experiments **26**

5.1 Dataset . . . 26

5.2 Trading framework . . . 27

5.3 Proposed architectures . . . 31

5.3.1 Two-block architecture . . . 31

5.3.2 End-to-end . . . 32

5.4 Neural networks . . . 33

5.4.1 Proposed feed-forward networks 33
5.4.2 LSTM 36
5.4.2.1 Definition 36
5.4.2.2 Proposed networks 38
5.5 Results 41
5.5.1 Two-block architecture 41
5.5.2 End-to-end models 44
5.5.3 Summary of long-term performances 45
5.5.4 Analysis on the capacity of overfitting of the models 48

6 Experiments on data of different frequencies 51
6.1 Dataset 51
6.2 Trading framework 52
6.3 Results 53

7 Incorporating transaction costs 55
7.1 Dataset 55
7.2 Trading framework 56
7.3 Results 57

8 Conclusions 60

Bibliography 62

Chapter 1

Introduction

Over the last decade, deep learning has been the cause of a whole revolution in multiple disciplines. The way some problems are approached in fields like image recognition, natural language processing, recommendation systems or bioinformatics has completely changed and neural networks provoked major improvements from the cutting-edge techniques used before. Out of all the fields impacted by this revolution finance is not an exception.

In nowadays finance world, most records and observations are captured electronically by internet-connected devices. This makes finance become a data based discipline and allows investors to access a wide range of market data that was not available for them decades ago. As the amount of available data increases, the tools used to deal with it become more and more complex, whereas the range of problems that can be tackled becomes broader. Nowadays, machine learning (and, more specifically, deep learning) is used in problems such as fraud detection, credit assessment and risk management. If we want to look at more concrete examples, deep learning could help to handle online prices of millions of items and assess inflation, or it could be used to process satellite imaging to assess the activity of oil rigs or the commercial activities in harbours by recognizing the amount of containers stored [12].

Nevertheless, there is one problem in finance where deep learning has not provided significant improvements yet: portfolio optimization. Portfolio optimization is the decision making process of continuously selecting the best allocation of a budget into different financial investment products with the objective of maximizing certain function. For instance, one could try to maximize the returns while minimizing the risk. This idea was first brought by Markowitz in 1952 [16] and since then it has established the foundation of how the portfolio optimization problem is approached. Numerous developments have been done to the original Markowitz's modern portfolio theory, which was proven to suffer from a great number of limitations, up to the point that the current research

on the topic focuses on the use of deep learning tools to overcome them. However, not much success has been achieved yet. The special characteristics of financial data and the intricacies of real life trading make it really tricky to set up a good model. Moreover, there is one main difference between finance and other applications where deep learning provides excellent results: in portfolio optimization we are not trying to replicate a task that humans can already do well, such as recognizing images or responding appropriately to verbal requests. This might change completely the problems deep learning may face.

It is this wide range of options that can still be explored when facing portfolio optimization via deep learning what motivates this thesis. The main objective is to analyze the ability of neural networks to detect and exploit predictability on financial data (mainly historical asset returns) to design profitable portfolios. This decouples in 3 subtopics that are explored separately in this project. Firstly we study how different architectures can lead to different performances, focusing on whether the neural networks should still include some ideas of classical portfolio theory or not. Secondly we explore the impact that the frequency of the data has on the predictability the models are capable of learning. And finally we discern the influence of transaction costs on the profitability of the models to conclude if they could be brought into real-life trading scenarios or not. All this is done by running extensive backtests on different scenarios, mainly on intraday cryptocurrency data, evaluating them with some of the most well-known performance measures and comparing them to some standard benchmarks of the industry.

All the work carried out to achieve these goals is presented on the rest of this document, which is structured as follows.

- Chapter 2 introduces some useful concepts about quantitative finance that are used throughout all the project and includes a brief study on the nature of financial data.
- Chapter 3 reviews previous work related to the challenges faced in this project, focusing on the latest advances on deep learning applied to portfolio optimization.
- Chapter 4 proves the need of a complex tool like deep learning to approach portfolio optimization by verifying that simpler heuristic strategies do not work in this problem.
- Chapter 5 contains the main experiments carried out for this project. Different architectures are described and backtested under a specific trading framework.
- Chapter 6 contains experiments on the frequency of the data.
- Chapter 7 analyzes the changes in performance when transaction costs are taken into account.
- Chapter 8 discusses how the results obtained from the experiments can be put together to understand the difficulties of portfolio optimization and finally concludes the thesis.

Chapter 2

Primer on financial data

Even though numerous types of data exist in finance, we will focus on the one that comes from the prices of assets such as stocks or cryptocurrencies. Each of these individual prices generates a time series $\{p_t\}$, where p_t denotes the price of the given asset at (discrete) time index t. This index depends on the frequency of the data. We will say that the data is low-frequency if it has been gathered daily, weekly, monthly and so forth. On the other hand, we will say that the data is high-frequency if it has been gathered on an intraday basis, such as hourly, minute-by-minute or second-by-second.

2.1 Asset prices and returns

Basic modeling for these price time series does not use the regular prices of the assets as defined above. Instead, it uses the log-prices:

Definition 2.1.1. The log-price of an asset at time t is $y_t := \log p_t$

The most fundamental model describes the log-prices as a randow walk with drift: $y_t = \mu + y_{t-1} + \epsilon_t$, being ϵ_t the noise of the time series. The main downside of this model is that the random walk is not stationary. For this reason it is more appropriate to work with the asset returns:

Definition 2.1.2. The simple return (a.k.a. linear return) at time t is $R_t := \frac{p_t - p_{t-1}}{p_{t-1}}$

Definition 2.1.3. The log-return at time t is $r_t := y_t - y_{t-1} = \log \frac{p_t}{p_{t-1}}$

Under the assumption of the fundamental model described above, the log-returns now satisfy $r_t = \mu + \epsilon_t$, so we obtain the desired stationarity. It is also worth noting that $r_t = \log(1 + R_t)$, thus $r_t \approx R_t$ when $R_t \approx 0$. To exemplify how these concepts come into play, we plot in Figure 2.1 the daily log-prices and log-returns of the PepsiCo, Inc. (PEP) stock from 2002 to 2021:

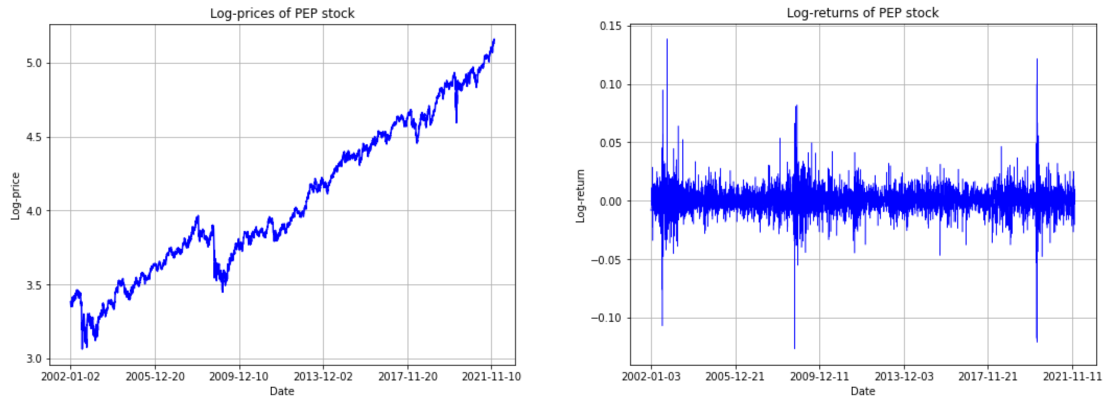

Figure 2.1: Daily log-prices and log-returns of PEP stock

In this type of financial data, not only there is a low signal-to-noise ratio, but there also are some particularities that one should take into account when working with it. These come from empirical observations of different financial markets throughout the years and they are only broad generalizations of the data, so we will refer to them as stylized facts. A detailed statistical analysis of them is done in [5]. The most relevant stylized facts to this project are summarized below.

Firstly, there is an absence of linear autocorrelations of asset returns, except for very small intraday scales when the noise microstructure starts to reveal itself. On the other hand, there actually exists autocorrelation of the absolute value of the returns, and it decays slowly as a function of the time lag. Secondly, there is a significant non-Gaussianity in the distribution of the log-returns, due to two main reasons. The first one is that these distributions are heavy-tailed, and the second one is the asymmetry: one can observe large drawdowns in stock prices but huge upwards movements are not that usual. These two properties can be observed in Figure 2.2.

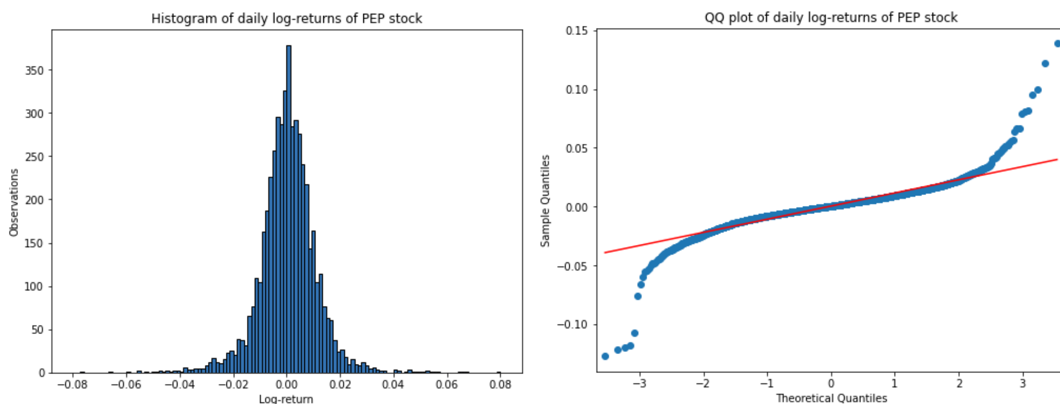

Figure 2.2: Histogram and QQ plot of PEP stock daily log-returns

The shape of the log-returns distributions are not the same at different time scales though, as with lower frequency data they resemble more and more a normal distribution. Another important stylized fact is the intermittency: the time series of the returns display irregular bursts that show

a high variability. Finally, it is observed that high-volatility events tend to happen close in time. This is known as volatility clustering and can be easily observed in the log-returns in Figure 2.1, where three main volatility clusters can be observed in 2002, 2008 and 2020.

In reality, we would not just deal with a single asset but with a universe of N assets. For this reason we denote the log-returns of the N assets at time t as a vector $r_t \in \mathbb{R}^N$ (for the simple returns, $R_t \in \mathbb{R}^N$). The goal is to model the vector r_t conditional on the previous historic data $\mathcal{F}_{t-1}$. Following [8], r_t follows a multivariate stochastic process with conditional mean and covariance matrix, denoted respectively as $\mu_t = \mathrm{E}[r_t \,|\, \mathcal{F}_{t-1}]$, $\Sigma_t = \mathrm{E}[(r_t - \mu_t)(r_t - \mu_t)^T \,|\, \mathcal{F}_{t-1}]$. For simplicity, one may assume that r_t follow an i.i.d. distribution, thus the conditional mean and covariance become constant: $\mu_t = \mu$, $\Sigma_t = \Sigma$. In practice, both μ and Σ have to be estimated using the past T observations. The simplest estimators are the sample estimators:

Definition 2.1.4. Given the vectors of returns $r_t \in \mathbb{R}^N$, $1 \leq t \leq T$, the sample mean estimator is defined as $\hat{\mu} = \frac{1}{T} \sum_{t=1}^{T} r_t$. The sample covariance is defined as $\hat{\Sigma} = \frac{1}{T-1} \sum_{t=1}^{T} (r_t - \hat{\mu})(r_t - \hat{\mu})^T$

The sample estimators turn out to be very noisy, specially the sample mean, leading to unacceptable errors when designing the portfolios. To correct this, more sophisticated and robust estimators exist, e.g., shrinkage estimators or Black-Litterman estimators, but they fall out of the scope of this project.

2.2 Portfolio basics

The next useful concept that must be defined are portfolios. Portfolios simply represent how a certain budget is allocated among a collection of financial products. We denote by B the capital budget in dollars, and by $w \in \mathbb{R}^N$ the vector of normalized dollar weights assigned to each asset, such that $\sum_{i=1}^{N} w_i = 1$. Other constraints could be specified for w such as cardinality, leverage or maximum position constraints. One constraint that will be present throughout the whole project is the long-only constraint ($w \geq 0$), which does not allow for short positions.

From the notation described above, it is immediate to verify that Bw denotes the dollars invested in each asset. From this follows:

Definition 2.2.1. Let Bw_i be the initial wealth of asset i at time $t-1$. Let $R_{i,t}$ be the linear return of asset i at time t. The portfolio return is defined as:

$$R_t^P = \frac{\sum_{i=1}^{N} Bw_i(1 + R_{i,t})}{B} - 1 = w^T R_t$$

From the last definition one can deduce that the expected return and variance of the portfolio are $w^T \mu$ and $w^T \Sigma w$, respectively. Moreover, we can define:

Definition 2.2.2. Let B be the initial budget. Assuming fully reinvestment at each time stamp, the Net Asset Value (a.k.a. wealth or cumulative return) is:

$$\text{NAV}_t = B \prod_{s=1}^{t} (1 + R_s^P)$$

Nevertheless, when trading financial instruments in real life one might wish to modify the portfolio occasionally to adapt to new market trends and maximize the benefits. Then the vector of weights will depend on the time index t, and we will denote it by w_t. In this case, exchanges impose some transaction fees to the practitioners every time an instrument is bought or sold. These fees are not negligible and can have a big impact on the final wealth of a given portfolio investment strategy, specially if the weights are rebalanced often. We can incorporate these transaction costs to our previous definitions as follows:

Definition 2.2.3. Given a transaction fee rate C, the portfolio return at time t becomes:

$$R_t^P = \sum_{i=1}^{N} \tilde{R}_{i,t}$$

$$\tilde{R}_{i,t} = w_{i,t-1} R_{i,t} - C \left| w_{i,t} \left(1 + w_{i,t-1} R_{i,t}\right) - w_{i,t-1} \right|$$

As expected, when $C = 0$ this last definition becomes the one above where transaction costs are not considered.

Until now we have been talking about portfolios in general terms, but it is interesting to introduce some well-known basic portfolios that will be of use later. Even though they are not widely used by practitioners as more refined strategies can provide better results, they will serve as benchmarks to compare the performance of our designed portfolios.

The first and most naive portfolio is known as Buy & Hold. It consists on selecting one asset and allocating the whole budget B to it for the whole investment period. One can use different criteria to make the choice. Following the notation described previously, this portfolio can be expressed as $w = e_i$, being e_i the canonical vector with a 1 on the i-th position. There is no diversification in this strategy.

Oppositely to the Buy & Hold, the Equally Weighted Portfolio (a.k.a. EWP, 1 over N portfolio or uniform portfolio) tries to exploit the benefits of diversification by allocating the capital equally across all assets. This can be expressed as $w = \frac{1}{N}\mathbf{1}$.

Although the expected return is very relevant when evaluating the performance of the model, one needs to control also the probability of going bankrupt, which is known as risk. This is where Markowitz's modern portfolio theory [16] comes into play. The most basic risk measure is the

variance of the returns. Usually, the higher the mean return, the higher the variance, and vice versa. The idea of Markowitz's mean-variance portfolio (MVP) is to find a trade-off between the expected return $w^T\mu$ and the risk of the portfolio given by the variance $w^T\Sigma w$ by solving the following convex optimization problem:

$$\begin{aligned} \max_{w} \quad & w^T\mu - \lambda w^T\Sigma w \\ \text{s.t.} \quad & \mathbf{1}^T w = \mathbf{1} \end{aligned}$$

where $\mathbf{1}$ denotes the all-ones vector and λ is a parameter that controls how risk-averse the investor is. By sweeping this parameter one can recover the whole trade-off curve. In practice, more constraints would be added to the problem defined above like sparsity, turnover or long-only constraints. Throughout the work developed in this project, whenever referring to the MVP we will assume the latter is included in the problem.

2.3 Performance measures

After having designed and backtested different investment strategies, it is necessary to have some ways to analyze and compare their performance. The most obvious one is to just look at the gains over the investment period:

Definition 2.3.1. The annualized non-cumulative return is $K\mathrm{E}\left[R_t^P\right]$, where K denotes the number of trading periods in a year.

Note that the metric is annualized in order to be able to compare the results from strategies that work with data of different frequencies or investment periods of different length.

However, as we mentioned before when stating Markowitz's modern portfolio theory, it is not enough to look only at the final wealth obtained if the probability of going bankrupt is different between the strategies. The simplest way to quantify this is by looking at the standard deviation (or variance) of the returns:

Definition 2.3.2. The annualized volatility of an investment is given by $\sqrt{K\mathrm{Var}\left[R_t^P\right]}$, where K denotes the number of trading periods in a year.

The main problem of using the volatility as a risk measure is that it penalizes both the unwanted high losses and the desired high gains (as they both deviate considerably from the mean), whereas most investors would not mind this upside risk. One measure that takes into account only the returns below the mean is the semi-variance:

Definition 2.3.3. The annualized semi-variance of an investment is $\sqrt{K}\mathrm{E}\left[\left(\left(\mathrm{E}[R_t^P] - R_t^P\right)^+\right)^2\right]$, where K denotes the number of trading periods in a year and $(\cdot)^+ = \max\{0, \cdot\}$.

The last risk measure we will use is the drawdown, which calculates the decline from the historical peak of the cumulative return:

Definition 2.3.4. Let $X(t)$ be the cumulative return at time t. The drawdown at time t is defined as:

$$DD(t) = \frac{HWM(t) - X(t)}{HWM(t)}$$

where the High Water Mark is given by

$$HWM(t) = \max_{1 \leq \tau \leq t} X(\tau)$$

We define the maximum drawdown over an investment period of length T as

$$MDD = \max_{1 \leq t \leq T} DD(t)$$

Comparing the performance of different portfolios by looking at their final wealth and their risk separately might not allow to extract clear information. For this reason, some metrics have been defined that combine both and give a more complete summary of the overall performance. One of these is the Sharpe Ratio X[21], which calculates the expected gains per unit of risk:

Definition 2.3.5. The annualized Sharpe Ratio is defined as

$$\sqrt{K} \, \frac{\mathrm{E}\left[R_t^P\right] - r_f}{\sqrt{\mathrm{Var}\left[R_t^P\right]}}$$

where K is the number of trading periods in a year and r_f is the risk-free rate.

In the subsequent experiments of this project we will always take $r_f = 0$ as there does not exist any risk-free investment in the environments considered. It is worth noting that, as stated in [14], this way of annualizing the Sharpe Ratio only holds true for i.i.d. returns and under our assumption it is just an approximation that may lead to some notable errors. Nonetheless, as we will use this measure only to compare performances of portfolios tested under the same universe, the serial correlations may impact the Sharpe Ratio of each portfolio in a similar way. Hence, to make things simple we will stick to the definition given above.

Chapter 3

Literature review

Modern portfolio theory (MPT), introduced in 1952 by Markowitz [16], still plays an important role both in research and in practice. Despite its popularity, it faces a great number of limitations as it makes some assumptions that are not obeyed in real financial markets. Mainly, MPT relies on the fact that the returns follow a Gaussian distributions and thus investors only need to care about the expected return and variance of the portfolio returns to make their decisions. However, empirical observations prove that the distributions of returns are fat-tailed and asymmetric [5].

One idea to overcome these issues is to either improve the way the estimators are calculated by using shrinkage, robust estimators or the Black-Litterman model [2], or by reformulating the optimization problem to make it robust [6, Chapter 20].

Latest trends on the topic, however, leave aside all these ideas and focus on neural networks. This approach goes back to more than 20 years ago, as [17, 18] already proposed the first end-to-end framework via a very simple feed-forward structure, even without any hidden layer. They only focused on the univariate case, optimizing the performance for a single asset, with decent results. However, there is little discussion on how their models can be extrapolated to portfolio optimization. Moreover, their testing period goes from 1970 to 1994 so the market conditions back then are not comparable to the current ones.

Subsequent approaches to portfolio optimization via deep learning have been devoted to directly forecast the prices or returns of the assets by using different types of neural networks (such as multilayer perceptrons, convolutional networks or LSTMs) that take as input the historical prices, returns or any kind of alternative data [9, 15]. These models are straightforward to implement as they are simply supervised learning regression problems. Nonetheless, future market prices or returns are difficult to predict due to their low signal-to-noise ratio so in general these type of approaches perform poorly. Additionally, these methods only forecast the future prices, but they

do not specify how these predictions are turned into market actions. Another layer of logic is required for this reason, which may make the models less versatile.

The last two papers mentioned, together with a big part of the literature published on this area, contain some major flaws in their backtests that make the results difficult to interpret. The first and most important one (present in both papers) is not comparing the models implemented to standard benchmark portfolios such as the EWP or any other more refined strategies actually embraced by practitioners. It is impossible to get the big picture of the situation if it is just stated that the models provide good returns without showing how non-deep learning models would have performed in the same situation, because the architectures implemented may not be better than those. Another observed flaw is using cutting-edge technologies on data from decades ago when this technologies did not exist and the market conditions were totally different from nowadays. Hence, it is way easier to beat the markets by using them. This happens, for example, in [9], where LSTM networks are used to trade in the 1990s stock market. The last flaw, more difficult to spot, is incurring in a look-ahead bias, meaning that at each timestep information from the future that would not have been available at that moment is used to simulate a trading decision. Thus can happen for multiple reasons and one must be very careful when implementing the models to avoid it as it can completely change the results.

Building networks that output a portfolio allocation instead of a price forecast is an even newer approach. The latest developments are done in [23] and continued in [22]. The former presents an end-to-end framework that bypasses the traditional forecasting steps and allows the models to determine the portfolio weights by updating the parameters of a neural network optimizing the Sharpe ratio. The portfolios designed under this framework do not allow shorting and are budget constrained. The latter paper extends the former by including two new loss functions that can be optimized apart from the Sharpe Ratio (variance and mean-variance) and by allowing to incorporate other constraints to the portfolios designed such as leverage, maximum position and cardinality. Even though the first results on these papers seem promising, they are tested considering very low transaction fees (0.01% and 0.02%). Greater fees, more similar to what we would encounter when trading in real life) may cancel out all profits and forbid these methods to be taken into production.

Lastly, the idea of combining the power of neural networks to the knowledge of modern portfolio theory and mean-variance portfolios (instead of designing end-to-end models containing only neural networks) is still in a very early stage. The first (and only, so far) detailed study of this approach is done in [3]. The idea is to let μ and Σ of a mean-variance portfolio be a function of the input returns with some trainable parameters, and then learn this parameters by updating them according to how the corresponding MVP performs. In the work carried out in this thesis we explore both approaches: end-to-end models and architectures that include the MVP in the training process.

Chapter 4

Preliminary experiments

It is well-known that deep learning is nowadays a very powerful technique able of providing solutions to problems that were incredibly difficult to solve before. But it is also true that it is a really complex tool that requires complete understanding of its intricacies and how the problem itself can be approached in this way. For this reason, before going into the deep learning ideas we deem necessary to analyze in detail if the portfolio optimization problem can be simplified somehow, and, if so, whether classical and less complicated techniques already work for this easier formulation.

4.1 Univariate sizing

Classical portfolio optimization deals with a set of N assets and tries to find the best budget allocation $w_t \in \mathbb{R}^N$. This is a multivariate problem where one does not only have to understand how the historical data affects the future returns, but also how the returns of the assets interact among them. One way to get rid of this is to look at each asset independently, so that the universe becomes a single asset ($N = 1$). We now have a univariate problem, as $w_t \in \mathbb{R}$. Since the constraint $\sum_{i=1}^{N} w_i = 1$ does not make sense under this universe, the goal is not to decide how we distribute our budget, but to determine which portion of the budget are we investing on the asset and which portion are we holding in cash (meaning that it does not change) at every timestep. This is known as the sizing problem. Ideally, we would like to invest the whole budget on the asset whenever the returns are positive ($w_t = 1$) and to sell the asset as soon as the returns become negative ($w_t = 0$). This would make us obtain the maximum cumulative return possible with a drawdown of 0.

The main question of the sizing problem is how do we decide the sizing w_t at each timestamp. As the signal-to-noise ratio of the returns is very low, forecasting them given the past ones is not an easy task. In the following subsections we will explore different approaches.

4.2 Experiments on daily data

4.2.1 Dataset

The initial dataset used for this purpose contains the daily adjusted close prices of 456 component stocks of the S&P500 index from July 2013 to June 2021, providing 2000 timestamps of data. This dataset has been downloaded from Yahoo Finance. The reason why not all 500 stocks are included in the dataset is that some of them did not exist since 2013 so they had an unacceptable number of consecutive missing values at the beginning. These stocks with incomplete data were not considered for the subsequent experiments, remaining in this way the aforementioned 456. No other preprocessing tasks were carried out at this point.

4.2.2 Moving average-based strategies

The first methods that we will introduce rely on one of the most basic indicators used by practitioners to decide when to trade. If, for each time stamp, we calculate the average of the prices over a past lookback period, we obtain a smoothened version of the price evolution with some delay. For this reason it is considered that moving averages might be useful to describe the tendency of the prices getting rid of part of the noise. The number of prices we consider as a lookback is very relevant, as the longer the lookback the smoother and more lagged the plot is going to be. For illustrative purposes, we show below the moving averages we obtain using the past 8 and 40 daily prices as a lookback for the HOLX stock from March 2019 to June 2021:

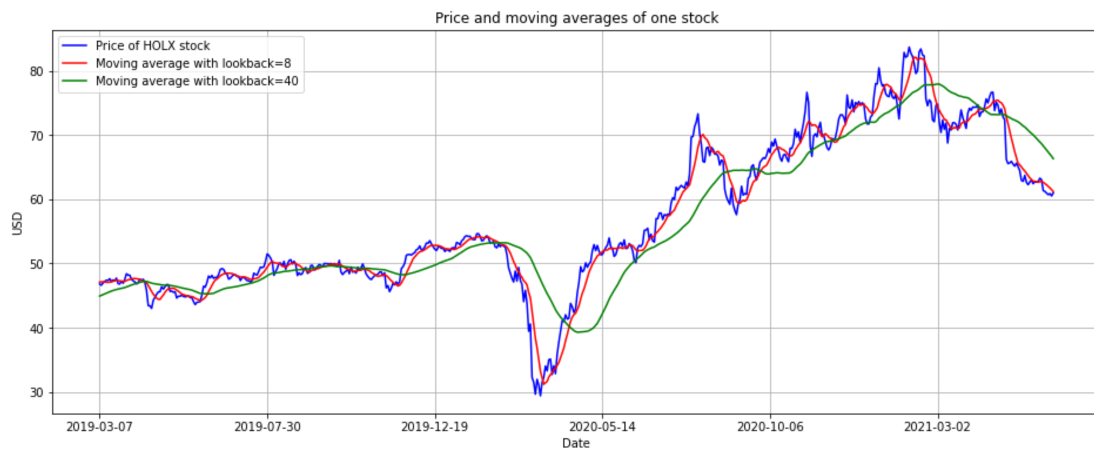

Figure 4.1: Price and moving averages of HOLX stock using different lookbacks

Two different ways of deciding the sizing based on moving averages have been explored. They are both binary, meaning that the sizing will be either 0 or 1 so that we either hold all our budget in cash or via the asset. The idea is to emulate, in a more formal way, the simplest idea one could think of when trying to trade: buy when the price seems to raise, sell when it is going down. How we determine when the tendency of the price is uprising or not is the main problem here.

The first method implemented simply fixes the length of the lookback period and looks at the crossovers of that moving average and the price of the asset. Whenever the price of the asset is above the moving average we hold the asset. Thus, the sizing is defined as:

$$w_t = \begin{cases} 1 & \text{if } \frac{1}{l}\sum_{i=1}^{l} p_{t-i} < p_{t-1} \\ 0 & \text{otherwise} \end{cases}$$

The second method intends to mitigate the negative effect of the noise that can appear on the first one, since abrupt changes on the price would provoke undesired switches on the sizing. So instead of comparing a moving average to the actual price, we calculate two different averages, one with a significantly shorter lookback than the other. We refer to them as fast and slow moving averages. Again, the asset is hold whenever the fast moving average is above the slow one:

$$w_t = \begin{cases} 1 & \text{if } \frac{1}{l_1}\sum_{i=1}^{l_1} p_{t-i} < \frac{1}{l_2}\sum_{i=1}^{l_2} p_{t-i} \\ 0 & \text{otherwise} \end{cases}$$

In the experiments carried out, we used a lookback of $l = 10$ days for the simple MA, and $l_1 = 25$, $l_2 = 7$ for the slow and fast ones, respectively. The returns after applying the sizing are calculated as $R_t^s = w_{t-1}R_t$ (without taking into account transaction costs). We look at the annualized non-cumulative return, the maximum drawdown and the Sharpe Ratio evaluated on these R_t^s to analyze their performance. Each point in the boxplots corresponds to the evaluation of the respective measure on the returns R_t^s of one individual asset along all the investment period.

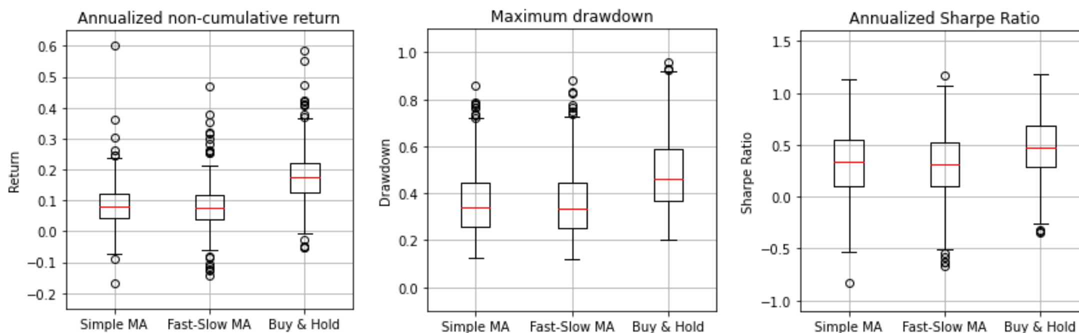

Figure 4.2: Performance of the moving average-based sizing strategies

Clearly these two methods cannot be regarded as appropriate sizing strategies as they perform far worse than the Buy & Hold benchmark. To understand better what is happening we plot below the evolution of wealth over time when applying this sizing throughout the whole investment period.

Figure 4.3: Wealth over time for TSN stock after applying sizing via a simple MA

It is easy to identify from Figure 4.3 when the changes in sizing occur. Ideally, the flat periods should coincide in time with the downfalls of the Buy & Hold plot. However, the lag introduced by the moving averages is so large that sometimes they end up happening mainly during the next uprising period. This means that not only we do not avoid losing money in these downfalls, but also we do not fully exploit the price rises.

4.2.3 Volatility targeting

Since volatility is easier to forecast due to its clustering nature, a more reasonable strategy to find a sizing that improves the benchmark is trying to reduce it. One of the most typical ways to do so is called volatility targeting. The main idea is to choose from the beginning what is the maximum volatility we can tolerate, and then adjust our position accordingly so that we do not exceed it. We estimate the volatility by fixing a lookback and taking the standard deviation of the returns during this lookback. Given a target V of the maximum amount of annualized volatility we want to allow and a lookback l, the sizing is calculated as follows:

$$w_t = \max\left\{ \frac{V}{\sqrt{252 \cdot \frac{1}{l}\sum_{i=1}^{l}\left(r_{t-i} - \frac{1}{l}\sum_{i=1}^{l} r_{t-i}\right)^2}}, 1 \right\}$$

Whenever the recent volatility is lower than the target, we allocate the full budget to the asset. If it is greater, we decrease the sizing proportionally. The value of the lookback has to be chosen carefully since a lookback too large might not be able to capture the high-volatility events, while a lookback that is too short will not get rid of the intermittences and may lead to unpredictable

results. To decide which lookback to use in our experiments, we verified visually that the moving volatilities smoothened properly the plot of the log-returns for a few assets on all the investment period as shown in Figure 4.4.

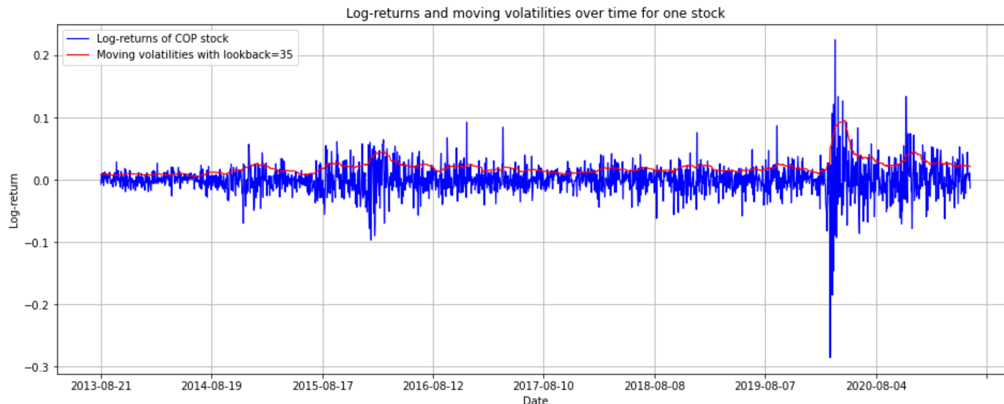

Figure 4.4: Log-returns and moving volatility of COP stock

The line in red shows the moving volatility using 35 days as a lookback, which is the value we finally chose. It can be observed that it is not too noisy and that after a cluster it goes quickly back to the baseline as desired. It is worth noting that this method of deciding the lookback is not correct as it incurs in a look-ahead bias: a decision made considering all the future data is used at every timestamp to test our strategy. The proper way to do this would be by having extra data previous to the investment period used only for this purpose. However, the scarcity of data made this a bit difficult and all in all it is not a decision that has a huge impact on the outcome.

To test the viability of volatility targeting, 4 different targets have been tried (without accounting for transaction costs). This time we use the annualized non-cumulative return, volatility and Sharpe Ratio to measure the performance of the strategy, plotted in Figure 4.4.

The annualized volatility boxplot proves that the forecasts done are pretty accurate since in generally the volatilities remain below the targets. As we make the targets smaller, the returns also become smaller proportionally so when balancing returns and risk via the Sharpe Ratio we observe that they all perform similarly, even when comparing to the benchmark. That was expected for the higher targets as in these cases the sizing stays at 1 for most of the time (hence it replicates the Buy & Hold), but it was not so clear how the lower targets would perform. Although the results may make us think that volatility targeting is a suitable strategy for the more risk-averse investors, it must be remarked that transaction costs have not been considered. Therefore, in real life the average return would decrease providing Sharpe Ratios probably below the Buy & Hold.

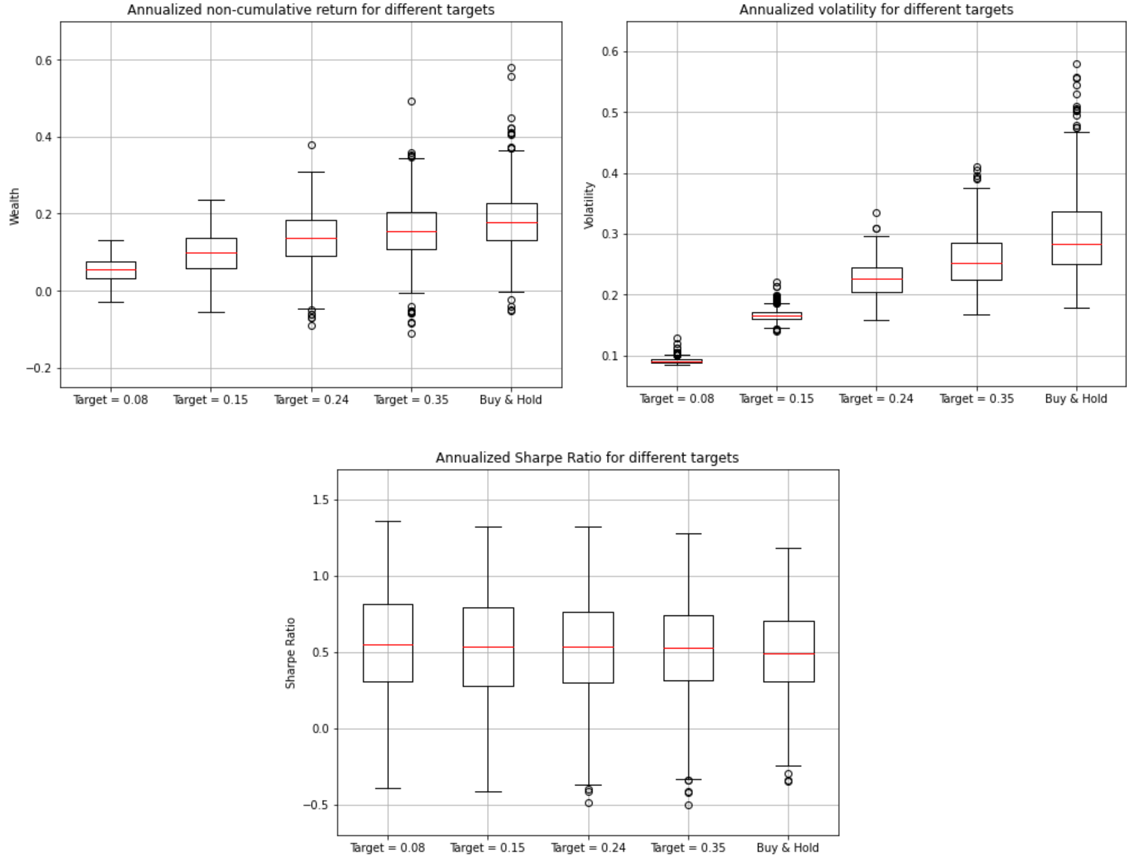

Figure 4.5: Performance of the volatility targeting sizing strategies

4.2.4 Machine learning models

Since the basic heuristic strategies did not provide good results, the next logical step is to dig into classic supervised machine learning algorithms. At time $t-1$, we want to predict the optimal sizing for time t by looking at different indicators calculated on the log-returns from time $t-l$ to $t-1$. As input for the algorithms, 10 features have been used. They correspond to the following 5 indicators calculated with two different lookbacks, $l = 15$ and $l = 40$. If we denote by r the set of log-returns $\{r_{t-l}, r_{t-l+1}, \ldots, r_{t-1}\}$, then the features are defined as:

- Standard deviation: $x_{1,t} = \sqrt{\mathrm{Var}\left[r\right]}$
- Semi-variance: $x_{2,t} = \mathrm{E}\left[\left((\mathrm{E}[r] - r)^{+}\right)^{2}\right]$
- Signal strength: $x_{3,t} = \frac{\mathrm{E}[|r|]}{\sqrt{\mathrm{Var}[r]}}$
- Sharpe Ratio: $x_{4,t} = \frac{\mathrm{E}[r]}{\sqrt{\mathrm{Var}[r]}}$
- Probabilistic Sharpe Ratio: $x_{5,t} = \mathrm{Z}\left[\frac{(\hat{S}-S^{*})\sqrt{l-1}}{\sqrt{1-\hat{\gamma}_3\hat{S}+\frac{\hat{\gamma}_4-1}{4}}}\right]$

where $\hat{S}$ denotes the observed Sharpe Ratio (calculated as $x_{3,t}$), S^* a user-defined benchmark Sharpe Ratio, $\hat{\gamma_3}$ and $\hat{\gamma_4}$ the skewness and kurtosis of the returns in r, respectively, and Z denotes the cumulative distribution function of the standard normal distribution. This measure was introduced in [7, p. 203] and it intends to provide an adjusted estimate of the actual Sharpe Ratio by removing the inflationary effect caused by short series with skewed or fat tails. This measure estimates the probability that $\hat{S}$ is greater than a hypothetical S^*. In our experiments we will take $S^* = 0$.

Apart from the input features, we have to decide which labels we use to feed the algorithms. Two options have been explored. The first one is to use as labels the ideal sizings: allocating the whole budget to the asset when the returns are positive and holding it in cash when they are negative:

$$y_t = \begin{cases} 1 & \text{if } R_{t+1} > 0 \\ 0 & \text{otherwise} \end{cases}$$

Nevertheless, predicting the sign of the returns is already quite a difficult task because of the noise present in financial data. One idea to overcome this issue would be to set a threshold greater than 0 that the returns have to surpass so that their label becomes 1. The main downside of this idea is that we would be exposed to the volatility cluster: in events of high volatility a big part of the returns would be above this threshold while in more stable periods we would be aiming too high and the sizing would remain at 0 for most of the time. This is precisely the contrary of what we want. One solution for this is described in [7, p. 44] and consists of making this threshold dynamic, increasing in periods of high volatility and getting close to zero when the returns are more trustworthy. The resulting labels can be expressed as:

$$y_t = \begin{cases} 1 & \text{if } R_{t+1} > \delta \sqrt{\frac{1}{l} \sum_{i=1}^{l} \left(R_{t-i} - \frac{1}{l} \sum_{i=1}^{l} R_{t-i} \right)^2} \\ 0 & \text{otherwise} \end{cases}$$

The δ factor indicates how conservative we want the sizing to be and must be chosen carefully. If it is too small the labels would be very similar to the ones defined previously, while if it is too large the labels would only be 1 when some positive outliers happen. We will take $\delta = \frac{2}{3}$

5 different machine learning algorithms have been implemented:

- **Linear regression**

- **Linear regression with L_1 regularization** (a.k.a. LASSO). The α regularization hyper-parameter is learnt testing values between 10^{-6} and 10^2 on a validation set between the training and test splits.

- **Linear regression with L_2 regularization** (a.k.a. ridge regression). The α regularization hyperparameter is learnt testing values between 10^{-6} and 10^{2} on a validation set between the training and test splits.
- **K-Nearest Neighbors** regressor using 200 neighbors.
- **Random forest** regressor with 50 trees

For the algorithms that need a loss function to be trained, we use the mean squared error respect to these labels.

To train the machine learning models we need to split our data into training and test sets. To make the most out of the temporality of the data, we will apply an anchored walk-forward approach where each window contains 6 months of testing data. This means that on the first window we use the first 6 months of data to train the model parameters and we use them for "real-life" trading in the next 6. After that, we retrain the model on the first 12 months of data to apply the optimal learnt parameters on the months 13-18. As the training set grows in size, we use linearly decaying weights so that the errors in predicting the most recent samples have a bigger impact on the overall loss function.

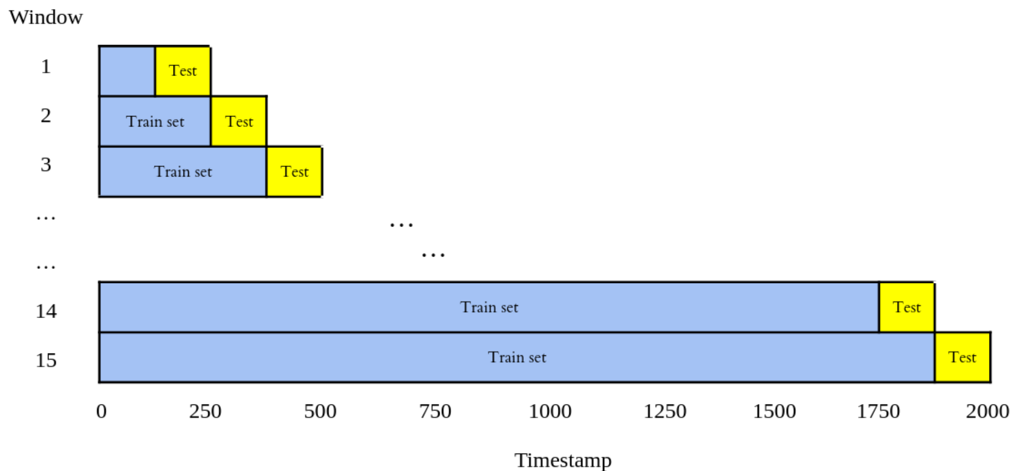

Figure 4.6: Anchored walk-forward approach to train the ML models

The tests on these algorithms have been carried out without including transaction costs. To understand how they perform, in this case it is enough to look at the annualized volatility and annualized Sharpe Ratio boxplots. Figures 4.7 and 4.8 show the performance obtained by each algorithm with the labels based only on the sign and with the labels that account for the volatility, respectively.

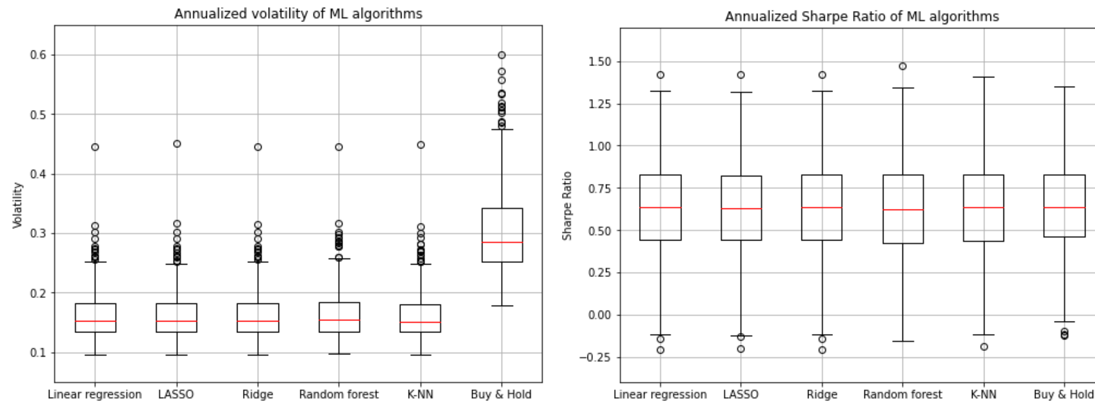

Figure 4.7: Performance of ML algorithms for sizing with fixed labeling thresholds

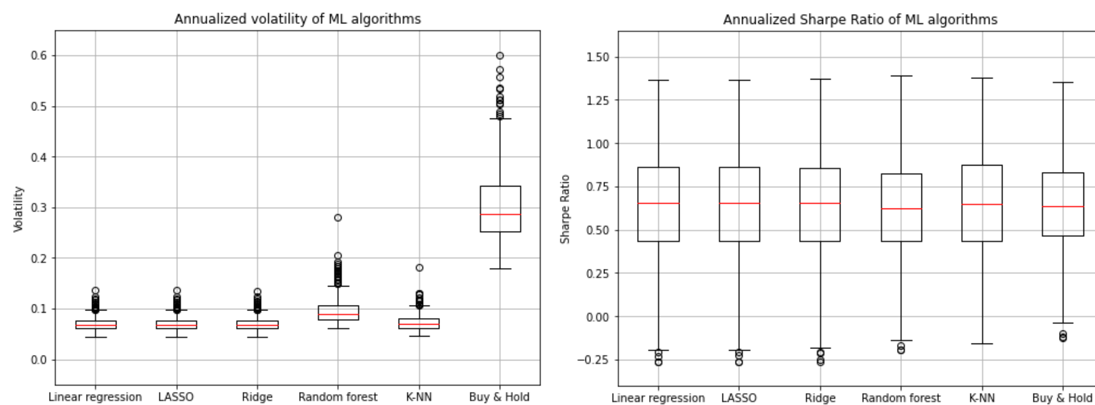

Figure 4.8: Performance of ML algorithms for sizing with dynamic labeling thresholds

All 5 models provide very similar results when using the same labels. They end up achieving Sharpe Ratios nearly identical to the benchmark and in that sense there is not much difference between both labelings. However, the volatilities when using the dynamic threshold are smaller. That was expected as in this case we intend to buy the asset less often, only in periods of low volatility or when the returns are huge. The fact that the models are able to obtain the same risk-adjusted returns than the Buy & Hold but reducing the risk is the same we observed when determining the sizing via volatility targeting. Since the moving volatilities were two of the input features for the algorithms, this may indicate that the machine learning is mostly learning to decrease the volatility to obtain decent performances. That is another proof that the returns are incredibly more difficult to predict than the volatility and supports the stylized facts mentioned in Section 2.1 related to the noisiness of the data and the clustering nature of the high volatility events.

4.2.5 Direct reinforcement learning

The last sizing strategy designed is based on what was called direct reinforcement learning,. We will follow a simplified version of the framework described in [17]. The key idea is to define the sizing as a non-linear function of the past lookback returns and the last sizing (in case we want to

include transaction costs in the training), with some parameters that will be trained to optimize the Sharpe Ratio of the in-sample data. In order to get a testing environment comparable to the previous ones, we have to slightly modify the strategy described in the reference. Mainly, our model uses as input the log-returns instead of price differences (so that we allow to trade portions of the financial products instead of fixed units), and we do not allow shorting. All in all, the sizing becomes a logistic curve (instead of a tanh as in the reference) of a linear combination of the past returns and the last sizing:

$$w_t = \left(1 + e^{-\theta^{\mathrm{T}} u_t}\right)^{-1}$$

where $u_t = (1, R_{t-l}, R_{t-l+1}, \cdots, R_{t-1}, w_{t-1}) \in \mathbb{R}^{l+2}$ and $\theta \in \mathbb{R}^{l+2}$ denote the parameters we want to learn. These parameters are optimized via a gradient ascent that maximizes the Sharpe Ratio. Firstly we backtest the model without taking into account transaction fees so that it is comparable to the previous strategies implemented. In this case u_t becomes $u_t = (1, R_{t-l}, R_{t-l+1}, \cdots, R_{t-1}) \in \mathbb{R}^{l+1}$ and $\theta \in \mathbb{R}^{l+1}$.

If we denote by T the length of the investment period, $R_t^s = w_{t-1} R_t$, $A = \frac{1}{T}\sum_{t=l}^{T} R_t^s$, $B = \frac{1}{T}\sum_{t=l}^{T} R_t^{s\,2}$, then $S = \frac{A}{\sqrt{B-A^2}}$. In the case where the model is trained without considering transaction costs, the gradient of the Sharpe Ratio S respect to the $\theta \in \mathbb{R}^{l+1}$ vector can be obtained analytically after successive applications of the chain rule:

$$\frac{dS}{d\theta} = \frac{dS}{dA} \cdot \frac{dA}{d\theta} + \frac{dS}{dB} \cdot \frac{dB}{d\theta} = \sum_{t=l}^{T} \left(\frac{dS}{dA} \cdot \frac{dA}{dR_t^s} + \frac{dS}{dB} \cdot \frac{dB}{dR_t^s} \right) \frac{dR_t^s}{d\theta} =$$
$$= \sum_{t=l}^{T} \left(\frac{S(1+S^2)}{A} \cdot \frac{1}{T} + \frac{-S^{\frac{3}{2}}}{A^2} \cdot \frac{2}{TR} \right) \frac{R_t \left(1 - (2w_{t-1} - 1)^2\right)}{4(1 + w_{t-1} R_t)} \cdot u_t$$

Since this method requires learning some parameters, the data must be split into training and test subsets. We will use a rolling-window approach: we begin using the first 1000 days of data to find the optimal θ in this period and we use them to trade from day 1001 to day 1100. Then we train again the model using as a train set the data from day 100 to day 1100 and the learned parameters and tested on the following 100 days. This is repeated 10 times until we have traded on all the last 1000 days of data we have. It is worth noting that this way of training the model taking advantage of the temporal component of the data causes that we only have half of our original data to check the performance of the model. Figure 4.9 illustrates how this rolling window is used.

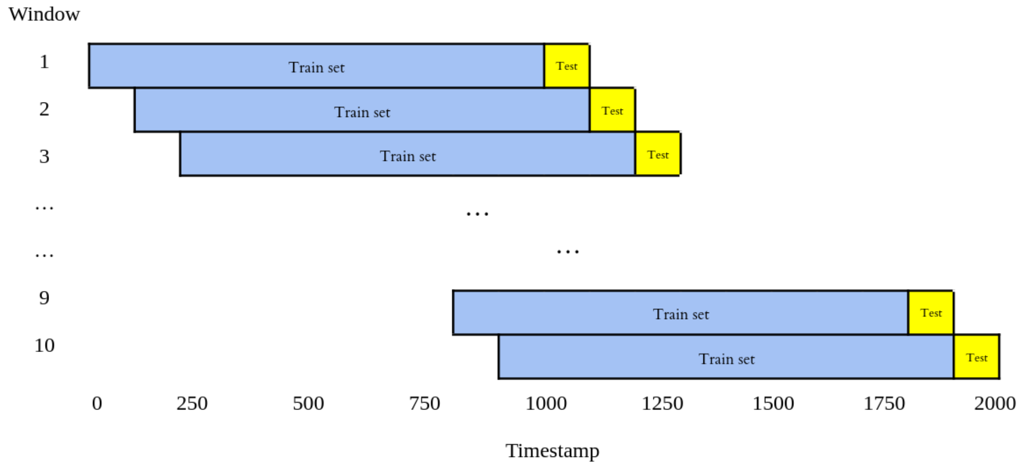

Figure 4.9: Rolling-window approach to train the direct RL model

The subsequent algorithm describes how the training is carried out inside each window.

Algorithm 1 Training of the direct reinforcement learning trader at each window

---

Input: Training return series r with length T, learning rate η, lookback l, number of epochs N
Output: Model parameters θ
$\theta_0 \leftarrow 0$
$\theta_{1..l} \leftarrow \mathcal{U}(0, 1)$
for 1 **to** N **do**
  for $t \leftarrow l$ **to** T **do**
    $w_t \leftarrow \left(1 + e^{-\theta^{\mathrm{T}} u_t}\right)^{-1}$
    $R_t^s \leftarrow w_{t-1} R_t$
  end for
  $S \leftarrow \frac{\mathrm{E}[R_t^s]}{\sqrt{\mathrm{Var}[R_t^s]}}$
  $\theta \leftarrow \theta + \eta \frac{dS}{d\theta}$
end for

In the experiments carried out we use $N = 2000$ epochs, a lookback of $l = 20$ days and a learning rate of $\eta = 0.3$. Before going in depth in the results obtained, it is important to verify that the model is actually learning and maximizing the Sharpe Ratio in the training data.

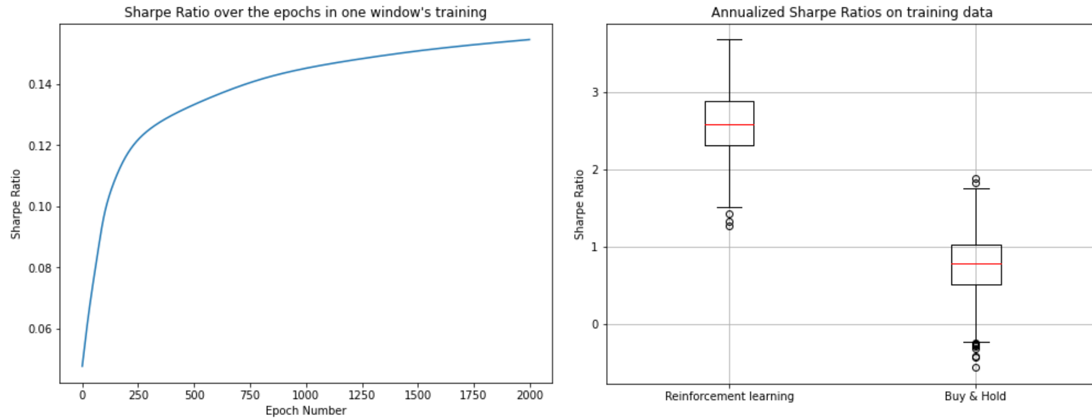

Figure 4.10: Proof of convergence of the reinforcement learning trader

On the left of Figure 4.10 we can see how the Sharpe Ratio increases with the number of epochs until converging for the first training window of a random asset. This means that the number of epochs and the learning rate chosen were suitable for our environment. The boxplots on the right prove that the Sharpe Ratios calculated on in-sample data for the first training window of all assets are significantly higher than the ones obtained on the same data with the Buy & Hold benchmark. Thus, we can affirm that the direct reinforcement learning is training as it is supposed to.

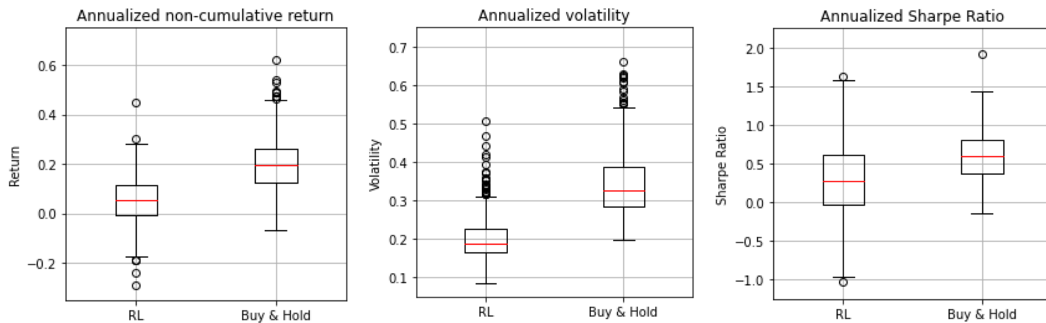

Figure 4.11: Performance of the reinforcement learning trader on daily data

The boxplots in Figure 4.11 tell us that the RL trader could not learn parameters able to perform well enough in the out-of-sample data as the Sharpe Ratios obtained are far below the benchmarks', with some important negative outliers. The only reason that could explain this behaviour is a high degree of unpredictability and the fact that past returns do not affect the future in the same way every time. If the trader wanted to perform better on the test data it should have learnt at least to replicate the benchmark by setting θ_0 big enough and the rest of parameters to 0. However, forcing it to maximize the Sharpe Ratio in the in-sample data provoked this poor performance.

4.3 Experiments on minute-by-minute data

The results obtained for the direct reinforcement learning on stock daily data were far worse than the original references suggested. Nonetheless, recently the same ideas were tried but using high-frequency cryptocurrency prices instead [11]. The higher predictability observed in this reference agrees with numerous studies, such as [4], that show that cryptocurrency prices do not strictly follow a random walk since all the speculation behind make them somehow influenced by sentiment. Because of that, we decided to replicate ourselves the same tests on this type of data.

4.3.1 Dataset

Following the aforementioned reference, these experiments were conducted on a dataset containing the close prices of 6 of the main cryptocurrencies (BTC, ETH, LTC, ADA, XRP and XMR) from 01 January 2021 to 11 November 2021, accounting for around 450,000 minutes of data. It was downloaded from Binance and did not contain missing values so no preprocessing was needed.

4.3.2 Direct reinforcement learning

In order to be able to apply the same framework described in Section 4.2.5 we need series containing only 2000 minutes of data. For each cryptocurrency we created 200 series of that length by splitting the 450,000 minutes in subsets of 2240 consecutive minutes and then getting rid of the last 240 minutes of each. In this way there is no overlap between series and the 4-hour gap between them makes the backtests more independent. We will evaluate the performance measures on each of these 1200 series separately, after applying the same rolling-window approach and training the θ parameters for each window as we did with the daily stock data.

Since we have not changed the implementation of the algorithm respect to the last tests with daily data, there is no need to check again that the model is training correctly and converging with the hyperparameters we chose. Nonetheless, it is still interesting to analyze how the Sharpe Ratios evolve over the epochs both when evaluated on training and test data. We chose 6 random series out of the 200 obtained from the cryptocurrency XMR and at each epoch we calculated the Sharpe Ratio on training and test data of the first window with the parameters found up to that point.

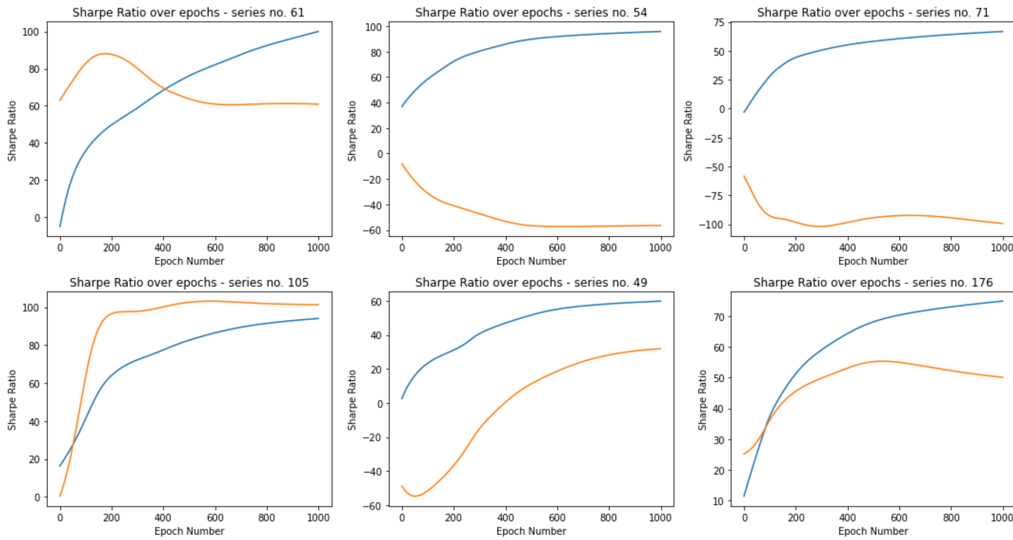

Figure 4.12: Sharpe Ratios over epochs for training and test data of 6 random series

Blue lines in Figure 4.12 show the Sharpe Ratios in training data and the orange ones in test data. In an ideal case, the orange lines qould keep increasiong along the 2000 epochs and converging at the end (despite being a bit more noisy). What we observe though is that in some cases a peak ratio is achieved quite before finishing the 2000 epochs, and in others the learning is totally useless as the test Sharpe Ratios keep decreasing since the beginning. These two cases are not specially worrying though as negative Sharpe Ratios are difficult to analyze: they can worsen either by providing more negative returns or less volatility. While the former is to be avoided, the latter may be desired. But the first and last plots indicate that sometimes our method overfits and is not capable of extrapolating the information learnt on training data to the future.

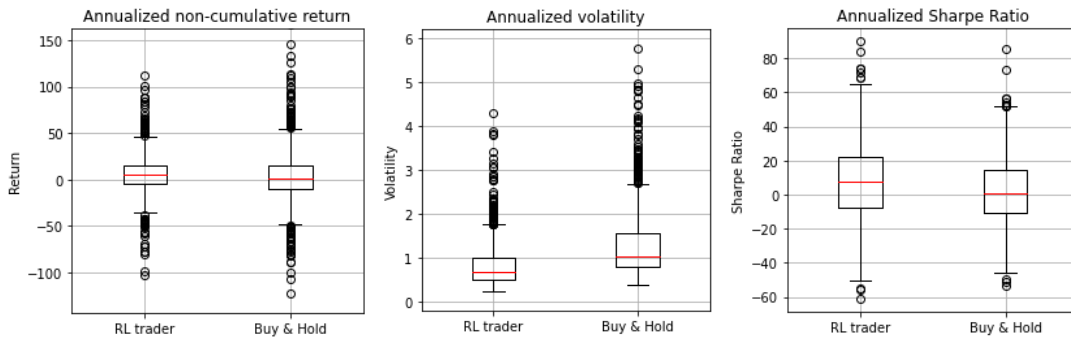

Figure 4.13: Performance of the reinforcement learning trader on minute-by-minute data

The results in Figure 4.13 differ a lot from the ones we got from daily data as now the RL trader outperforms the benchmark in terms of risk-adjusted measures. Also, a close-up at the return boxplot reveals that the median is now quite above the benchmark (5.2 for the RL trader against 0.8 for the Buy & Hold). Since the model appears to perform really well on these backtests, it is necessary to analyze what happens when we account for transaction costs, simulating a real-life trading environment.

We take a transaction fee of $C = 0.001$ and we incorporate the associated costs (and the previous sizing) in the training as described in Section 4.2.2.4. All the improvements that we observed above now vanish as it can be seen in the results in Figure 4.14.

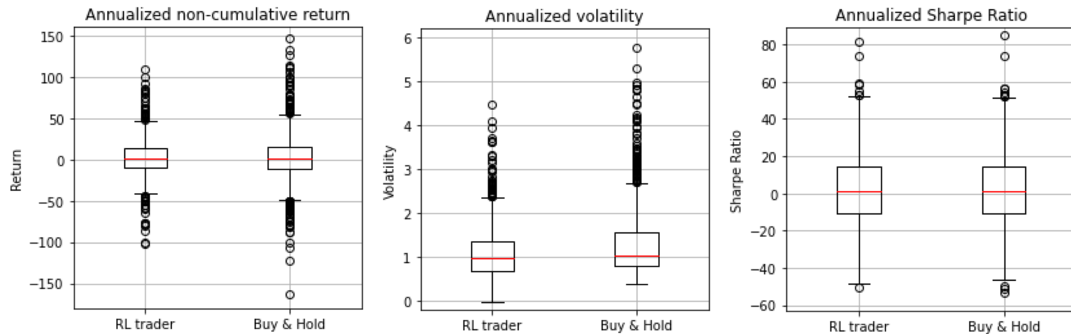

Figure 4.14: Performance of the reinforcement learning trader on minute-by-minute data (with transaction costs)

Since now the big sudden changes in the sizing are penalized by the transaction costs, the model can not fully learn how to exploit the predictability on the returns while keeping similar portfolios from one timestep to the next. The median of the annualized returns has gone down from 5.2 without transaction costs to 0.7 now and the differences in terms of risk have reduced. Hence, the Sharpe Ratios become more equal to the benchmark, both in terms of median of all the backtests (around 0.8 both for the RL trader and for the Buy & Hold) and in terms of outliers.

4.4 Overall assessment

The simulations carried out in this chapter confirm that portfolio optimization is a difficult problem that neither it can be broken down into univariate easier subproblems nor it can be approached successfully with classical techniques. When dealing with daily data, the methods implemented were not capable of clearly beating the Buy & Hold benchmark, even without taking into account the transaction fees. When switching to higher frequency data, it seemed that the reinforcement learning trader could at least detect some predictability in the data although in real life trading the transaction costs would wipe out all the gains. These results reinforce our initial hypothesis stating that we need to look for more complex techniques such as deep learning if we want to obtain better results and mark our next steps in this project.

Chapter 5

Main experiments

The results obtained in the tests done in Chapter 4 proved that predicting the behaviour of financial products is not an easy task and it can not be reduced as applying some heuristic techniques to univariate time series of prices or returns. Therefore, in this chapter we go one step further and we carry out the first experiments with deep learning models for multivariate portfolio optimization. The core idea is that these models output at each timestep the optimal portfolio allocation $w_t \in \mathbb{R}^N$, and these allocations define the portfolio we hold for a given period.

5.1 Dataset

Since obtaining a great amount of low-frequency data requires using stock data from more than 20 years ago (when the market conditions were not comparable to nowadays), the last experiments in Section 4.3 indicate that on minute-by-minute data it is possible to obtain better results, and minute-by-minute data is easier to obtain if it is from cryptocurrencies, in these experiments we use a dataset of minute-by-minute cryptocurrency prices. Specifically, the dataset contains close prices of 14 of the main cryptocurrencies from 1 January 2021 to 6 May 2021. This provides around 180,000 minutes of data. The 14 cryptocurrencies have been chosen mainly by market capitalization and their prices have been obtained from Binance. They 14 considered assets are:

- Cardano (ADA)
- Algorand (ALGO)
- Avalanche (AVAX)
- Bitcoin (BTC)
- Polkadot (DOT)
- Ethereum (ETH)
- ChainLink (LINK)
- Litecoin (LTC)
- Terra (LUNA)
- Polygon (MATIC)
- Solana (SOL)
- Uniswap (UNI)
- Monero (XMR)
- Ripple (XRP)

No missing values were found on the dataset. Hence, there was no need for any preprocessing tasks. It is worth noting that the period comprised from January 2021 to May 2021 was of bull markets, without important or sustained price downfalls.

5.2 Trading framework

How the neural networks are trained along time and how the trading is executed are key aspects to understand the functioning of the deep learning models designed. The Python package `deepdow` [13] has been used to implement this framework as it helps to ease all its intricacies.

First of all, we see the financial time series as a 3D tensor with dimensions indicator, time and asset. To illustrate this, in our case we will be dealing with minute-by-minute (time dimension) log-returns (channel dimension) of multiple cryptocurrencies (asset dimension). Graphically, one can imagine something like:

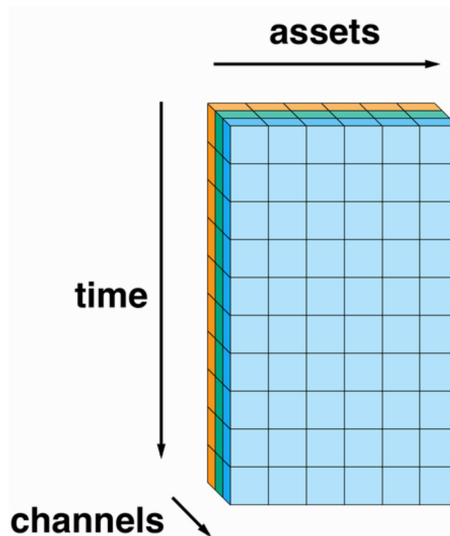

Figure 5.1: Financial time series seen as a 3-dimensional tensor. From [13].

If we fix a timestep (representing now), and we choose the length of the lookback, the horizon and a gap period, we can split our tensor into 3 disjoint subtensors X, g and y (as in Figure 5.2). X contains information about the past and present. The second tensor g represents information contained in the immediate future that we cannot use to make investment decisions (because, for example, we do not have enough time to train or to bring our algorithms into production). Finally, y is the future evolution of the market.

For illustrative purposes, let's consider a tensor of 12 timesteps, where lookback=5, gap=1 and horizon=4. We can move along the time dimension and apply this same decomposition at every time step on a rolling-window fashion. We would obtain 3 pairs of feature tensor and label tensor. This can be seen in Figure 5.3.

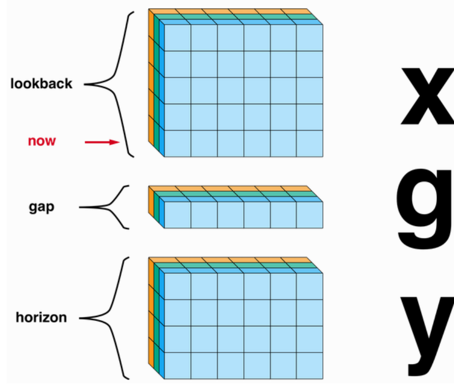

Figure 5.2: Splitting the time series into past, immediate present and future data tensors. From [13].

The networks we study receive as input the feature tensor X and return a weight allocation w. In other words, given the past knowledge X we construct a portfolio w that we buy right away and hold for horizon time steps. So, if f denotes some neural network with parameters θ, then $f(X, \theta) = w$. To train the parameters θ, we apply a gradient descent to a defined loss function. This loss function must receive as input the weights and the future information tensor and output a real number. We will denote it by $L(w, y)$.

In our experiments, we always use a lookback of 150 minutes, a gap of 1 minute and a horizon of 30. The loss function is the annualized Sharpe Ratio of the returns obtained in the horizon with the portfolio w output by the network.

As we want to simulate real-life trading, just splitting the original dataset of 180,000 minutes of data into one train and one test datasets is not the best idea as one would want to rebalance the portfolio often and it might not be necessary to look at past data that is so far in time. For this reason, we apply an external rolling-window (similar to the one illustrated in Figure 4.9) that tells how often we retrain the parameters θ and how many data samples we use to train them. The train splits contain 1050 minutes of data, which, considering the lookback, gap and horizon fixed, account for 869 pairs of samples containing a feature tensor and a label tensor. We apply the trained model to trade on the following 180 minutes, although we do not update the portfolio at every minute. Instead, we reoptimize it every half an hour. At time index 1051 we input the past lookback returns to the model and we use its output as the portfolio we hold for the next horizon timesteps (until the 1080th minute). Then again at time index 1081 we design a new portfolio based on the last lookback returns and we change our previous position to hold this one for the next half an hour. We repeat this 6 times in total (trading until the 1230th minute). After that, we shift the training set 180 minutes forward, training in the new train split and trading from time index 1231 to time index 1410. In this way we define a new window. We repeat this procedure 1000 times so that the whole investment period comprises 180,000 minutes of data.

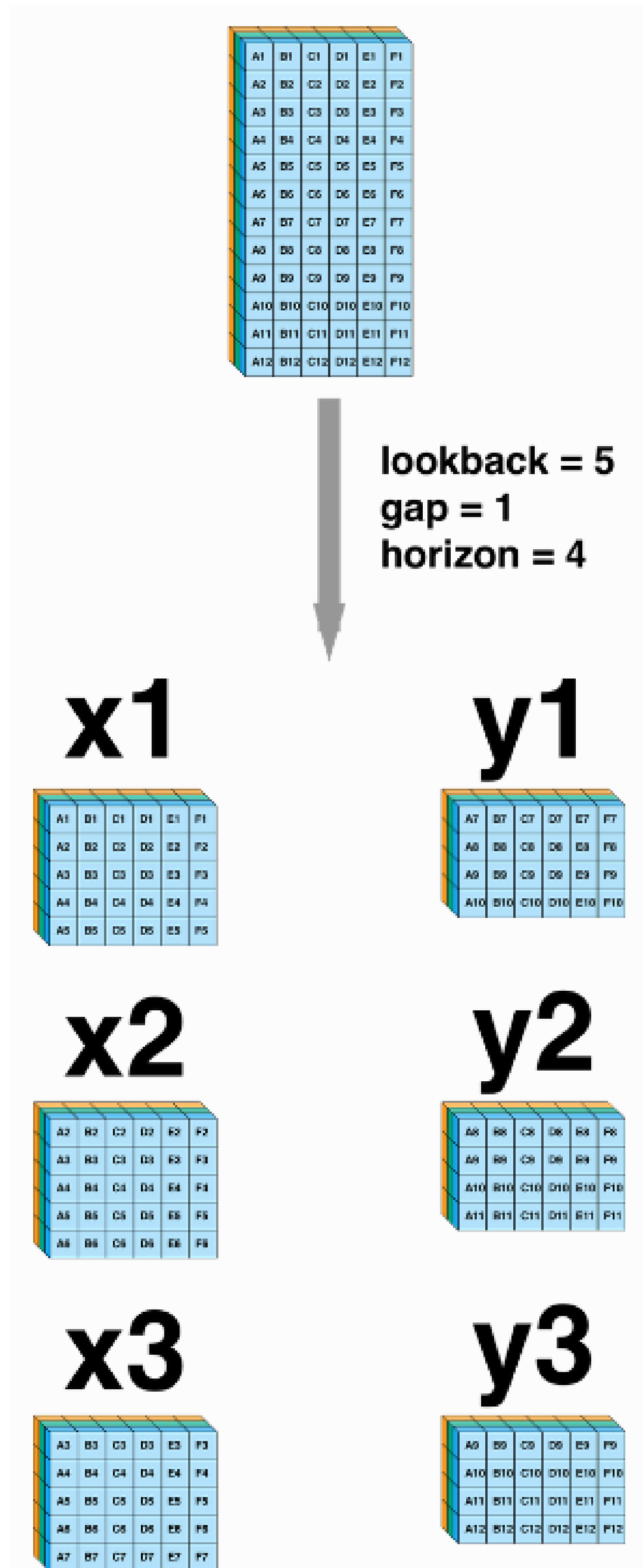

Figure 5.3: Generation of features and label tensors on a rolling-window basis. From [13].

At the first window, the network parameters are initialized via PyTorch's default initialization for each layer. But in all the next training periods we initialize the parameters with the optimal values found in the previous window. This helps to make the most out of the temporality of the data and to make the models more easily adaptable to changes in the market. However, if we look at every window as a separate backtest, they become less independent and can contribute to magnify problems such as overfitting or falling into local minima. Some preliminary experiments indicate though that this method of training the parameters is preferred to completely resetting the network at each window.

Finally, to evaluate the models we look both at the short-term and the long-term performance. For the short term, we consider each window as an individual backtest and we evaluate the measures described in Section 2.3 on each trading period consisting of 180 minutes of data. We can then boxplot these 1000 values for each model to facilitate the comparison among them. If we want to keep the temporality and get a 1-to-1 comparison in time we can plot them in a rolling fashion, so that we fix a lookback and at each timestamp we evaluate the measures on the past lookback portfolio returns and we plot the obtained values on the time axis. For the long-term, we evaluate the performance measures on the whole 180,000 minutes of trading obtained after concatenating all windows. In this case we only get one value for each measure and model so we can simply display them on a table.

Since the goal of these experiments is to analyze if deep learning can outperform the classical portfolios, their performances must be compared to the ones obtained with these portfolios that we regard as benchmarks. Specifically, 3 benchmarks are used: the Equally Weighted Portfolio, a MVP with the expected return vector μ calculated as the sample mean of the past lookback returns (leaving the same 1-minute gap in between) and with Σ being the sample covariance matrix, and a MVP where μ is calculated as an exponentially weighted average of the past lookback returns, with factor of $\frac{2}{l+1}$. Both MVPs have a risk-aversion parameter of $\lambda = 6$ and include a long-only constraint.

5.3 Proposed architectures

5.3.1 Two-block architecture

Modern portfolio theory divides the portfolio design procedure in two steps. The first one intends to forecast the behaviour of the considered financial products and the second optimizes the allocation given this forecast and some constraints. The reason why many of the classical portfolios do not work completely well is because of the errors in the forecast due to the noise present on the original data. This noise affects Σ only slightly (which can be calculated as the sample covariance matrix without too much error), but has a very important impact on the μ vector of expected returns.

Since the main issues of classical portfolios come from the forecasting step and not from the convex optimization problem, our goal is to analyze if applying a neural network to predict μ can provide better estimations than standard methods such as sample means or exponentially weighted averages. For this reason, the models built in this section consist of two clearly distinct blocks that perform separate tasks: estimating the expected return vector μ and calculating a MVP using this μ and the sample covariance matrix. This is illustrated in Figure 5.4.

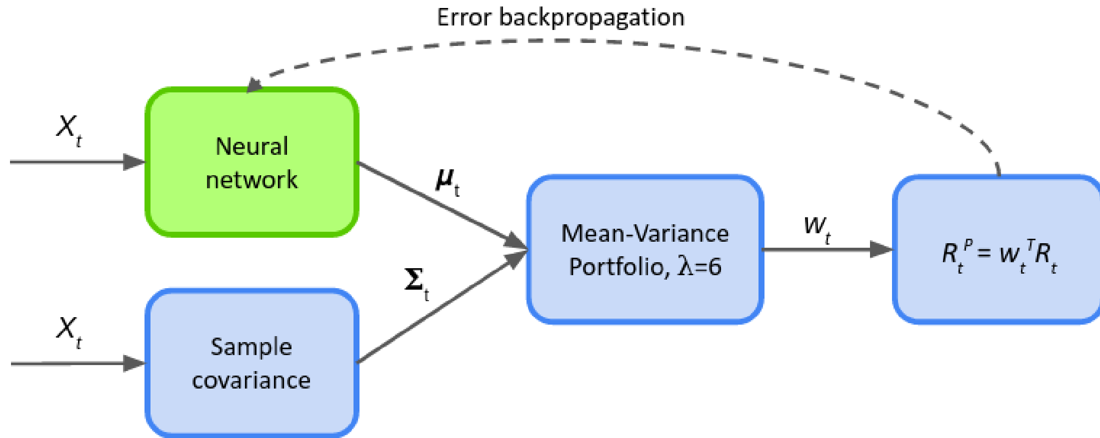

Figure 5.4: Pipeline of the models under a two-block architecture

The first block receives as input the past lookback returns of all the assets and outputs the μ vector via a neural networks composed of MLP and/or LSTM units. The second block solves the convex optimization problem of the mean-variance portfolio with a long-only constraint and fixing the risk-aversion parameter to $\lambda = 6$:

$$\begin{aligned} \max_{w} \quad & w^T \mu - 6 w^T \Sigma w \\ \text{s.t.} \quad & \mathbf{1}^T w = \mathbf{1} \\ & w \geq 0 \end{aligned}$$

The μ vector used in the problem is the output from the first block and Σ is the sample covariance matrix without shrinkage or any additional transformations. To update the parameters of the neural network in the first block we backpropagate the error from the output of the second across the optimization problem with the help of the Python package `cvxpylayers`. Understanding the exact way the gradient is propagated through the differentiable convex optimization layer falls out of the scope of this project. More details can be found in [1].

It is worth mentioning that even though we set a high value for the risk-aversion parameter of the mean-variance portfolio, the model is able to absorb it by scaling the μ vector conveniently. As we do not use any activation function on the output of the first block, its output can be scaled by simply increasing or decreasing the parameters of the network proportionally. The main reason to set this hyperparameter in this way is because it is the same value we use for the mean-variance portfolio benchmarks, so we can do more meaningful comparisons.

5.3.2 End-to-end

Current research on deep learning for portfolio optimization suggests that the forecast and optimization steps can be skipped in the models. The idea is that neural networks that from the returns directly output the portfolio allocation w in one step may provide better performances as they avoid the forecasting of the expected returns and covariances which are the root cause of the poor performance of classical portfolio designs.

In this section we follow the approach described in [22]. The core idea is to first get a fitness score for each asset via a neural network, and afterwards apply some differential functions to this scores vector to transform them into portfolio weights satisfying some specific constraints. Mathematically, we denote by N the number of assets, l the lookback, $\mathcal{R}_t \in \mathbb{R}^{l \times N}$ the input of the neural network which contains the current information of the market (log-returns in our case), g_1 a neural network with trainable parameters θ, $g_2 : \mathbb{R}^N \to \mathbb{R}^N$ a differentiable function and $s_t = (s_{1,t}, \dots, s_{N,t}) \in \mathbb{R}^N$ the vector of scores. Then the pipeline of these models simply becomes $s_t = g_1(\mathcal{R}_t; \theta)$, $w_t = g_2(s_t)$. As we backpropagate the gradient from the Sharpe Ratio obtained in the next horizon returns with this portfolio, it is necessary for h_2 to be differentiable.

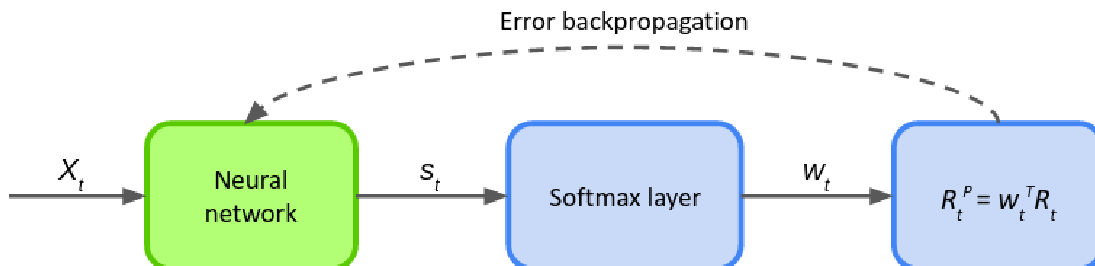

Figure 5.5: Pipeline of the end-to-end models

One could look at this architecture as something similar to the two-block architecture described in Section 5.3.1, where the second block that solves the mean-variance optimization problem has been replaced by some differentiable function g_2 (so the covariance matrix Σ is not needed). Now the question becomes what function should g_2 be. In [22] 4 different functions are proposed in order to satisfy different constraints: long-only, maximum position, leverage and cardinality. To make these backtests coherent and comparable to the ones with the two-block architecture, we only consider the long-only constraint. Therefore, h_2 becomes a softmax activation of the scores:

$$w_{i,t} = g_{2,i}(s_t) = \frac{e^{s_{i,t}}}{\sum_{j=1}^{N} e^{s_{j,t}}}$$

5.4 Neural networks

5.4.1 Proposed feed-forward networks

4 different feed-forward neural networks have been implemented and tested. Not all of them can be considered deep learning models as they do not even have a hidden layer and simply perform a linear transformation to the input. They all take as input the tensor containing the returns of all assets in the past lookback timesteps and output a vector of size equal to the number of assets. If applied into the two-block architecture, this vector is the μ used in the MVP block. In the end-to-end models, this vector represents the unnormalized weights of the portfolio (or scores), before applying the softmax function to them. To avoid confusion in the notation, the output of this neural networks will be denoted as a vector $z_t \in \mathbb{R}^N$, regardless of what their final purpose is going to be. The details of the 4 networks implemented can be found below:

1. **1 single independent linear layer per asset**: The first network implemented calculates each component of the output vector as a linear combination of the past lookback returns of that corresponding asset. The way the returns are combined to generate the output is different for each of them. As the network works independently for each asset, it is not capable of learning any interactions between them. If $z_{i,t}$ denotes the i-th component of the output of this network, and $\theta_{i,j}$ its parameters, we have that the i-th component of the output is computed as

$$z_{i,t} = \theta_{i,0} + \sum_{j=1}^{l} \theta_{i,j} r_{i,t-j}$$

From the equation above it is immediate to see that the number of trainable parameters of this network is $N(l+1) = 2,114$. This network is used as a baseline, meaning that the next ones are built by adding more complexity to this one, which is sketched in Figure 5.6.

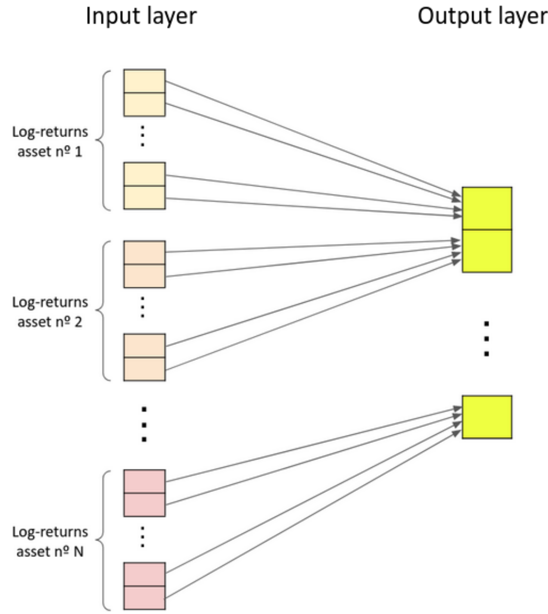

Figure 5.6: Baseline linear network implemented

2. **1 fully-connected linear layer**: The main downside of the previous two networks is that each component of the output vector is calculated independently by just using the returns of the corresponding asset. The most straightforward way to overcome this issue is by connecting all returns of all assets to each component of the output vector, so that, comparing to Figure 5.6, the connections between the input and the output become a dense layer.

   Noting by $\theta_{i,j,k}$ the weight that connects the return of asset k at time $t-j$ to the i-th component of the output vector, and by $\theta_{i,0i}$ its bias, the output can be calculated as:

$$z_{i,t} = \theta_{i,0} + \sum_{k=1}^{N} \sum_{j=1}^{l} \theta_{i,j,k} r_{k,t-j}$$

   So we now have only one linear layer whose input is the matrix of Nl returns and whose output is the whole vector of size N, without a non-linear activation function. One drawback of this network is the increase in the number of parameters, which now is $N(Nl+1) = 29,414$.

   This structure does generalize the previous one as its same behaviour could be replicated by setting to 0 all the weights of returns contributing to a component of the output vector that does not correspond to their asset.

3. **2 layers, first one independent for each asset**: In this network we take the structure of the baseline, we apply a rectified linear unit to the output of that layer and we add an extra fully-connected linear layer after that takes this vector of size equal to the number of asset and returns another same-sized vector. So we end up adding a hidden layer (see Figure 5.7).

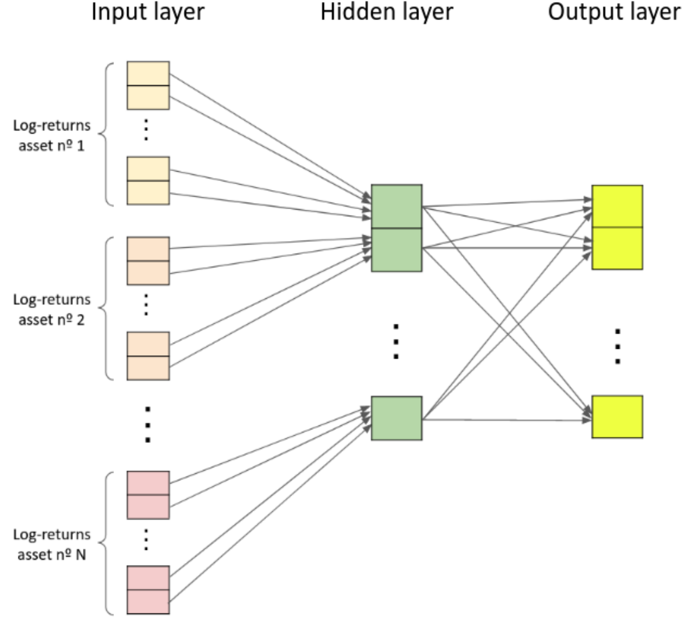

Figure 5.7: Feed-forward network with one hidden layer

Denoting by $\theta^2_{i,j}$ the parameters of this new layer and by $\theta^1_{i,j}$ the parameters of the previous one, the final output of this network can be expressed as:

$$z_{i,t} = \theta^2_{i,0} + \sum_{k=1}^{N} \theta^2_{i,k} \max \left\{ 0, \theta^1_{k,0} + \sum_{j=1}^{l} \theta^1_{k,j} r_{k,t-j} \right\}$$

This additional layer at the end allows the model to learn some correlations between the assets while not adding as much complexity as the network with only one fully-connected layer. The number of trainable parameters is now $N(l+1) + N(N+1) = 2324$.

It is worth remarking that, even though we built this network by adding complexity to network number 1, it can not directly replicate its performance due to the non-linear activation function added in between of the two layers.

4. **2 layers, first one shared for all assets**: The last feed-forward network implemented is the least complex one (in terms of number of trainable parameters) that can account for the interactions among assets. Its structure is similar to the one in Figure 5.7, with the difference that now the weights of the first layer are the same for each group of log-returns. So the trainable parameters $\theta^1_{i,j}$ are shared among all assets: $\theta^1_{0,j} = \theta^1_{1,j} = \cdots = \theta^1_{N,j}$ and we denote them all by θ^1_j. Its output is:

$$z_{i,t} = \theta^2_{i,0} + \sum_{k=1}^{N} \theta^2_{i,k} \max \left\{ 0, \theta^1_{0} + \sum_{j=1}^{l} \theta^1_{j} r_{k,t-j} \right\}$$

The number of trainable parameters of this network is $l + 1 + N(N+1) = 361$.

5.4.2 LSTM

5.4.2.1 Definition

The second type of neural networks implemented are LSTMs. Before describing the networks of this type implemented, we define briefly how they work and why they are more suitable to temporal data than standard feed-forward networks. The guide below follows Olah's notation on [20].

Oppositely to traditional feed-forward networks, recurrent neural networks have loops in them which allow information to persist. The network looks at some input x_t and outputs a value h_t. The loop feeds this output value to the network again so that it can be used to determine the next output given the next input. Although the loop might seem to complicate the network, a RNN can be understood as multiple copies of the same feed-forward network, each of them passing information to the successor. Figure 5.8 shows an unrolled recurrent neural network.

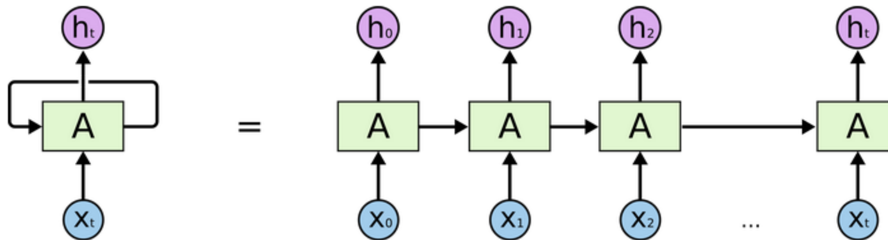

Figure 5.8: An unrolled recurrent neural network. From [20].

The chain-like structure of RNNs makes them very suitable for sequential data and time series, specially for problems where it is known that the output at a given timestamp depends somehow on the previous states of the time series. Some of these problems include speech recognition, translation, image captioning and language modeling.

Theoretically, recurrent neural networks should be capable of dealing with dependencies that go really far back in time. In practice, they face one important problem: the vanishing gradient. A RNN is also trained via gradient descent, meaning that all parameters that contributed to the final output should have its weight updated using backpropagation through time. Looking at the unrolled representation in Figure 5.8 it is easy to see that it is not only the neurons right below the output that we have to update, but also all the neurons far back in time.

When applying the chain rule to do the backpropagation, we would be multiplying the gradient consecutively by the weights that transmit information from the network at one timestep to the next one, which are initialized randomly and close to 0. Thus, the components of the gradient that go furthest in time would be very small in absolute value and the corresponding weights would never be modified substantially, making it difficult for the network to learn dependencies between events that happen very far in time.

Here is where Long Short Term Memory networks (LSTM) come into play. They are a type of recurrent neural networks (so they can still be unrolled) where the repeating module contains four single neural network layers (instead of one as the standard RNNs), interacting in a very specific way. The core idea behind the LSTMs is the cell state, represented by the horizontal line running through the top of the diagram in Figure 5.9. They run down the entire chain structure, only with some minor linear changes. The cell state allows information to flow along time and the changes on it are carefully regulated via structures called gates.

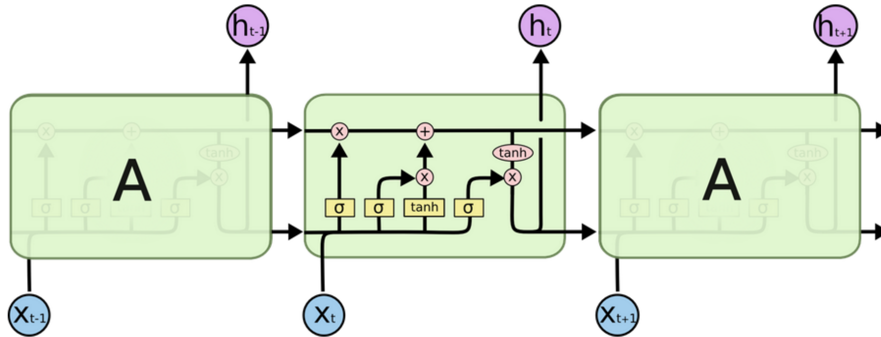

Figure 5.9: Insight on the LSTM repeated module. From [20].

Before going through the LSTM architecture step-by-step, it is necessary to introduce the notation we are going to use. d_1 and d_2 denote the dimensionality of the inputs and outputs, respectively. For the i-th module, $x_i \in \mathbb{R}^{d_1}$ denotes its input, $h_i \in \mathbb{R}^{d_2}$ its output and $C_i \in \mathbb{R}^{d_2}$ its cell state. θ denotes the set of weights and b the set of biases. By $[\cdot, \cdot]$ we denote the concatenation of two vectors, by $\odot$ the element-wise Hadamard product and the sigmoid function σ is $\sigma(x) = \frac{1}{1+e^{-x}}$, outputting values in the range $(0, 1)$. Let's describe now what is done in each layer:

1. The first step in the LSTM is to decide what information we are keeping and what information we are discarding from the cell state that comes from the previous module. This is done in the forget gate layer, which corresponds to the left-most sigma box in Figure 5.9. It is calculated as $f_t = \sigma(\theta_f \cdot [h_{t-1}, x_t] + b_f)$. For each component of f_t, a value close to 0 means that we want the corresponding component in the cell state C_{t-1} to forget its value while a number close to 1 means that we want it to keep the previous information.

2. The next step is to decide what new information is going to be stored in the cell state. This is done in two parts. On one hand, the input gate layer determines which values of the cell state we want to update: $i_t = \sigma(\theta_i \cdot [h_{t-1}, x_t] + b_i)$. On the other hand, a tanh layer creates a vector of new candidate values to be added to the cell state: $\tilde{C}_t = \tanh(\theta_C \cdot [h_{t-1}, x_t] + b_C)$. This operations happen in the second and third yellow boxes in Figure 5.9.

3. Thirdly, we combine the previous layers to update the old cell state C_{t-1} into the new one C_t. This is done as $C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$. These element-wise operations take place in the three left-most pink circles.

4. At this point the network is ready to decide its output, according to the cell state and the current input. We decide which parts of the cell state we want to output: $o_t = \sigma(\theta_o \cdot [h_{t-1}, x_t] + b_o)$, and we ensure the final output is in the range (-1, 1) by applying a tanh function: $h_t = o_t \odot \tanh C_t$. This output, calculated on the two right-most pink boxes in Figure 5.9, is copied also to the next module.

From the calculations above it can be immediately deduced that the number of parameters of a whole LSTM unit is $4\left((d_1 + d_2)\, d_2 + d_2\right)$. It grows linearly on d_1 and quadratically on d_2, so it is important to be very careful on increasing the hidden size as it makes the model considerably more complex.

5.4.2.2 Proposed networks

4 networks containing LSTM units have been implemented and tested in this project. Some of them are analogous to some feed-forward networks described in Section 5.4.1, where at least one of the linear layers has been replaced by an LSTM unit. The details of the 4 networks are described below:

1. **1 independent LSTM unit per asset**: In the first network there is one separate LSTM unit for each asset that takes as input the last lookback returns of the corresponding asset and calculates the respective component of the output vector. Thus the model consists of N LSTMs, each of them with input size l and hidden (output) size 1. For every LSTM, its output is multiplied by a real number (again different for each asset), which is also a trainable parameter of the model. Since the output of an LSTM is in the range (-1, 1) due to the last tanh function, this factor is needed so that the model can learn to differentiate the good assets from the excellent ones, and the bad from the poor, as they all might have values very close to 1 or -1. Also, in the two block-architecture this factor lets the model deal with the risk-aversion parameter of the MVP portfolio as it can conveniently scale the μ to make it more or less risky.

   This structure is similar to the network number 1 described in Section 5.4.1 as each component of the output vector is calculated independently and it can not detect any correlations among assets. The main difference is that the linear combination performed in the only layer there is here replaced by the output of an LSTM unit multiplied by a scaling factor. However, the LSTM adds a lot of parameters, reaching a total of $N(4((l+1)1+1)+1) = 8,528$. Figure 5.10 illustrates how the inputs and output on this network are connected.

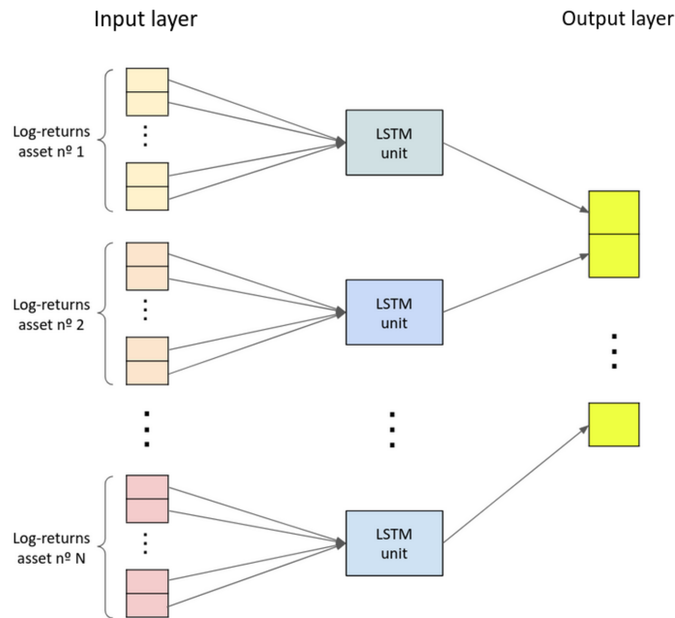

Figure 5.10: First LSTM network implemented

2. **1 fully-connected LSTM unit**: This second LSTM is analogous to the fully-connected linear layer already described, in terms of how it is able to learn the interactions. Basically we substitute the linear layer there for an LSTM of input size Nl (the whole matrix of returns of all the assets flattened) and hidden size N. For the same reasons explained in the previous network, the output vector is multiplied element-wise by another vector of scaling factors. Although this model can fully exploit the interactions between returns of different assets, it gets more complex as the number of trainable parameters of an LSTM increases quadratically with the hidden size. Precisely, it has $4((Nl + N)N + N) + N = 118,456$ parameters.

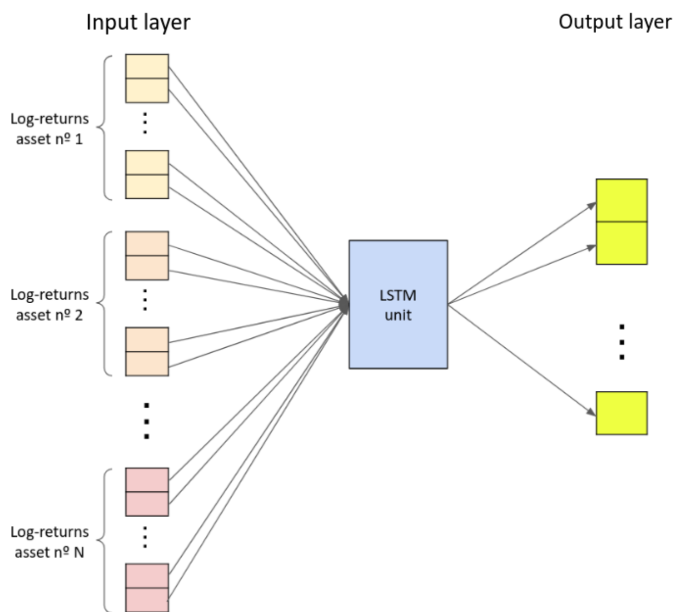

Figure 5.11: Fully-connected LSTM network implemented

3. **1 independent LSTM unit per asset + 1 fully-connected linear layer**: The third network applies the same idea described in the third feed-forward network described to deal with the interactions among the assets. What we do here is taking the first LSTM described here, without the last scaling factor, and adding a fully-connected linear layer at the end which receives this vector of N components and output another one of the same size. There is no need to add the scaling factor now as the linear layer has no activation function, so by simply scaling their weights the model can learn to do the same. It has a total of $N \cdot 4((l+1)1+1) + N(N+1) = 8722$ trainable parameters.

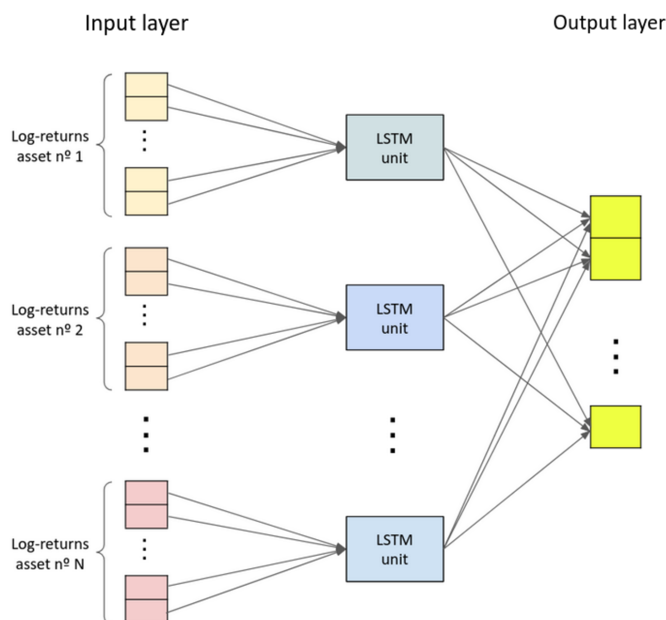

Figure 5.12: LSTM network with one additional dense layer

This network generalizes the first LSTM as it can completely replicate its behaviour. To do that, it would need to set all weights and biases of the additional linear layer to 0 except for the one that comes from the same asset, which should be equal to the scaling factor.

4. **1 independent LSTM unit per asset with 8 features**: The last model implemented is also independent on each asset but is significantly more complex than the first one. It still has one LSTM unit per asset whose input are the returns of that asset, but in this case the hidden size is increased up to 8. After applying these LSTMs to the returns, we get a set of N vectors of size 8. To transform these vectors into the output, we apply an independent linear layer to each of them. In this way, the learnt linear combination of the 8 elements of one vector becomes the corresponding component of the output vector. As we increased the hidden size of the LSTMs we should expect a much more complex model. Specifically, it has $N \cdot 4((l+8)8+8) + N \cdot 9 = 71,358$ parameters.

This network could perform equally to the first one by setting all weights and biases of the 8-to-1 linear layers added to 0 except for 1, which is left as the scaling factor.

5.5 Results

In this section we show the results obtained with all the networks and architectures described above when trading with the framework described in Section 5.2 along the period comprised by the dataset. Since a total of 16 different models have been tested, to facilitate their reading we the results are shown grouped by architecture and type of neural network, and separating the analysis of their short and long-term performances.

5.5.1 Two-block architecture

This subsection studies the performance of the models whose architecture integrates the resolution of a mean-variance problem. To avoid overloading this section with redundant plots, we just look at the Sharpe Ratios of the models and we leave the rest of measures (which study separately returns and risk) for the evaluation on the long-term done in Section 5.5.3. Here, instead of plotting directly the annualized Sharpe Ratio of each model measured per window, we plot the excess Sharpe Ratio respect to the Equally Weighted Portfolio benchmark. This excess Sharpe Ratio is basically the difference between the annualized Sharpe Ratio given by the model and the annualized Sharpe Ratio given by the EWP on that same window. The model is beating the benchmarks whenever this excess is positive.

The reason why we plot the excess Sharpe Ratio instead of the annualized Sharpe Ratio is because as the backtests are not fully independent and are done one immediately after another, looking directly at the Sharpe Ratio at each window ignores the temporal component of the backtests and leads to results that are difficult to read. Let's give a toy example. Imagine we test two models A and B on 8 windows. Model A gives the following sequence of Sharpe Ratios on each window: {-1, 0, 1, 2, 3, 4, 3, 2} and B gives the sequence {-2, -1, 0, 1, 2, 3, 2, 1}. Clearly model A would be performing better than B as it consistently provides a Sharpe Ratio higher in one unit. However, if we forget about the temporal component and we compare both sets straight we see that 6 out of the 8 values are the same in both sets and only two of them are different and can not be paired with a Sharpe Ratio of the same value in the other set: the 4 and the second 3 of the first set and the -2 and the last 2 of the second. So the conclusion we would immediately extract from that is that models A and B perform equally almost always except for some outlier cases where model A beats model B, and that is completely false. If instead of looking at the values by itself we analyze the 1-to-1 differences of the Sharpe Ratios that happen in the same window, we would obtain a set of 8 ones and that is a true indicator that model A is beating model B. This is the justification for using the excess Sharpe Ratio against a benchmark instead of the Sharpe Ratios by itself.

After this introduction on why we use this measure, we first analyze the results obtained by the models containing two blocks where the first one is a feed-forward network in a short-term perspective. Figure 5.13 shows the excess Sharpe Ratios compared to the EWP for all four models described in Section 5.4.1.2 and the two MVP benchmarks. We do not include the EWP in these plots as obviously all the excesses would be 0.

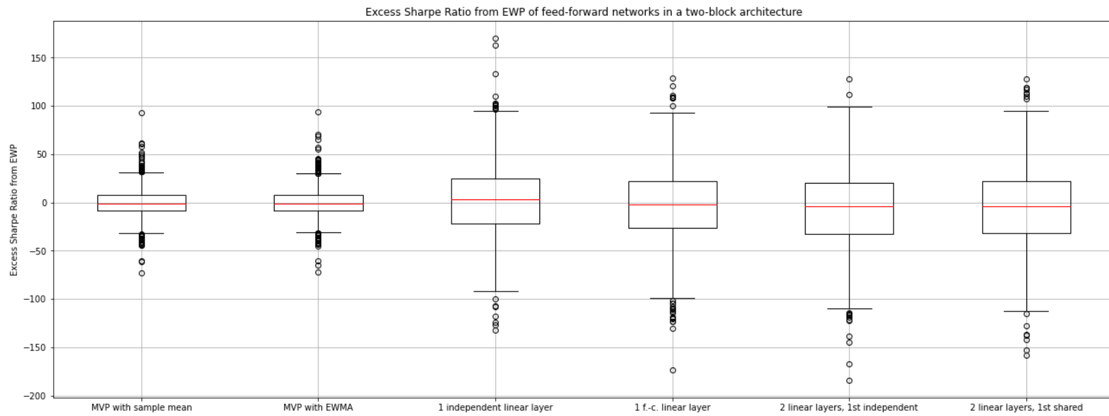

Figure 5.13: Short-term performance of the feed-forward models on a 2-block architecture

Although the great amount of outliers widens the scale of the boxplot and makes the differences less obvious, it can still be observed that the most simple of all models, the one with 1 independent linear layer per asset, is the only providing a median of the excess Sharpe Ratios above 0. The other models, which can learn the correlations among the assets, perform worse in this short-term basis, even worse than the benchmarks. This last comparison is a bit complicated to make though as the MVP benchmarks give a much narrower range of excesses than feed-forward models meaning that they perform more similar to the EWP than the other models. It is also interesting to note that, in terms of outliers, the simplest model has more outliers on the positive side and less and less significant on the negative, so it also provides good results in that sense. Finally, the fact that a more complex model like the one with a dense linear layer, which is a generalization of the model with 1 independent linear layer per asset as it could learn to trade in the same way by setting its parameters conveniently, performs worse suggests that some problems might be happening in its training process such as overfitting. This is analyzed in more detail in Section 5.5.4.

As a side note, regarding the benchmarks the previous boxplots show that the MVP with the sample mean and the MVP with an exponentially weighted moving average perform almost identically. Since their only purpose is to be references to compare the deep learning models to, there is no reason to keep showing them both so we will remove the MVP with the exponentially weighted moving average from the next plots.

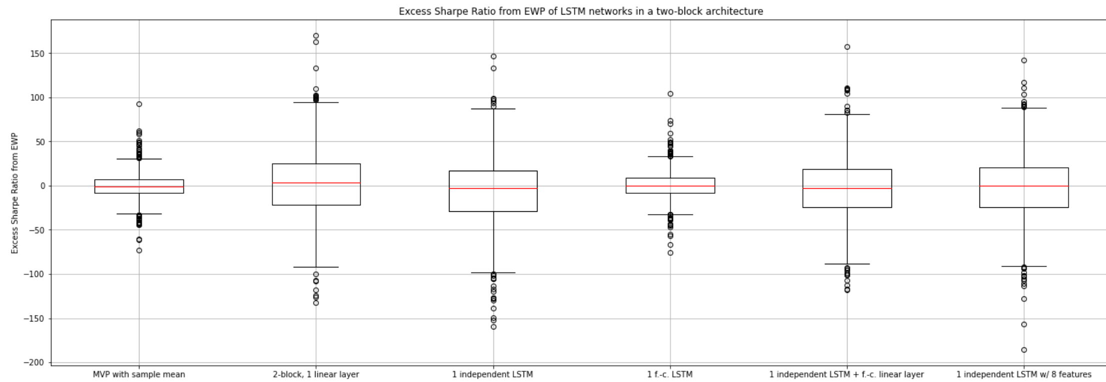

Figure 5.14: Short-term performance of the LSTM-based models on a 2-block architecture

Figure 5.14 repeats the same study we did in Figure 5.13 but for the models containing LSTM units in a 2-block architecture. We add to this boxplot the 1 independent linear layer model which provided the best results out of all the feed-forward models considered. In this case none of the models provides better results than the linear one. Although this poor performance of all 4 models was unexpected as the LSTMs are apparently more suitable for time series and problems of this kind there is one case worth exploring in more depth. The fact that the simplest of the LSTMs, which is basically a copy of the the best-performing feed-forward network in terms of how the network is connected only replacing the linear layers by LSTMs units that are more complex, can not perform better indicates that some problems are happening here on how the LSTM is being trained. Either the input and output do not have the best structure to fully exploit the temporality of the data, more epochs of training need to be done (although this crashes with the fact that the training time for one window should not exceed the 1 minute gap in order to be able to replicate this trading strategy in real life) or overfitting is happening. Again, more detail on this is carried out in Section 5.5.4.

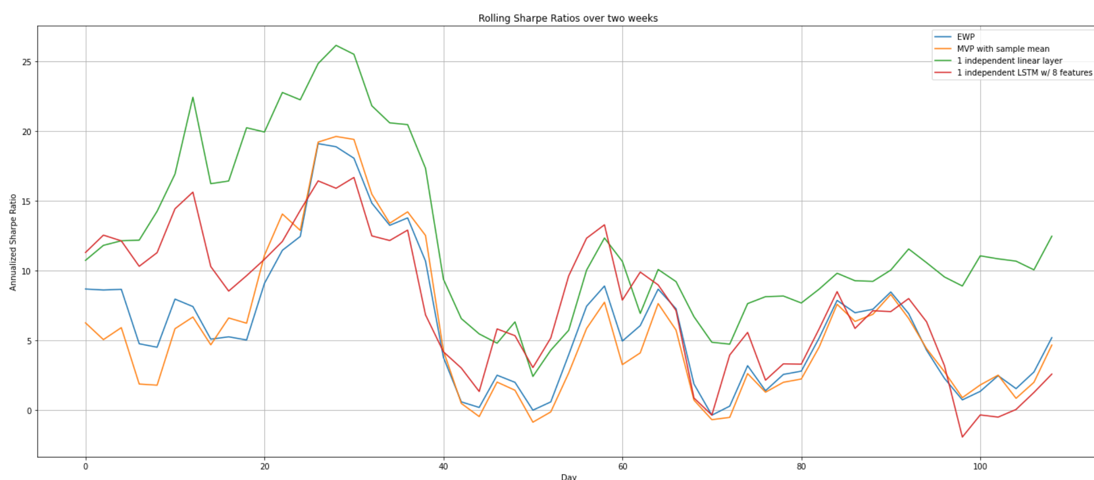

Figure 5.15: Rolling Sharpe Ratios on 2-week periods for the best-performing models on a 2-block architecture

To end this analysis, we calculate the Sharpe Ratios for the seemingly best-performing feed-forward and LSTM models in a rolling basis. Every 2 days (2880 timesteps) we evaluate the annualized Sharpe Ratio on the portfolio returns of the past two weeks (20,160 minutes). This way of plotting the measures exploits even more the temporal component of the backtests as a direct 1-to-1 comparison can be made at each timestamp considered and distinguish in which periods one model performs better than another and in what periods it does not. Figure 5.15 confirms that the simplest model is the best one so far as it is always considerably above the benchmarks, specially at the beginning and at the end of the investment period. On the other hand, it is not even clear if the LSTM model (also learning independently for each asset) performs better than the benchmarks. To extract a more accurate conclusion, it is necessary to look at their long-term performances in Section 5.5.3.

5.5.2 End-to-end models

After revising the performance of the models under a two block architecture, where the parameters of the neural network that predicts the expected return vector are updated taking into account its performance on a mean-variance portfolio, we repeat the same analysis on the end-to-end models, with the same neural networks described in Sections 5.4.1 and 5.4.2. We plot again the excess Sharpe Ratios for each model here, and we add the best model found so far to get the big picture of the performance of the end-to-end ones.

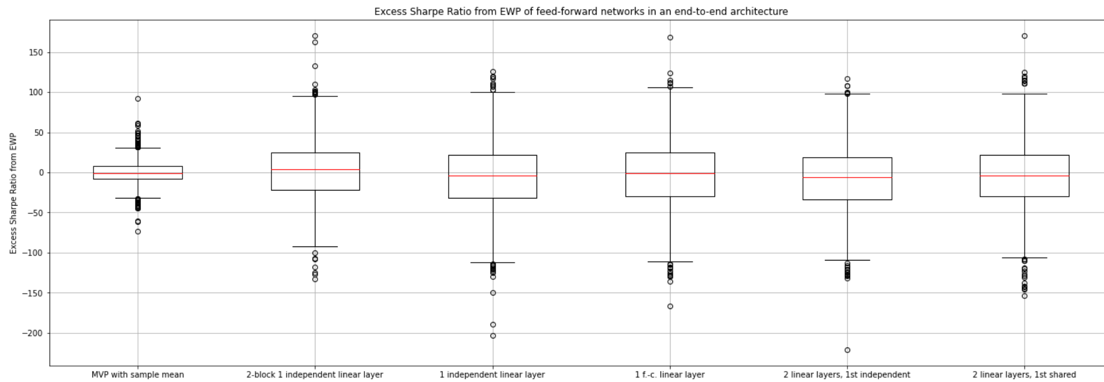

Figure 5.16: Short-term performance of the feed-forward models on an end-to-end architecture

Figure 5.16 indicates that the end-to-end feed-forward models are quite far from the best 2-block one. Moreover, none of them is clearly beating the benchmarks. If we compare them 1-to-1 with their equivalents on the 2-block architecture (Figure 5.13) it seems that end-to-end is generally a worse performing architecture. Also, the same conclusions can be extracted about the end-to-end LSTM models, from Figure 5.17. Their performance is in general poor and do not improve their corresponding 2-block models. Further analysis should be done to understand why the fully-connected LSTM performs so much more similar to the EWP than the rest of the models.

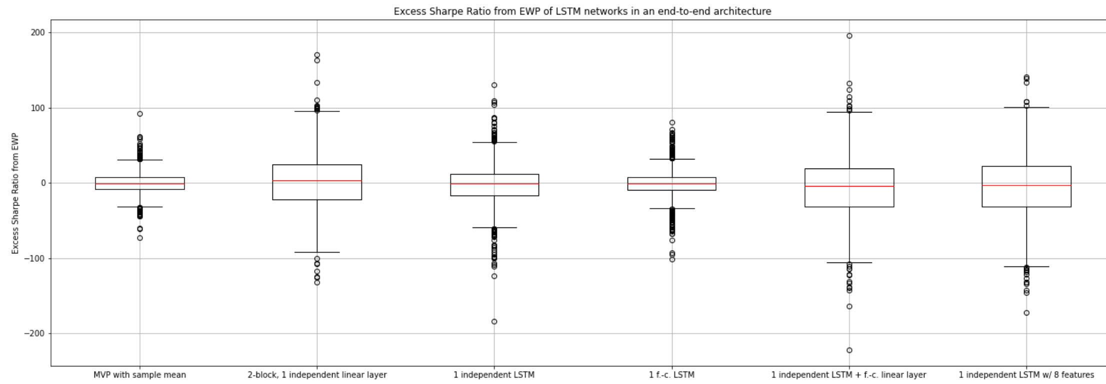

Figure 5.17: Short-term performance of the LSTM-based models on an end-to-end architecture

For the rolling Sharpe Ratios analysis, we include the feed-forward model with one fully-connected layer as it had the best median of the excesses and the LSTM which learns 8 features of each asset independently beacuse none of the LSTM stood out from the rest but at least this was the best one in the 2-block case so we can compare them directly.

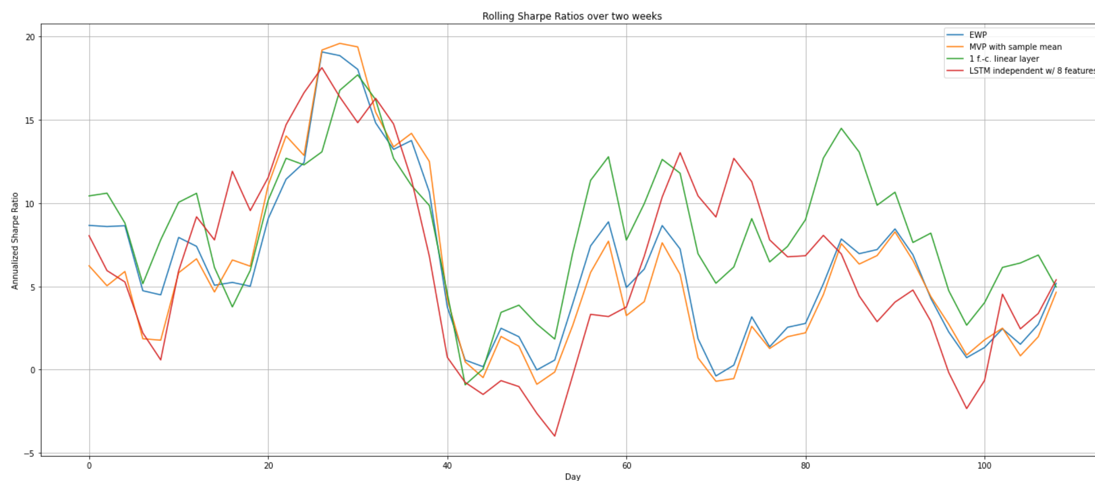

Figure 5.18: Rolling Sharpe Ratios on 2-week periods for the best-performing models on an end-to-end architecture

Here none of the models is significantly better than the benchmarks. The fully-connected linear one has better performance on the last half of the data and the LSTM does not provide clear differences.

5.5.3 Summary of long-term performances

Table 5.1 displays the usual performance measures (annualized non-cumulative return, annualized volatility, maximum drawdown and annualized Sharpe Ratio) for each model implemented evaluated on the whole 180,000 portfolio returns provided by each on the whole investment period. In general terms, it corroborates the arguments presented in Section 5.5.1 and 5.5.2.

	Return	**Volatility**	**Max DD**	**Sharpe**
Benchmarks				
EWP	7.482	1.230	0.204	6.087
MVP with sample mean	5.535	1.029	0.183	5.377
MVP with EWMA	5.541	1.030	0.181	5.378
2-block architecture				
1 independent linear layer	19.12	1.607	0.200	**11.89**
1 fully-connected linear layer	10.74	1.692	0.241	6.350
2 linear layers, 1st independent	9.429	1.859	0.202	5.073
2 linear layers, 1st shared	12.53	1.943	0.236	6.449
1 independent LSTM unit	10.34	1.629	0.191	6.345
1 fully-connected LSTM unit	6.207	1.035	0.183	5.996
1 independent LSTM unit + 1 f.-c. linear layer	9.000	1.355	0.244	6.643
1 independent LSTM unit w/ 8 features	11.48	1.505	0.236	**7.631**
End-to-end				
1 independent linear layer	14.39	2.296	0.294	6.271
1 fully-connected linear layer	14.53	1.873	0.252	**7.760**
2 linear layers, 1st independent	8.675	1.895	0.238	4.578
2 linear layers, 1st shared	12.68	1.830	0.235	6.930
1 independent LSTM unit	8.926	1.529	0.224	5.839
1 fully-connected LSTM unit	7.928	1.326	0.201	5.979
1 independent LSTM unit + 1 f.-c. linear layer	12.06	1.901	0.240	6.342
1 independent LSTM unit w/ 8 features	12.25	1.858	0.231	6.591

Table 5.1: Long-term performances of all models implemented (without transaction costs)

In first place, the models that seem to perform best on a short-term basis (of which we also plotted the rolling Sharpe Ratios) also perform best on the long-term. Specifically, the difference between the 2-block architecture with 1 independent linear layer per asset and the rest becomes now completely evident, with an annualized Sharpe Ratio of about twice the others. This difference comes mainly from a big increase in the returns but not in terms of risk. Nonetheless, it must be remarked that the huge value for the annualized return might be considerably impacted by some outliers that appeared in Figure 5.13, thus reducing the statistical significance of this value.

Secondly, it appears plain to see that it is worth adding the MVP block in the architecture of our models so that we take into account the covariance matrix also and we design the portfolio by solving an actual optimization problem (which we know that ideally should provide a desired portfolio) and not by simply letting a neural network decide. This comes obvious by seeing how the volatility of end-to-end models increases respect to their 2-block analogues without clear differences in terms of returns. The reason for this is that the MVP forces the model to minimize the variance up to some extent while end-to-end gives more freedom to allocate as much as possible to the assets with more promising returns without taking into account their correlations. As the returns are so difficult to predict, end-to-end models do not end up providing better returns but they do increase the risks.

Reminding that the Sharpe Ratio is the function being optimized for our networks, and that some of the models could replicate the benchmarks, one would expect them to perform at least equally to the reference portfolios. However, this not always the case. For example, the end-to-end model with 2 linear layer where the first one is independent for each asset could easily replicate the EWP by setting all the trainable parameters of the second layer (the dense one) to 0 so that the outputs of the softmax have all the same value. Overfitting may be the root cause of this. See Section 5.5.4 for a more detailed discussion on this.

The last relevant aspect in the table above is how the LSTMs do not perform better than the simplest linear models. The temporality of the data should make LSTM-type networks very convenient for this problem, however the results show the contrary. Again, a deeper analysis of this issue is carried out in the following section.

To finish this presentation of the results, we plot the cumulative wealth over time for the benchmarks and the best model for each combination of architecture and neural network type in Figure 5.19. It shows the dominance of the 2-block architecture with a simple linear layer per asset but also the return outliers can be observed, for example around timesteps 35,000 and 50,000.

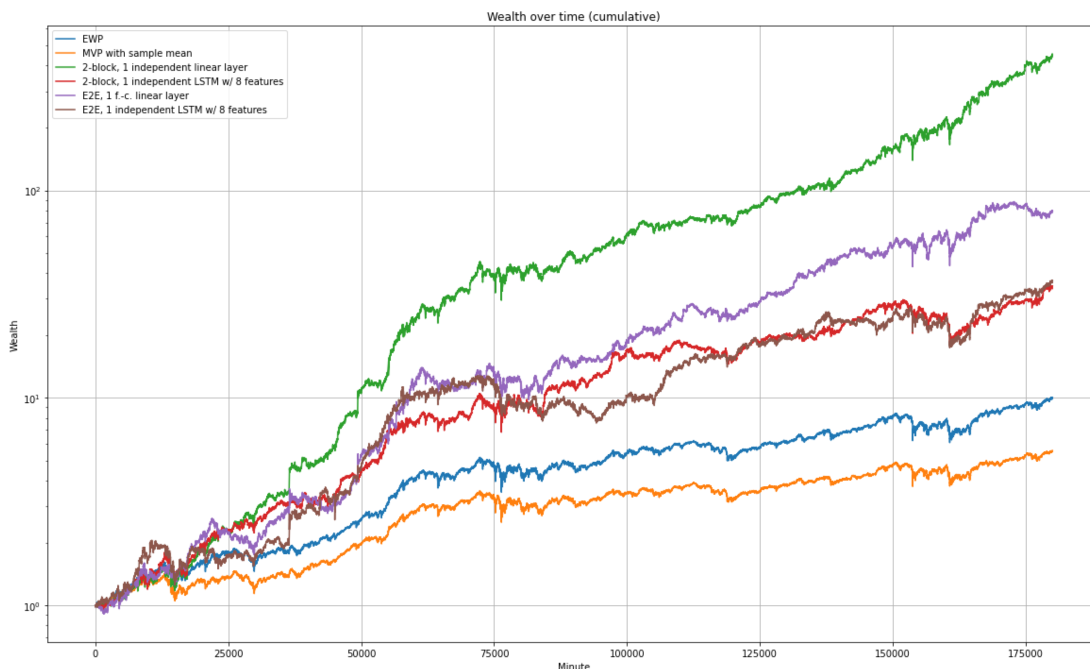

Figure 5.19: Cumulative wealth for the best-performing architectures

5.5.4 Analysis on the capacity of overfitting of the models

The results in Sections 5.5.2 and 5.5.3 suggested that the training of some models was not optimal since their performance differed from what we would have expected according to the complexity of the models and what features they should be able to extract from the data. To get a deeper insight on this matter, in this section we dig into how the models can learn by how we defined them and what may cause this poor performances.

Since the end-to-end models do not produce any relative differences among the models compared to what can be observed with the 2-block ones, here we focus only on the 2-block models, specifically on the feed-forward with 1 independent linear layer per asset, the feed-forward with 2 linear layers where the first is independent per asset and the second is dense, the analogous of the first feed-forward but with LSTMs and the best-performing LSTM that learns 8 features per asset. From Sections 5.4.1 and 5.4.2 we know that these models have 2114, 2324, 8528 and 71,358 trainable parameters, respectively.

As the models are trained with only 8 epochs, it is difficult to illustrate how the train losses evolve on these 8 epochs at each window in a meaningful way. Since we mainly want to see how one model overfits the data compared to another, it is enough to look if they yield better or worse in-sample performances and can learn more specific features of the training data that can not be extrapolated to the out-of-sample. Therefore, the boxplot in Figure 5.20 shows the training loss (annualized Sharpe Ratio) obtained by each of the aforementioned 4 models at the 8th epoch of the training at each window.

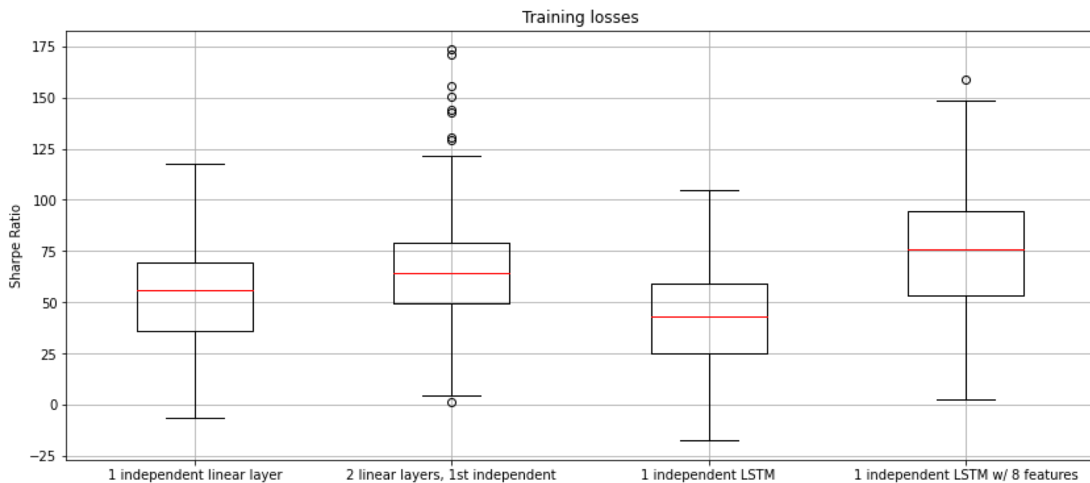

Figure 5.20: Sharpe Ratios on in-sample data at the end of the training for each window

We take as a reference the left-most column of the boxplot as it corresponds to the best-performing method out-of-sample. The first thing to observe is that the feed-forward network with 2 layers consistently ends up with higher Sharpe Ratios on the training data. However, it is not capable of translating this information to the test data. This is a clear symptom that overfitting is occurring

in this case. The number of parameters of both networks is not substantially different, so we deduce that the capacity of overffiting that the model has does not come strictly from the number of trainable of parameters of the model but also from how it is connected. There is no simpler (in terms of number of additional trainable parameters) and still sensible way of letting the model learn the interactions among assets than adding this extra dense layer at the end, so changing the structure of the model might not be the most appropriate approach to reduce this overfitting and we should deal with it in training time.

There exist various ways of reducing overfitting without changing the architecture of the model. One would be to add a validation set and early stopping. The validation set would be a split of the training data containing a small proportion at its end. We would use the remaining of the training data to train the parameters and at the end of each epoch we would evaluate the loss function on this validation set with the parameters calculated at that same epoch. If after a given number of windows the value of the loss function on the validation set has not improved, we would stop the training for that window and use the parameters that provided best results over the last epochs to trade on the test split. Another way to prevent overfitting is by adding dropout layers. However, some smaller untabulated experiments showed that dropout did not help much in our models. And even another approach to overcome overfitting would be to increase the size of the training datasets, so that the ratio of trainable parameters and number of samples decreases. This would imply changing the trading framework described in Section 5.2 so an integral review of the experiments should be carried out.

When looking at the loss values for the simplest LSTMs, we see that they are below the linear model. By how we set the LSTM, it should be able to at least perform equally to the linear model by just ignoring the memory and feedback and just caring about the input parameters. This does not happen as the low values of the Sharpe Ratios on in-sample data indicate that the model is not capable of learning as much. Thus overfitting is not the main issue here. The problem probably comes from a bad training procedure for this model. At each timestep the LSTM was input the whole lookback of returns. This may not be optimal as the LSTM memory should be able to understand how to interpret this lookback and keep it in his cell state if we just input one return at a time. To let the model learn more features, we would increase the hidden size instead and add a dense layer at the end. More research should be done in that sense to conclude which is the optimal way of setting up the LSTMs in this problem.

Another problem the LSTMs may be encountering here is an incomplete training, needing more epochs or a bigger learning rate to end up converging at the end of the training. However, increasing the number of epochs implies increasing the training time, and to avoid cheating in the backtests this time should not exceed the 1-minute gap imposed in our framework as otherwise these experiments could not be replicated in real life (with the experiments and the computational

resources we had available for them, with 8 epochs we are already close to reach this 1-minute threshold). But increasing the gap would most probably imply a decrease in performance as the model has to predict further in the future. So the pros and cons should be balanced carefully before deciding if it is worth letting the model train for more epochs or not.

Finally, from the last LSTM plotted, we can deduce that it is suffering both from overfitting (as its in-sample Sharpe Ratios are higher than the ones from the linear model but not when measuring them out-of-sample) and probably from bad training practices as the main issues commented about the simpler LSTM model must also be happening in this one.

Chapter 6

Experiments on data of different frequencies

The previous chapter showed which models performed best under the environment considered. Some important assumptions have been done when designing the framework but there is one specially worth analyzing in depth: the frequency of the data. Some of the models designed could clearly beat the benchmarks when working on minute-by-minute data if transaction costs were not taken into account. However, since in the experiments on Sections 4.2 and 4.3 the results changed completely when switching from daily to high-frequency data, our hypothesis is that as we lower the frequency of the data, the performance of the models worsens because the markets are more efficient. We test this hypothesis on the best-performing feed-forward model found in Section 5.5.

6.1 Dataset

4 datasets have been used in this section to carry out the planned experiments. Each dataset contains prices of the 14 cryptocurrencies mentioned in Section 5.1 measured with different frequencies. The first dataset contains minute-by-minute data from 5 July 2021 to 6 December 2021. The second has prices measured every 15 minutes from 24 June 2021 to 6 December 2021. The third contains hourly data from 23 May 2021 to 6 December 2021 and the last contains the prices gathered every 4 hours, from 12 January 2021 to 6 December 2021. The reason why each dataset comprises a period of different length is because after applying the trading framework described in Section 6.2, generating the maximum number of windows possible and creating the feature and label tensors, in all cases we obtain the same trading period, which goes from 6 July 2021 to 6 December 2021. Not all cases contain the same amount of data in all this period though.

6.2 Trading framework

For each frequency, the trading framework considered is analogous to the one described in Section 5.2. We set the lookback, gap and horizon to 150, 1 and 30 timesteps respectively. That means that in one case the lookback is 150 minutes, in the next one is 2250 minutes, in the third is 150 hours and in the last is 25 days. The train and test splits are generated with the same proportion of data: 1050 timesteps are used to train the model and the trading with the learnt parameters is performed in the following 180 timesteps. Since the amount of data is different in each case, the number of windows that can be rolled also changes. Specifically, in the 4h data we create 5 windows, with the hourly data we can create 20, in the 15-minute dataset we can roll 80 windows and with the minute-by-minute data we obtain 1200. This means that the whole investing period lasts 216,000 minutes and the portfolios are reoptimized 30, 120, 480 and 7200 times in each case, respectively.

In all cases, the feed-forward model with a single independent linear layer per asset on a two-block architecture is trained, always under the same hyperparameters. We use batch size of 64 samples, a learning rate of 0.01, a risk-aversion parameter for the MVP of 6 and 8 epochs per window. The trainable parameters of the network are reused by using the optimal ones of one window to initialize them at the next one.

The way these investment models are evaluated changes slightly from what is described in Section 5.2. The point of trading in the same temporal period for all 4 frequencies considered is to have the same environment and market conditions in all cases so we can carry out a 1-to-1 comparison.

As the amount of timesteps and the number of windows on which the models have been trained are the different in each case, it makes no sense to compare the short-term performances of them. For this reason we only evaluate the models on a long-term basis (over the whole 5 months investment period) with the usual measures. Firstly we have to compare the performance of the model on each frequency with the benchmarks (EWP and MVP with the sample mean as the expected return vector) applied on that same frequency. Secondly, we also have to compare the results of the model working in different frequencies among them to understand if higher frequencies are capable of exploiting the fast changes of the market or not.

	Return	**Wealth**	**Volatility**	**Max DD**	**Sharpe**
Frequency of 1 min					
EWP	2.727	2.705	0.782	0.302	3.486
MVP with sample mean	1.930	2.021	0.660	0.272	2.926
2-block, 1 independent linear layer	6.873	**12.76**	1.164	0.310	**5.905**
Frequency of 15 min					
EWP	2.751	**2.595**	0.926	0.303	**2.968**
MVP with sample mean	2.135	2.098	0.813	0.278	2.627
2-block, 1 independent linear layer	2.654	2.400	1.021	0.288	2.598
Frequency of 1 h					
EWP	2.681	2.542	0.905	0.302	2.963
MVP with sample mean	2.140	2.101	0.851	0.276	2.625
2-block, 1 independent linear layer	3.104	**2.943**	0.979	0.268	**3.170**
Frequency of 4 h					
EWP	2.554	2.467	0.842	0.302	3.032
MVP with sample mean	2.103	2.098	0.772	0.281	2.723
2-block, 1 independent linear layer	3.044	**2.989**	0.871	0.280	**3.495**

Table 6.1: Long-term performances with different frequencies

6.3 Results

Table 6.1 shows the standard performance measures (annualized cumulative return, annualized volatility, maximum drawdown and annualized Sharpe Ratio) for each of the frequencies tested. We also add the final wealth obtained in each case after the investment period, assuming an initial budget of 1$ and full reinvestment at each timestep. This allow us to see the difference of investing in a cumulative or a non-cumulative way and makes it easier to compare the performances of the neural network model when working on data of different frequencies

The first and most important interpretation that can be made from this table is that the neural network model is only capable of clearly beating the benchmarks on minute-by-minute data as there is a huge difference in return with only a small increase in volatility. In the other cases there are very thin differences, and when dealing with 15-minute data the neural network performs even worse than both the Equally Weighted Portfolio and the MVP with the sample mean.

If we want to compare the performance of the neural network model depending on the frequency of the data, the conclusions are quite similar. In minute-by-minute data the final wealth after the 5 month investing is around 4 times greater than in the other cases, without any significant differences in terms of risk. This proves our hypothesis that the markets become more efficient as we look at lower-frequency data and less predictability can be found by the neural networks. For illustrative purposes and to exemplify the differences of performances in the long term due to the exponential behaviour of the cumulative return, we plot in Figure 6.1 the evolution of wealth over time for neural network model depending on the frequency of the data they have been trained with.

Figure 6.1: Cumulative wealth when using data of different frequencies

As a side note, it is true that one could argue that the results obtained in this chapter might be biased because the architecture of the model chosen for these experiments was one that we knew that already provided good results in minute-by-minute data. Although more extensive research should be done by trying different architectures in every frequency, these results give a first idea of what happens on the predictability of the returns that the neural networks can find when we decrease the frequency.

Chapter 7

Incorporating transaction costs

The results in Section 5.5 showed very promising results for some of the models backtested as there were significant differences when compared to the benchmarks. Nevertheless those results do not reflect the truth of how these models would perform when applied to real-life trading. The reason is that when bringing them into into production and trading the cryptocurrencies considered, the exchanges would impose some transaction fees that penalize excessive trades. The goal of this chapter is to analyze the impact of transaction costs on the best-performing models found in Section 5.5. According to the results seen in Section 4.3 when repeating the experiments including transaction costs, and the literature overview developed in Chapter 3, our hypothesis is that transaction costs will make all the gains vanish and the models will not perform better than the benchmarks anymore.

7.1 Dataset

Since the objective of these experiments is to replicate the work done in Chapter 5 but including transaction costs when calculating the portfolio returns (as described in Section 2.2) and evaluating the models, the dataset used for this experiments has to be the same we used in those. As a reminder, it conatins the minute-by-minute close prices of 14 of the main cryptocurrencies by market capitalization (ADA, ALGO, AVAX, BTC, DOT, ETH, LINK, LTC, LUNA, MATIC, SOL, UNI, XMR and XRP) from 1 January 2021 to 6 May 2021, accounting for around 180,000 timesteps in a period of bull markets.

7.2 Trading framework

In the following experiments transaction costs are included in the model evaluation but not in the training procedure, meaning that the Sharpe Ratio used as the loss function of the network that we want to optimize does not take them into account. For this reason, no changes have to be made to the trading framework described in Section 5.3. The rolling-window approach to split the data into successive training and test sets remains exactly the same, rolled on 1000 windows with the same lookback, horizon and gap.

Only three models are tested. Compared to the results in Section 5.5 they are only going to worsen their performance so the ones that did not perform well enough in those experiments are not worth of deeper analysis here. These three models are: the best-performing overall (two-block architecture with one independent linear layer per asset), the best LSTM (two-block architecture with one independent LSTM unit per asset with 8 features) and the end-to-end model with a fully-connected linear layer as it was the best of all the end-to-end ones. The allocations used are the same as the ones tested in Section 5.5, so there is no need to retrain the models again.

In the computation of the returns (done as defined in Section 2.2), three different values for the transaction fee C have been considered. $C = 0$ represents the absence of transaction costs, which is basically what was done in Chapter 5. $C = 0.001$ is a standard fee that one would encounter often when trading in real life. For example, Binance imposes a fee of 0.1% per spot trade (the same as $C = 0.001$) to all newly registered and beginner traders. Finally, we also use $C = 0.0002$ as in [22] this was the highest value that still allowed them to obtain better performances than the benchmarks. However, this value is unrealistic and in any exchange we would encounter considerably higher fees.

The performance of the models is studied here by looking only at the long term, considering the whole 4 months as the investing period and evaluating the performance measures there. These results are again compared to some standard benchmarks (which now also include transaction costs). Since the mean-variance portfolios where the μ vector is calculated as the sample mean and as an exponentially weighted moving averaged provided very similar results, we discard the latter and only show the former in the plots, together with the Equally Weighted Portfolio.

7.3 Results

Figure 7.1 shows the evolution of cumulative wealth over time for the 3 aforementioned models and the 2 benchmarks with transaction fees of $C = 0.0002$ and $C = 0.001$.

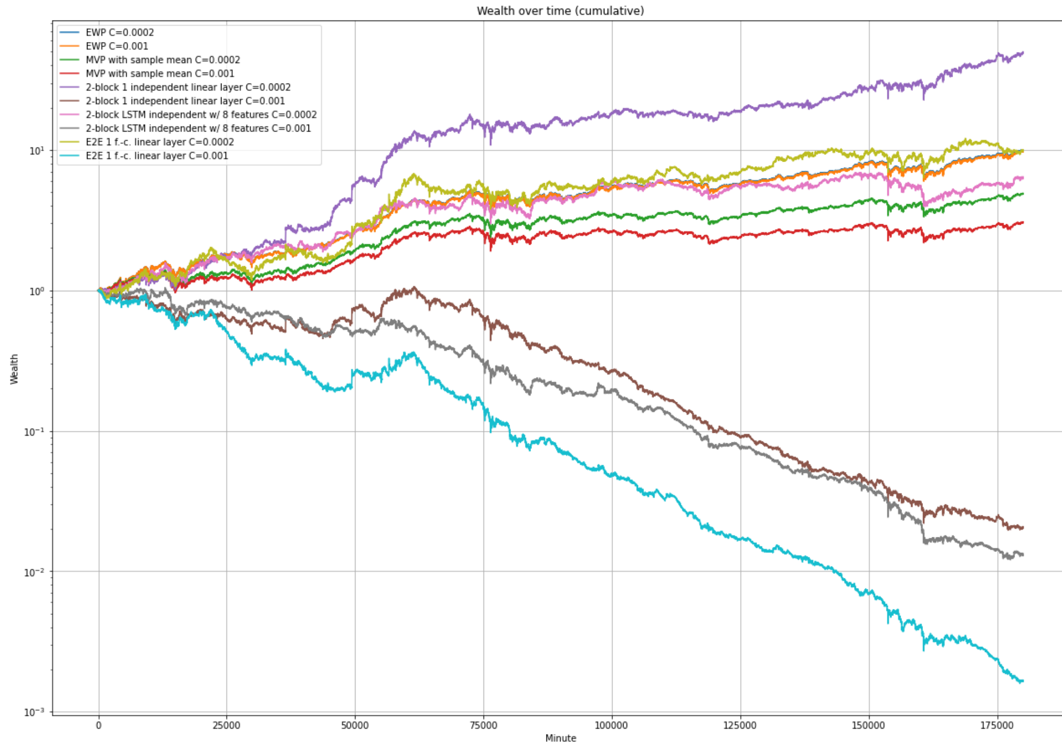

Figure 7.1: Cumulative wealth with different transaction fees

First thing that stands out from the plot above is that with a fee of 0.1% in all deep learning models the transaction costs become much bigger in absolute value than the returns so that we not only perform worse than the benchmarks, but also we end up quickly losing all the initial budget and after the 4 months only arround a 1% of it would be remaining. However, this does not happen with the benchmarks, specially with the EWP. To dig deeper into the reasons of this, it is necessary to understand what causes the transaction costs to be greater or smaller.

In our framework, transaction costs happen from two main reasons. Firstly, as we reoptimize our portfolio every half an hour with a new output of the neural network, the allocations might change completely from the timestep before the reoptimization to the one right after so, recalling the way of calculating the returns described in Section 2.2, this greatly impacts the return after the reallocation. The second cause of transaction costs is that at every minute the returns we get are not proportional to the portfolio positions, so if we want to reinvest this new returns while keeping the same portfolio as before we need to do small trades that distribute this new returns as desired. While the former transaction costs happen only every half an hour, the latter affect the portfolio at every minute.

The fact that the benchmarks (and specifically the EWP) are less affected from transaction fees indicates that what causes this huge drops in performance of the deep learning models are the reoptimizations we do every half an hour. This reoptimizations do not happen for the EWP as the allocations remain the same along the 180,000 minutes, and are less significant for the MVP because the only thing done to reoptimize it is calculate a new μ and solve the same convex optimization problem with this new expected return vector. In this sample mean, 120 out of the 150 returns used for its calculation were also present in the computation of μ for the previous portfolio, and they all have the same weight, so it is expected that the changes in final allocations will not be drastic.

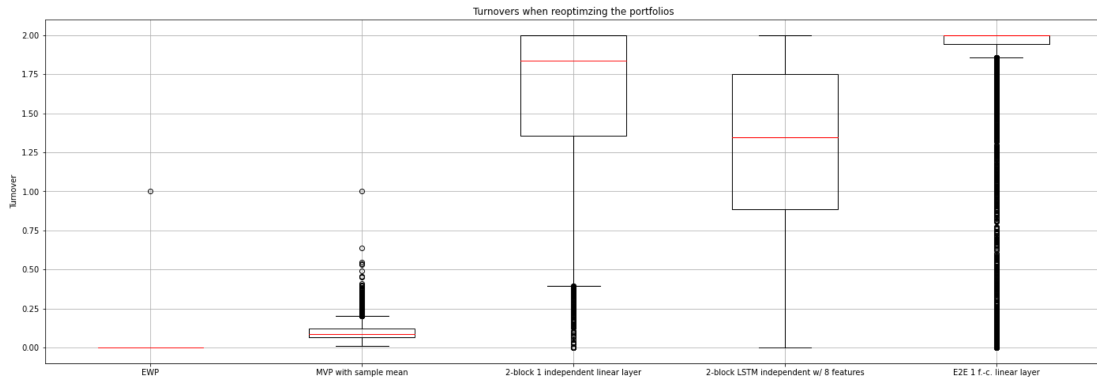

Figure 7.2: Turnovers when reoptimizing the portfolios

Figure 7.2 plots the turnovers of the portfolios calculated by each model at every reoptimization. In total each model does 6000 reoptimizations throughout the whole investment period (6 per window). Turnovers are a measure of how much the portfolio changes and are directly proportional to the transaction costs that they cause. The turnover is defined as follows:

Definition 7.3.1. The turnover TO at time t is $\text{TO}_t = ||w_t - w_{t-1}||_1 = \sum_{i=1}^{N} |w_{t,i} - w_{t-1,i}|$

Since in each portfolio we allocate the whole budget (i.e. the sum of their weights is 1), turnovers are in the range [0, 2]. 0 means that the portfolio remains exactly the same, while a turnover of 2 indicates that all the old positions have been sold and the budget is now allocated completely to assets that were not hold before.

Obviously, the turnovers for the Equally Weighted Portfolio are all 0 (except for the first time we buy the portfolio, when it is 1). The changes in the MVP benchmark are still small, but about the three other models it can be observed that the turnovers are usually close to their maximum value. That means that our models need to alter their positions a lot to exploit the predictability they have found and can not adapt smoothly to changes in the markets.

	Return	**Volatility**	**Max DD**	**Sharpe**
C=0%				
EWP	7.484	1.229	0.204	6.087
MVP with sample mean	5.535	1.029	0.183	5.377
2-block, 1 independent linear layer	19.11	1.606	0.200	**11.89**
2-block, LSTM independent w/ 8 features	11.48	1.505	0.236	7.631
E2E, 1 f.-c. linear layer	14.53	1.873	0.252	7.760
C=0.02%				
EWP	7.472	1.229	0.204	6.077
MVP with sample mean	5.193	1.055	0.186	4.921
2-block, 1 independent linear layer	12.68	1.607	0.208	**7.890**
2-block, LSTM independent w/ 8 features	6.548	1.497	0.253	4.374
E2E, 1 f.-c. linear layer	8.451	1.863	0.253	4.534
C=0.1%				
EWP	7.423	1.229	0.204	**6.037**
MVP with sample mean	3.810	1.055	0.187	3.609
2-block, 1 independent linear layer	-10.0	1.622	0.212	-6.18
2-block, LSTM independent w/ 8 features	-11.4	1.509	0.255	-7.61
E2E, 1 f.-c. linear layer	-16.9	1.880	0.257	-8.99

Table 7.1: Long-term performances incorporating transaction costs

On the other hand, the wealth evolution of the trades carried out with a fee of 0.02% seems at first sight acceptable. The transaction costs appear to wipe out most of the gains we had without them compared to the benchmarks, but it is not clear at all whether they still perform better. For a more extensive analysis, Table 7.1 shows the main performance measures evaluated on the whole trading period of 4 months for each model and transaction fee value.

With $C = 0.02\%$ the LSTM and the end-to-end models considered lose most part of the gains up to the point that the Sharpe ratios indicate that it does not compensate for the risk. However, the 2-block structure with one different linear layer per asset, which provided the best results without transaction costs, still performs better than the benchmarks, both in terms of returns and risk-adjusted returns. This does not mean that the model is more resistant to the changes in allocations, only that it had a bigger margin to lose.

Chapter 8

Conclusions

Motivated by the lack of excellent deep learning based financial investment strategies, this thesis presents a detailed study of various factors that must be taken into account when approaching the portfolio optimization problem via deep learning. The experiments conducted analyze the main obstacles that current research on the topic is facing in terms of model architecture, frequency of the data and the impact of transaction costs. Although the results might seem disappointing at first because none of the models developed performs well enough to be used as a real-life trading algorithm, some important conclusions can be extracted that give hints on how future research on this area should be approached.

Our experiments on models' architectures prove that it is not necessary to leave behind all the classical ideas of financial theory if we want to apply deep learning to portfolio optimization, as end-to-end models perform slightly worse than the ones that only estimated the expected return vector via a neural network and input it to a mean-variance portfolio. This reinforces the idea that the main downside of modern portfolio theory is not what the portfolio formulations are trying to optimize, but how the estimators that model the distribution of the returns are calculated.

Moreover, we notice that to estimate the expected return vector for a mean-variance portfolio it is needless to take into account the correlations and interactions among the assets, meaning that the expected return of each asset can be acceptably estimated as a function of the past returns of only that same asset. This task can be left to the covariance estimator. This helps to alleviate the model complexity, decreases the probability of overfitting and makes it more scaleable so it gives a good starting point for further models.

In terms of the type of data the models should use, posterior simulations confirm that only on minute-by-minute data neural networks can learn features that were invisible to classical portfolios. In this case the Sharpe Ratios provided by the benchmarks can be nearly duplicated on a period

of bull markets (without accounting for transaction costs), while as soon as we lower the frequency to quarterly data these improvements disappear. For further experiments, it would be interesting to work with data of even higher frequency and verify if the noise microstructure becomes more evident and exploitable there.

And the forth key idea is that, even though some of the models implemented seem to detect a lot of predictability in the data, it can not be exploited in real life trading as the transaction costs imposed by exchanges cancel out all the gains. The maximum fee our models could tolerate in order to still perform better than the benchmarks is only 0.02%. This coincides with some of the literature published recently on this topic [22], but is still far from what one would encounter in the most famous exchanges.

Throughout the work carried out in this thesis some obstacles were encountered that should be addressed in future work if we want to obtain statistically significant backtests and understand better what deep learning is truly capable of in portfolio optimization. First of all we should try to reduce the overfitting that was present in most of the models implemented by, for example, using validation sets, dropout layers or more extensive training splits. Although completely avoiding overfitting is almost impossible due to the lack of autocorrelations on the returns, big improvements could be made in this sense. Additionally, a deeper study on the poor performance of the LSTM networks should be done to comprehend if more training epochs are needed or the structure of the inputs and outputs is not appropriate. And, to get the big picture of the performance of the models, tests on bearish markets should also be carried out. In this case we might consider allowing short positions so that the models become less dependent on the markets.

Finally, and most importantly, if we want to design investment strategies based on neural networks that can actually be applied in real life, the trading framework we use must be rethought. Right now the neural networks learn to allocate the budget considering only how well this portfolios are going to perform in the future. However, as transaction costs appear to cancel all the gains, we should force the deep learning to take into consideration how much the positions are being changed at each timestep to find a trade-off between allocating the money to promising assets and avoid the corresponding transaction costs. This falls on the area of reinforcement learning, and, more precisely, deep reinforcement learning (DRL). Approaching the portfolio optimization problem via DRL is still on a very early stage and the developments of this line of research should be followed closely in the future.

Bibliography

[1] Akshay Agrawal, Brandon Amos, Shane Barratt, Stephen Boyd, Steven Diamond, and J Zico Kolter. Differentiable convex optimization layers. *Advances in neural information processing systems*, 32, 2019.

[2] Fischer Black and Robert Litterman. Global asset allocation with equities, bonds, and currencies. *Fixed Income Research*, 2, 1991.

[3] Andrew Butler and Roy Kwon. Integrating prediction in mean-variance portfolio optimization. *Available at SSRN 3788875*, 2021.

[4] Cathy Yi-Hsuan Chen and Christian M Hafner. Sentiment-induced bubbles in the cryptocurrency market. *Journal of Risk and Financial Management*, 12(2):53, 2019.

[5] Rama Cont. Empirical properties of asset returns: stylized facts and statistical issues. *Quantitative finance*, 1(2):223–236, 2001.

[6] Gerard Cornuejols and Reha Tütüncü. *Optimization methods in finance*, volume 5. Cambridge University Press, 2006.

[7] Marcos Lopez De Prado. *Advances in financial machine learning.* John Wiley & Sons, 2018.

[8] Yiyong Feng, Daniel P Palomar, et al. *A signal processing perspective on financial engineering*, volume 9. Now Publishers, 2016.

[9] Thomas Fischer and Christopher Krauss. Deep learning with long short-term memory networks for financial market predictions. *European Journal of Operational Research*, 270(2):654–669, 2018.

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

[11] Thomas E Koker and Dimitrios Koutmos. Cryptocurrency trading using machine learning. *Journal of Risk and Financial Management*, 13(8):178, 2020.

[12] Marko Kolanovic and Rajesh T Krishnamachari. Big data and AI strategies: Machine learning and alternative data approach to investing. *JP Morgan Global Quantitative & Derivatives Strategy Report*, 2017.

[13] Jan Krepl. Deepdow: Portfolio optimization with deep learning (v0.2.0), June 2020. Zenodo. `https://doi.org/10.5281/zenodo.3906121`.

[14] Andrew W Lo. The statistics of Sharpe ratios. *Financial analysts journal*, 58(4):36–52, 2002.

[15] Yilin Ma, Ruizhu Han, and Weizhong Wang. Prediction-based portfolio optimization models using deep neural networks. *IEEE Access*, 8:115393–115405, 2020.

[16] Harry M Markowitz. *Portfolio selection*. Yale University Press, 1968.

[17] John Moody and Matthew Saffell. Learning to trade via direct reinforcement. *IEEE transactions on Neural Networks*, 12(4):875–889, 2001.

[18] John Moody, Lizhong Wu, Yuansong Liao, and Matthew Saffell. Performance functions and reinforcement learning for trading systems and portfolios. *Journal of Forecasting*, 17(5-6):441–470, 1998.

[19] Michael A Nielsen. *Neural networks and deep learning*, volume 25. Determination press San Francisco, CA, USA, 2015.

[20] Christopher Olah. Understanding LSTM networks. 2015. `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`.

[21] William F Sharpe. Mutual fund performance. *The Journal of Business*, 39(1):119–138, 1966.

[22] Chao Zhang, Zihao Zhang, Mihai Cucuringu, and Stefan Zohren. A universal end-to-end approach to portfolio optimization via deep learning. *arXiv preprint arXiv:2111.09170*, 2021.

[23] Zihao Zhang, Stefan Zohren, and Stephen Roberts. Deep learning for portfolio optimization. *The Journal of Financial Data Science*, 2(4):8–20, 2020.