Imperial College London

MATH70129 - Portfolio Management

Imperial College London

Department of Mathematics

Assessed Coursework

Authors:
Max Dedieu (02289032)

Lecturer:
Johannes Muhle-Karbe

May 9, 2023

Contents

1 Introduction

Portfolio management is a dynamic field that has evolved significantly over the years. The field is concerned with making investment decisions and allocating assets to achieve the desired financial goals. The mathematical foundations of portfolio management, such as the Markowitz model, have long been used to optimize portfolio returns while minimizing risk. However, with the advent of technology, new techniques have emerged that promise to revolutionize portfolio management.

One such technique that has garnered significant interest in recent years is reinforcement learning. Reinforcement learning is a machine learning technique that allows an agent to learn how to make decisions by trial and error. It involves identifying the best course of action by rewarding the agent for positive outcomes and punishing it for negative ones. The potential applications of reinforcement learning in portfolio management are vast, as it could help optimize the allocation of assets and improve portfolio returns.

Moreover, the rise of cryptocurrencies has brought about new challenges and opportunities in portfolio management. Cryptocurrencies have gained widespread attention and adoption, making them an increasingly popular investment option. However, the volatility and complexity of cryptocurrencies make them a challenging asset to manage. As such, the application of reinforcement learning techniques to the management of crypto portfolios could be particularly promising.

This coursework seeks to explore the potential benefits of using reinforcement learning to manage a portfolio of cryptocurrencies. The aim of this work is to study and implement the result of the article [2]. We will dive into acquiring and shaping the data, designing the agent and the state-reward environment, and lastly evaluate its performance against multiple "dummy" portfolio. We will use the programming language Python to carry out this coursework. Mainly, we will use the libraries *Numpy*,*Pandas* and *PyTorch*. All the code for this coursework can be found on Github.

2 General setting

2.1 The Cryptocurrency Market

The cryptocurrency market has been evolving very rapidly in recent years, mainly after the COVID-19 crisis in 2020. Growing interest and applications have helped increase the market cap of some assets ("coins") to record levels. The largest coin currently traded is Bitcoin, at just over 530B USD in March of 2023. The cryptocurrency market has a large offering of assets, some more risky and volatile than others. There are some strong correlations between some assets, so we need to be careful when creating a portfolio if we want to benefit from diversification. As of March 2023, the largest cryptocurrencies are: Bitcoin (BTC), Ethereum (ETH), Binance Coin (BNB), Cardano (ADA), Tether (USDT), XRP (XRP), Solana (SOL), Polkadot (DOT), Dogecoin (DOGE), and Avalanche (AVAX). At the time of the paper [2], Bitcoin was a steady asset and not the speculation vector that it now is. This is why the authors chose it as a risk-free "cash" asset. We understand this is no longer a viable assumption, as better alternatives, such as Tether, have emerged. For the next parts, we assume the risk-free asset is USDT (Tether).

2.2 Assumptions and Objectives

The cryptocurrency market is "24/7", with no down periods like in other exchanges such as NYSE or Euronext. This means there is a constant stream of information we need to analyze and act on. In this manner, we wish to manage a portfolio of $m + 1$ cryptocurrencies (m risky assets and 1 risk-free asset) by rebalancing the weights attributed to each asset at a fixed interval. In the original paper, this interval was every 30 minutes, but we will experiment with different intervals, as it can be seen as a hyperparameter among others. The goal will be to use market data from X minutes into the past to make the best decision about the new weights and evaluate the performance of those weights at the end of the next X minutes.

We make assumptions about the market and the agent to simplify the model. We suppose that we do not have any impact on the market, and that there is no slippage. This means that our agent will be able to trade immediately for the whole size it desires, at the price observed in the market. This allows faster decision making and shifts the complexity of the model from the execution of the orders to the weight allocation.

We establish that we cannot go short of an asset, and have to invest all of our capital at each time-step. This means the weight vector $w_t \in \mathrm{R+}^{m+1}$ has all its elements summing up to 1 for each time-step $t \geq 0$. We also take into account trading fees to create a more sophisticated model. Most crypto-exchanges deal in fixed rate fees. This means we pay a fixed percentage p of the notional for each transaction. Granted we can exchange any asset for any other asset, the formula for the fees f_t (from time $t-1$ to time t) is:

$$f_t = \frac{p}{2} \sum_{i=1}^{m+1} |w_{t,i} - w_{t-1,i}| \tag{1}$$

Where w_{t-1} and w_t are portfolio vector at time $t-1$ and t. We need to discount any return we make on period t by a factor of $(1-f_t)$ to take into accounts the fees.
As a convention, when writing down the weight vector, we always assume the risk free asset is the last element of the weight vector (the $m+1^{\text{th}}$ element), namely, we start each simulation with an initial allocation :

$$w_0 = (0, 0, ..., 0, 1) \in \mathrm{R}^{m+1} \tag{2}$$

2.3 Reinforcement learning concepts: The state, the actions and the rewards

In reinforcement learning, we develop a machine learning model called an agent that, when provided with a snapshot of the world (i.e., a state), chooses an action from the action space and receives a reward based on the relevance of that action. The goal of training an agent is to teach it to select the action that yields the best reward for a given state. Various methods exist for achieving this, depending on the state and action space. If the action space is finite (e.g., a Tic-Tac-Toe board), we can assign a desirability value to each state and select the action that leads to the state with the highest desirability at each step. However, when the state space is non-finite, as is the case here, we use a Q function that returns the desirability of a (state, action) pair.

In this project, we define the weights of the portfolio at time t, referred to as w_t, as the agent's action, and the environment's state as a tensor S_t combined with the previous weights w_{t-1}. By including the previous weights w_{t-1}, we can account for trading fees, which enables the agent to avoid changing weights too frequently and capture any future returns that are strictly non-negative. In the following sections, we will discuss the structure of the $(S_t)_{t\geq 0}$ in greater detail, but for now, we assume that it contains the prices of the assets in our environment, grouped by time period such that we can compute returns for a fixed period with a single state S_t.

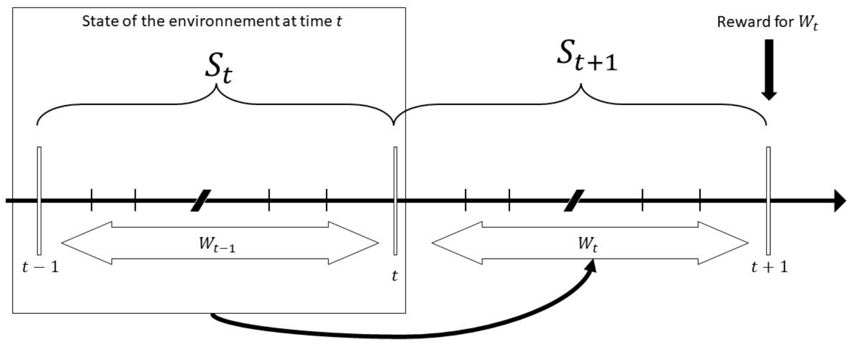


Figure 1: Schematics of the environment evolution over time for the reinforcement learning process

As seen in figure 1, the state (S_t, w_{t-1}) is used by the agent to select the next action w_t. It is important to note that the tensor S_t is composed of the last n data-points, meaning the most recent data-points in S_t is the data-points at time t. This ensures we do not use future information.
For a series of state and actions between time $t=1$ and $t=t_f$, we define the reward R as below :

$$R(S_1, w_1, ..., S_{t_f}, w_{t_f}) = \frac{1}{t_f} \sum_{t=1}^{t_f} \log((1-f_t)\Gamma(S_t) \cdot w_t)$$

Where Γ is the return function, that gives the prices returns from a price state S_t. We note the term $\frac{1}{t_f}$ which allows us to compare rewards for different lengths of simulation, as we will see later this is helpful for

training.
As mentioned in the original article, there are a few important distinctions that set this problem apart from regular reinforcement learning settings. The most important one is that the actions of the agent don't influence the state of the environment (apart from the previous weight), this is because of the no-impact hypothesis. Most Reinforcement learning problems have the agent directly impact its environment, but here the constitution of a crypto-portfolio don't influence the prices of the underlying assets. This is hugely helpful for training, because we can come to any point in time and restart the simulation from there, we can evaluate many strategies concurrently because the state is the same for a given time step. Secondly, the actor re-balances its portfolio at a given interval, but it has no constraints to go from a trading strategy to a completely different one. This means the actor's actions don't hold any repercussions on its future rewards, and that the reward function is fully known. In other settings, one must evaluate the reward function using Bellman's equation and possibly another Machine learning model (see [3]). However here the reward for a given action if fully expresses as $\log((1 - f_t)\Gamma(S_t) \cdot w_t)$. This also makes the training much simpler, as we will see in the next parts.

3 The data pipeline

Reinforcement learning is a subset of machine learning, and as such, requires a tremendous amount of data to be trained on. The first part of our project will be to acquire such data, and shape it as presented in the article.

3.1 Downloading the data

We acquire the data from the online exchange coinbase. The ease of access of their API and the high requests limits allows us to download a lot of data to train our agent. We access the candlestick "OHLC" data of cryptocurrencies with fixed period-size, called "granularity" (by chunk of 15-minutes for example) between a start date and an end-date. There is a hard-limit of data one API call can return, so we add some logic in the code to divide the process in multiple calls and concatenate the results together. In the end, we obtain a *pandas* dataframe as shown below:

	Close		High		Low	
Date	BTC	ETH	BTC	ETH	BTC	ETH
2023-04-11 12:00:00	30152	1918	30155	1918	30064	1914
2023-04-11 11:45:00	30079	1916	30102	1918	30074	1916
2023-04-11 11:30:00	30098	1917	30144	1919	30079	1916
2023-04-11 11:15:00	30125	1919	30150	1922	30103	1918
2023-04-11 11:00:00	30131	1920	30147	1921	30117	1920
...	...	...	...	...	...	...

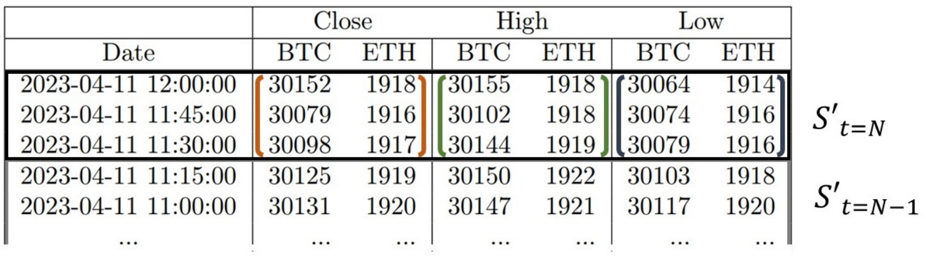
Table 1: Sample raw Dataframe of Closes,Highs and Lows of Bitcoin and Etherum for a set period and a granularity of 15 minutes

There are some missing values in the downloaded data. We have to be careful as we cannot blindly either drop all the N/A values or fill them with interpolation for instance. The article explores the possibility of filling the missing data with dummy data with a decay rate. This means a time-series with the most recent non-N/A price of S_0, N period of missing data and a decay rate α will have the missing data S_t filled as such $S_t = S_0 * (1 - \alpha)^i$.
In our case, the missing data are filled with the most recent past non-N/A value in the the dataframe, and dropped otherwise. This ensures we do not use future information.

3.2 Processing the data

Now that we have acquired the data, we need to process them so that we can feed it into our autonomous agent. The data has higher dimensionality than a standard time-series. We can view our data in through a time perspective (date-time index), a "type" perspective (Close, High, Low), and the asset perspective. As we can see the data frame 1 is of shape $(N, 3, m)$.
In reinforcement learning, an important aspect of training is giving context to the agent. Should we provide time series for each type/asset, one date at a time, the agent couldn't understand the overall trend of the price, and couldn't make an informed decision. This is something that was taken into account by Deep-Mind

when training their Atari agents [1]. The team is stacking frames of the game together and feed it into the network batch by batch, so the agent can detect a trend (direction of the ball in Pong for example).
In this fashion, we introduce a hyper-parameter n the "number of input period before t", or the "context-length". Our aim is to pass to the agent data-points taken in the intervals $[k.n, (k+1).n]$. Moreover, we need to express the data in a normalized way so that the agent can generalize. Indeed, the assets might have very different prices (BTC is in the order of 30,000USD and ETH in the order 1,900USD for instance). The data passed to the agent should be in the same order of magnitude for all the assets, to make the learning process easier. The article [2] proposes a way to express the prices in such a way.



	Close		High		Low	
Date	BTC	ETH	BTC	ETH	BTC	ETH
2023-04-11 12:00:00	30152	1918	30155	1918	30064	1914
2023-04-11 11:45:00	30079	1916	30102	1918	30074	1916
2023-04-11 11:30:00	30098	1917	30144	1919	30079	1916
2023-04-11 11:15:00	30125	1919	30150	1922	30103	1918
2023-04-11 11:00:00	30131	1920	30147	1921	30117	1920
...	...	...	...	...	...	...

$$S'_t = \left[V_t^{(C)}, V_t^{(H)}, V_t^{(L)} \right]$$

Figure 2: Visualization of the data processing mechanism on the same data ($n = 3$)

Calling m the number of assets in our environments, N the number of batch of n time-step in our dataset ($\textit{total size} = n * N$), the dataframe 1 is a stack of N rows of 3 elements ("Close","High","Low"). We call the "close","open" and "low" matrix $V_t^{(C)}$,$V_t^{(H)}$ and $V_t^{(L)}$. Each of these 3 elements is a matrix of prices for the m assets, in the last n trading periods before t. $V_t^{(X)} = [P_{t,1}^{(X)}, ..., P_{t,n}^{(X)}]^T$. The prices vectors $P_{t,i}^{(X)}$ are of size m, where $i \in [1, n]$. We call the vector $(V_t^{(C)}, V_t^{(H)}, V_t^{(L)})$ the time-batch $(S'_t)_{t\geq 0}$ (described above in figure 2). Should we remove the indexes, the data would now be a tensor of order 4, shaped like $N, 3, m, n$. We can see the new shape of our date on figure 3.

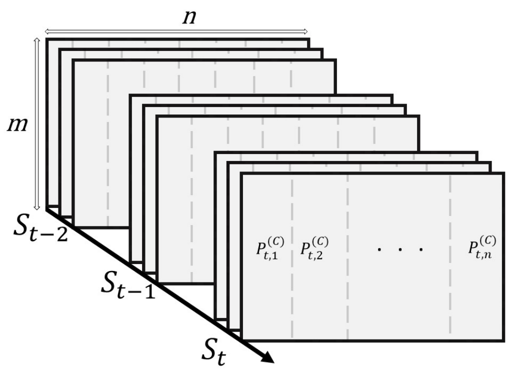


Figure 3: Visualization of the aggregated data and tensor $(S'_t)_{t\geq 0}$

As we can see in figure 2, for each time-batch t, the data is not normalized, it contains the raw prices of the assets, re-ordered by batch. To normalize the data, we introduce the element-wise division operator noted $\oslash$. Taking two vector in $\mathbf{R}^p$ $x = (x_1, ...x_p)$ and $y = (y_1, ..., y_p)$ with non-null elements, we have :

$$x \oslash y = (\frac{x_1}{y_1}, ..., \frac{x_p}{y_p}) \in \mathrm{R}^p$$

We then introduce the normalization operator Φ, which takes in raw prices input and normalize them as below :

$$\Phi(S'_t) = \Phi\left(\left[V_t^{(C)}, V_t^{(H)}, V_t^{(L)}\right]\right) = \left[W_t^{(C)}, W_t^{(H)}, W_t^{(L)}\right] = S_t$$

$$\begin{array}{lcllll} W_t^{(C)} & = & [& P_{t,j}^C \oslash P_{t,n}^C & \cdots & P_{t,n-1}^C \oslash P_{t,n}^C & \mathbf{1} &] \\ W_t^{(H)} & = & [& P_{t,j}^H \oslash P_{t,n}^C & \cdots & P_{t,n-1}^H \oslash P_{t,n}^C & P_{t,n}^H \oslash P_{t,n}^C &] \\ W_t^{(L)} & = & [& P_{t,j}^L \oslash P_{t,n}^C & \cdots & P_{t,n-1}^L \oslash P_{t,n}^C & P_{t,n}^L \oslash P_{t,n}^C &] \end{array}$$

We divide each individual price vector of time-batch by the last $(j = n)$ close price vector, such that the price vector are in the order of magnitude of $\simeq 1$. We can observe a visualization of the state tensor S_t on figure 4.

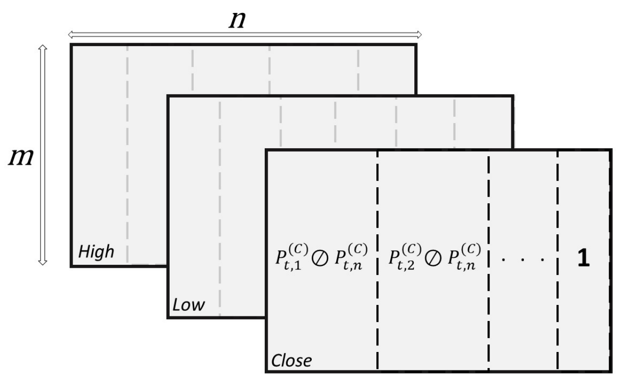

n
m
High
Low
Close
$P_{t,1}^{(C)} \oslash P_{t,n}^{(C)}$
$P_{t,2}^{(C)} \oslash P_{t,n}^{(C)}$
. . .
1

Figure 4: Visualization of a state tensor S_t

4 The Agent

4.1 Architecture

The agent is a machine learning model that take as input the state of the environment and outputs the best action for this particular state. There are many designs possible, for instance, if the long term reward is uncertain, we introduce the Q function that returns the 'quality' of a *(state,action)* pair. From there for a fixed state s, we find the action a that maximizes the value $Q(s,a)$, which is $\text{argmax}_{a\in A}Q(s,\cdot)$. In the more sophisticated models, we use a "policy" model that outputs $\text{argmax}_{a\in A}Q(s,\cdot)$, and a critic model that outputs $Q(s,a)$. This is used in training to improve the prediction of the actor "policy" model (see [3]).
In our case, the long-term is fully known. As such, we create a "policy", or "actor" network that we call π, such that $\pi : s \Longrightarrow \text{argmax}_{a\in A}Q(s,\cdot)$.
There are several architecture proposed by the original paper[2], from Recurrent Neural Network (RNN) to Convolutions Neuron Network (CNN). In this work, we will only implement the CNN architecture, as it is the one with the best and most consistent performance in the back-test shown in the latter part of the paper. The architecture of the CNN is shown in figure 5

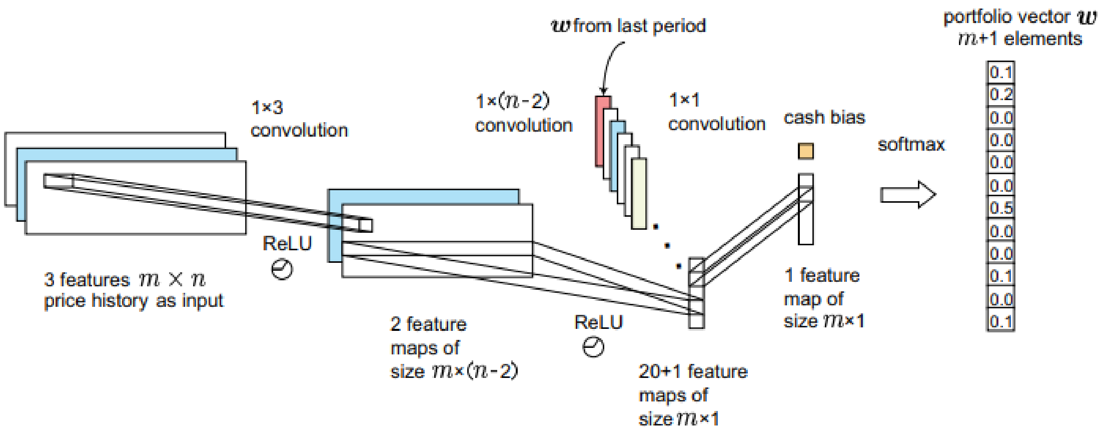


Figure 5: CNN architecture of the π-network, where $n = 50$, generalized from "Zhengyao Jiang, Dixing Xu, and Jinjun Liang. A Deep Reinforcement Learning Framework for the Financial Portfolio Management Problem. 2017. arXiv: 1706.10059" [2]

In the following sections, we use m to denote the number of risky assets, we then have $m+1$ assets in the environment (by accounting for the unique risk-free asset. The π-network is supplied with the data S_t (size $3, m-1, n$). We can observe some similarity with a CNN used in image recognition for instance, since the input feature map is of size 3 ("RGB" format). We will dive into this detail later (in the section on "E.I.I.E"), but we can first observe that the convolution kernels all have heights of 1. From the input, we apply a convolution transform to go from depth 3 to depth 2, we then apply the non-linear function $ReLU : x \longrightarrow max(x, 0)$. With the help of another convolution transform and $ReLU$ function, we reduce the dimension of our data from (3,m,n) to (20,m,1). We then introduce a new input, we use the weight vector w_{t-1} from the last step, without the risk free weight ($w_{t-1} \in \mathbf{R}^m$). As shown above, the transaction cost are computed based on the weight vector from time t and $t-1$. We include w_{t-1} in the state of the environment, so that the network can take it into account and reduce the attempt to profit off of very small returns. We concatenate the weight vector $w_{t-1}^T \in \mathcal{M}_{m,1}(\mathbf{R})$ to the transition state of the C.N.N. Using a last convolution kernel of size 1x1, we flatten the result to obtain a state in $\mathcal{M}_{m,1}(\mathbf{R})$. We are not quite finished yet, since we need to obtain a weight vector in $\mathbf{R}^{m+1}$. We add a new neuron "cash-bias", filled with a constant value ρ. We then apply a softmax function defined as below to each input $x_i, i \in \{1, .., m+1\}$:

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)} = y_i$$

This ensures the outputs y_i are positive, and $\sum_i y_i = 1$. We can use this output directly without any processing as the new weight vector w_t. The cash bias ρ is used to off-set the other neurons in the last layer of the CNN. The higher the cash bias, the more weights will be allocated to cash (the last component of the weight vector). This parameter can be seen as a heuristic for our model, it is a hint of how "cash-averse" the agent should be. In practice, we choose an initial value ρ_0 and let the model fit around this value. We also propose a technique where we evaluate the values of the neurons before the Sigmoid function, collect the mean value $\bar{\rho}$ and set the new cash bias value : $\rho \leftarrow r_\rho \rho + (1 - r_\rho)\bar{\rho}$, where $r_\rho \in]0, 1]$.

4.2 Identical Independent Evaluator

The architecture employed in this study uses a technique called Identical Independent Evaluator (I.I.E.) to evaluate the performance of the model on different types of assets. I.I.E.s are also referred to as mini-machines and are evaluated independently of each other until the last layers of the model. The convolution kernels have a height of 1, allowing each evaluation of one asset to be independent from the other ones. As a result, each mini-machine acts like its own independent network, it is evaluated based on its own set of weights and biases, fitted on historical performance, and the current most recent returns. The results are then normalized using the softmax function introduced earlier, so each asset's potential is translated into portfolio weights.

This combination of mini-machine is called an ensemble of I.I.E. (E.I.I.E.). The use of E.I.I.E. reduces the impact of individual evaluators' biases and errors, leading to a more accurate and trustworthy evaluation of the model's performance. This is especially beneficial when dealing with subjective or difficult-to-quantify evaluation metrics, small or unrepresentative data-sets, and situations where the model needs to be adapted to multiple types of assets.

There is three main advantages of using E.I.I.E, as the paper suggests [2]. Firstly, it is scalable in asset number, with the mini-machines being identical with shared parameters, so the training time of an ensemble scales roughly linearly with m. Secondly, it is data-usage efficient, as a mini-machine can be trained m times across different assets for an interval of price history, allowing asset assessing experience of the IIEs to be shared and accumulated in both time and asset dimensions. Finally, it is flexible in terms of asset collection, with an IIE's asset assessing ability being universal without being restricted to any particular assets, allowing an EIIE to update its choice of assets and/or the size of the portfolio in real-time without the need to retrain the network from scratch.

4.3 Training

The policy network we will use is a CNN, which means it can be trained using standard gradient-descent techniques, against a defined loss. The original paper provides the process to train the agent.
To train the agent, the paper introduces what it calls a PVM (Portfolio Vector Memory). This is a matrix of size $(m+1) \times N$, it represents the portfolio weights stacked across the N time-steps of the training data. The matrix is initialized the initial weight vector w_0 defined in 2.
The agent is trained using Online Stochastic Batch Learning (OSBL) We mentioned that giving context to the agent when evaluating a state is important, but it is as important when training the agent. We have seen the expression of the terminal reward at equation 1. We need multiple training steps to express the terminal reward, and we cannot use one particular *(state,action)* pair. This is where we use OSBL. We introduce a batch size n_b, and number of batch N_b. After a training step at time t, where the agent observed the state of the environment (S_t, w_{t-1}) and decided on an action w_t, we sample N_b starting time-step $t_{b_i}, i \in [1, N_b]$. The $(t_{b_i})_{i\in[1,N_b]}$ are sampled using a geometric law of parameter β, such that :

$$P_\beta(t_b) = \beta(1-\beta)^{(t-t_b-n_b)} \tag{3}$$

The batches contain n_b time-steps, such that following 3, the always end before time $t = t$. We sample a number N_b of these batches. For each batch, we can compute the truncated terminal reward using 1 (we replace t_f by n_b). We then use gradient-ascent against this reward to update the weights and biases of the CNN. We use gradient-ascent and not gradient-descent which is commonly used because we want to maximize the reward which we treat as our loss (we usually want to decrease the loss over-time). A commonly used method to increase the stability of the training process is to use separate target and regular policy networks. We use the target network for evaluation of prediction of future weights, and the regular network for weights updating. At regular time interval, we transfer the weights from the regular network to the target network, with an inertia term τ.

$$W_{\text{target}} \longleftarrow \tau W_{\text{regular}} + (1-\tau) W_{\text{target}}$$

We initialize the weights using Xavier weight initialization. The goal of this weight initialization is to set the initial weights of the network in a way that helps it converge faster and more reliably during training. The Xavier initialization method is designed to ensure that the variance of the outputs of each layer is roughly equal to the variance of its inputs, so that the activations neither vanish nor explode as they propagate through the layers. The method achieves this by setting the initial weights of each neuron to be drawn from a Gaussian distribution with a mean of 0 and a variance of:

$$\frac{1}{\text{fan-in} + \text{fan-out}}$$

Where fan-in is the number of input connections to the neuron, and fan-out is the number of output connections from the neuron.The intuition behind this formula is that the variance of the weights should be inversely proportional to the number of inputs and outputs, in order to balance the scale of the activations across layers. By setting the variance in this way, the Xavier initialization can help to mitigate the vanishing and exploding gradient problems that can arise during training, and lead to better performance and faster convergence.
We sum-up the training procedure in below algorithm 1.

Algorithm 1 Training algorithm for the Agent

```
Initialize memory
Initialize network and target network
$\theta_{\text{target}} \leftarrow \theta$                    ▷ Initialize $\pi$ and $\pi_{\text{target}}$ with the same same weights
for number of epoch do
    for timestep in range(number of episodes) do
        $w_{t-1} \leftarrow$ previous portfolio
        $S_t \leftarrow$ current state
        $w_t \leftarrow \pi(w_t t - 1, S_t)$
        memory[timestep] $\leftarrow w_t$
        if $\#_{\text{episode}} \geq n_b$ then
            for batch in range($N_b$) do
                $t_b \leftarrow P_\beta^{(\text{geo})}(\text{timestep}, n_b)$
                $M \leftarrow (S_t, w_t)_{t \in [t_b, t_b + n_b]}$
                $R \leftarrow Reward(M)$
                $\theta \leftarrow \theta - \eta \nabla_\theta R$
            end for
        end if
        $\theta_{\text{target}} \leftarrow \tau\theta + (1-\tau)\theta_{\text{target}}$
    end for
end for
```

5 Assessing the performance of the agent

5.1 General settings

Now that we have created the agent and its environment, we will have it learn on a training data-set and test its performance on a test data-set. As mentionned before, we gather the data using coinbase's API. We will use historical data on the past four months of the market, using a 3-1 split (3-fourth of the data used in training, 1-fourth in testing) as shown on table 2.

	Start date	End date
Training data-set	2022-12-28 06:55:13	2023-03-28 06:55:13
Test data-set	2023-03-28 07:42:00	2023-04-27 07:42:00

Table 2: Date for the test and training data-set

We will use in our environment eight currencies: Bitcoin (BTC), Ethereum (ETH), Binance Coin (BNB), Cardano (ADA), Tether (USDT), Solana (SOL), Polkadot (DOT), Dogecoin (DOGE), and Avalanche (AVAX). At the beginning of the simulation, we assume that all our capital is invested in Tether (USDT). We choose a granularity of 900 seconds, meaning the Highs, Lows, and Closes are computed at 15-minute intervals. We will also test our agent against an environment with various fees, knowing that trading fees are always a huge challenge for automated trading systems. Moreover, the train period also has a large drawdown event toward the end. Our model will have to adapt to this, as to not lose the accumulated wealth. The test period also has large returns for some of the coins. It will be interesting to see how the model will adapt to the transition from the training set to the test set and how well it can generalize to new data.

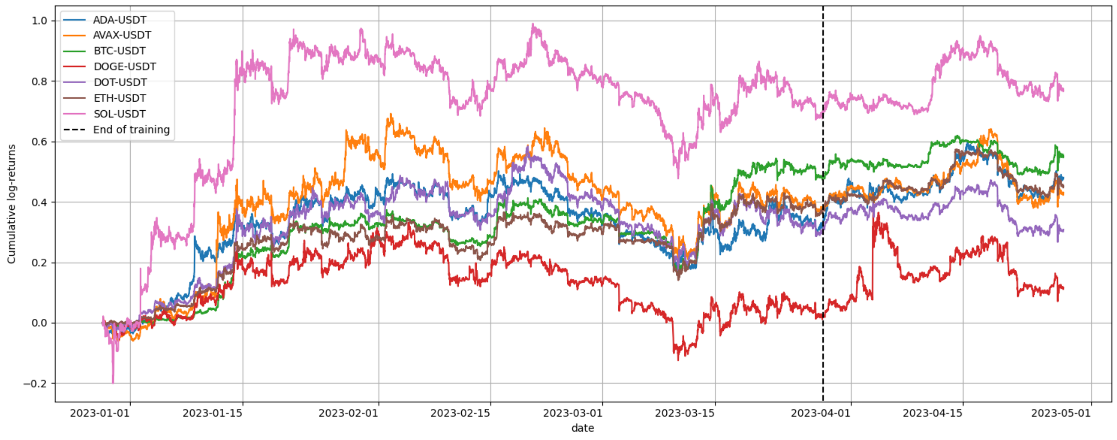


Figure 6: Cumulative log-returns of the assets in the environment on the training and testing period (separation between the two period is marked by a black dashed line)

5.2 Performance metrics

There are several metrics we can use to test a portfolio's performance on a fixed period. The most obvious metric is the return (annualized), but this does not take into account the risk profile of a potential investor. In this manner, we introduce the Sharpe ratio S_r and the maximum drawdown MD :

$$S_r = \frac{\mu - r}{\sigma}$$

$$MD_t = \frac{P_t^C}{\max_{t-s \leq \tau} P_s^{(C)}} - 1$$

where τ is fixed time-span

Now that we have metrics to evaluate the performance of a portfolio, we need to create "dummy" portfolios to assess the value of our agent over a "naive" investor. We can create many strategies in this way, but we will select three test portfolios:

- The Value Portfolio: Choose the best-performing asset for the first half of the test period (with some look-ahead bias) and assign all the wealth in this asset for the entire duration.
- The 1/N Portfolio: Assign a wealth of $\frac{1}{m+1}$ to each asset in our environment, without considering the correlation of risk.
- The Markowitz Portfolio: Use portfolio theory to construct the Markowitz portfolio without rebalancing during the test period. We will use a random risk-aversion parameter λ for the optimization.

5.3 Results

Using the metrics defined above, we will assess the performance of all the portfolios cited above. We are interested in sheer returns, risk management and robustness between train and test sets. We will study two settings, the first one without fees, to assess the maximal potential of our network, and the second with realistic fees, to see how the network would perform in a real-life setting.

5.3.1 No transaction fees

Without transaction fees, the agent will thrive, since it is not penalized for acquiring the smallest non-negative return. We compile the results of the tests in table 3.

	Training data-set				Test data-set			
	Final NAV	% Return	S_r	MDD	Final NAV	% Return	S_r	MDD
Agent Portfolio	**4047**	**305%**	**5.67**	**-9.1%**	**1193**	**19.3%**	**2.63**	**-9.6%**
Value Portfolio	2214	121%	1.96	-38%	1068	6.82%	0.94	-17%
1/N Portfolio	1495	49.5%	1.76	-24%	1047	4.72%	0.88	-13%
Markowitz Portfolio	1648	64.8%	2.35	-21%	1042	4.2%	0.85	-11%

Table 3: Table of performance of the agent's portfolio and other sample portfolio for the training periods in 2, no fees

As we can see, the best performing portfolio is the actor's portfolio, both on the train-set and test-set. The second best performing is the Value portfolio (picking a good performing coin and allocating 100% of the wealth on it). The results are very satisfying; the actor portfolio manages to capture the best return, with a Sharpe ratio of 5.67. From a risk management standpoint, the maximum drawdown is well-managed, considering the riskiness of the underlying assets and the recent turmoil in the market. We can plot the actual wealth of the different portfolios to have a better idea of how they would perform. We show the performance of the agent's portfolio on the train-set and test-set in Figure 7 and Figure 8, respectively.

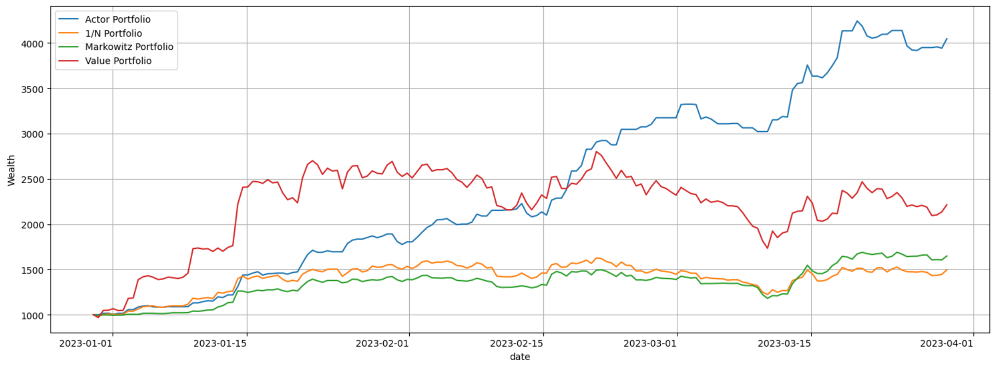


Figure 7: Performance of the portfolios on the training sets with no transaction fees

On the test sets, the results are also looking promising, as we can observe on figure 8.

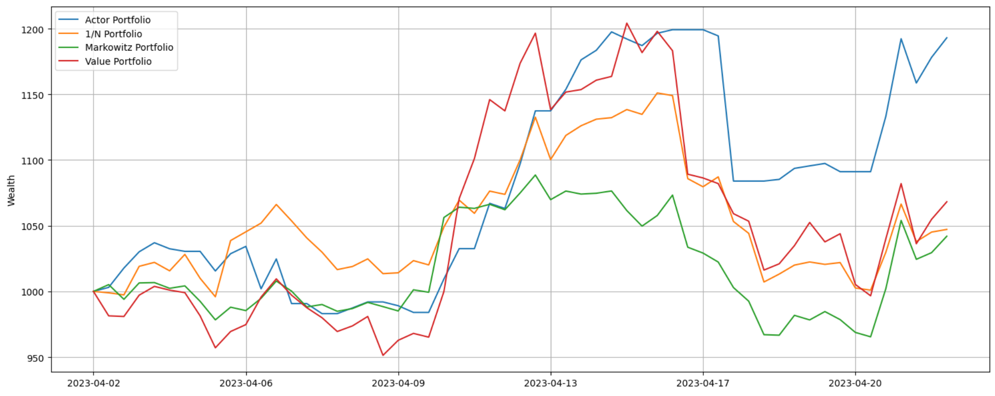
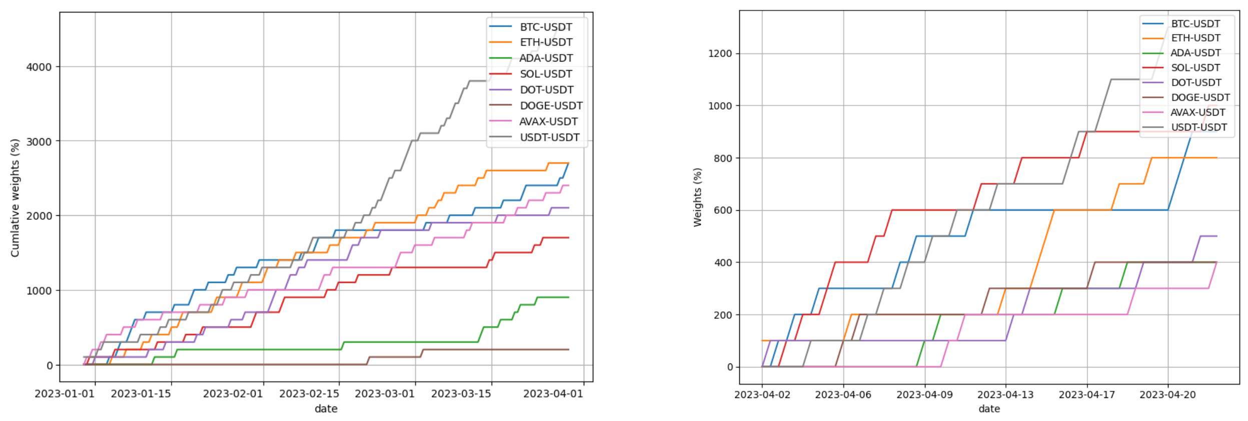


Figure 8: Performance of the portfolios on the test sets with no transaction fees

To get a better idea on how the portfolio is balancing the weight, we can plot the cumulative weights over time, on figure 9a and 9b. We see that there is a large part of wealth allocated to cash, this matches with periods of turmoil in the market if we refer to the individual cumulative log returns on figure 6.



(a) Training set (b) Test set

Figure 9: Cumulative weights of the actor, no transaction fees

5.3.2 Transaction fees

We previously explored the performance of the agent's portfolio without transaction fees, which is a scenario more ideal than what we see in practice. Using data from Coinbase, we can refer to their fee benchmarks to train the agent (see Table 4).

Tier			Taker Fee	Maker Fee
0	-	10K	60 bps	40 bps
1K	-	50K	40 bps	25 bps
50K	-	100K	25 bps	15 bps
100K	-	1M	20 bps	10 bps
1M	-	15M	18 bps	8 bps
15M	-	75M	16 bps	6 bps
75M	-	250M	12 bps	3 bps
250M	-	400M	8 bps	0 bps
400M	-		5bps	0 bps

Table 4: Trading fees on the online exchange coinbase, pulled from [4]

As we can see, the fees depend on the tier of the user and the type of order. The tier corresponds to the volume of orders on a rolling monthly basis. The type of order depends on whether the order adds liquidity to the book or removes it. This means that if the user sends a market order, it creates liquidity in the market and the fees are less significant. On the contrary, if the user fills an already existing order, they will pay the cost of reducing the volume of the order book as an extra fee to their transaction. We use a fixed transaction cost of 25bps (= 0.25%).

We note that the transaction fees will not influence the performance of the other portfolios since they do not rebalance the wealth during the simulation.

	Training data-set				Test data-set			
	Final NAV	% Return	S_r	MDD	Final NAV	% Return	S_r	MDD
Agent Portfolio	**2476**	**147.7%**	**3.72**	**-13%**	**1151**	**15.1%**	**2.5**	**-6.4%**
Value Portfolio	2214	121%	1.96	-38%	1068	6.82%	0.94	-17%
1/N Portfolio	1495	49.5%	1.76	-24%	1047	4.72%	0.88	-13%
Markowitz Portfolio	1648	64.8%	2.35	-21%	1042	4.2%	0.85	-11%

Table 5: Table of performance of the agent's portfolio and other sample portfolio for the training periods in 2, 25bps fees per transaction

As seen on table 5, the performance of the actor's portfolio are significantly reduced compared to the "no-fees" environment. However, the actor's portfolio remains the best performing one out of all the trading strategies. The drawdown are well managed both on the test and train set. The Sharpe ratios are well above the other portfolios. We can also plot the wealth over time in the test and train set to understand better how it evolves, on figure 11 and figure 10

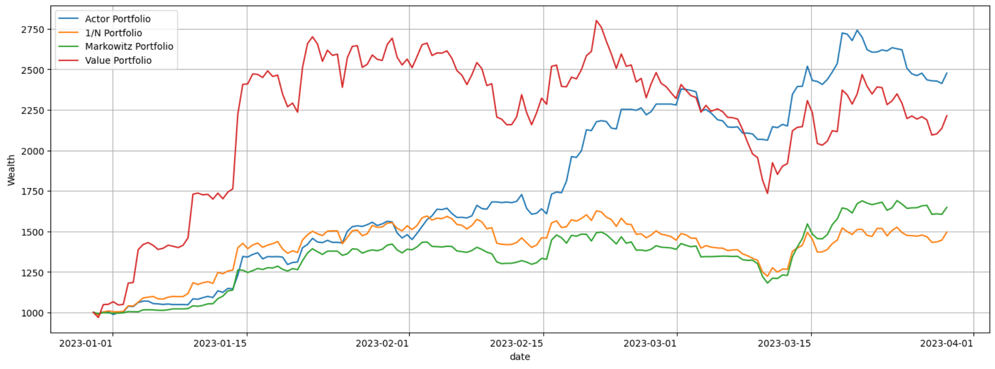


Figure 10: Performance of the portfolios on the training sets with 25 bps transaction fees

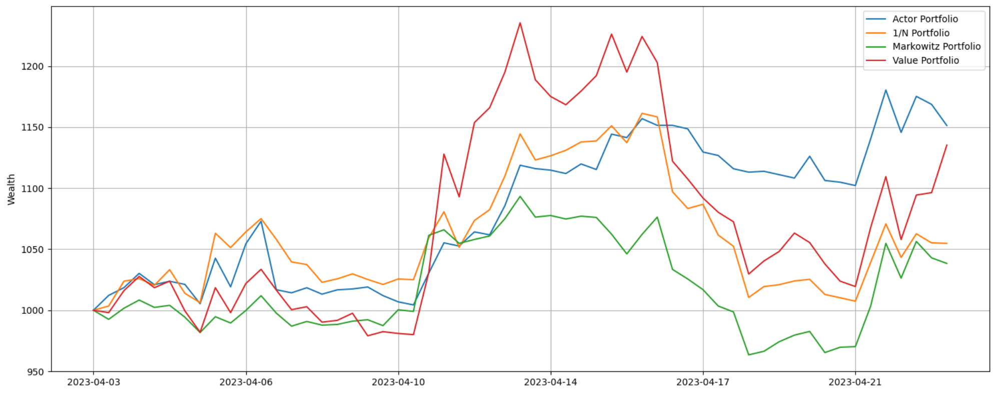


Figure 11: Performance of the portfolios on the test sets with 25 bps transaction fees

6 Conclusion

This coursework presents a Reinforcement Learning (RL) agent for portfolio management in the cryptocurrency market. The agent was implemented for a portfolio consisting of the seven most traded cryptocurrencies, and the data was processed to account for the model's inputs. After presenting the main steps of an RL problem, including the state, agent action, and reward, we explained the entire process of shaping the raw data. Then we implemented an RL architecture called an E.I.I.E., which consists of a CNN with a feedback loop.

The agent's performance was assessed using two datasets, a backtest dataset, and a test set dataset. The agent's portfolio was compared against other dummy portfolios, including the Markowitz portfolio, the 1/N evenly distributed weight portfolio, and the best-performing coin portfolio. The agent's portfolio was highly effective when no trading fees were applied, outperforming all other portfolios. However, when trading fees were included, the agent's portfolio struggled more but still managed to make impressive returns.

Overall, this work represents a significant step towards developing a practical and effective RL agent for cryptocurrency portfolio management. The implementation of a reinforcement learning agent for portfolio management in the cryptocurrency market shows promising results. The study highlights the potential of using reinforcement learning techniques for portfolio management and the importance of considering transaction fees when designing investment strategies. Further research and development could improve the agent's performance and lead to the creation of more effective investment strategies in the future.

To further research, we could try implementing the other architecture presented in the paper [2].

References

[1] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. arXiv: 1312.5602 [cs.LG].

[2] Zhengyao Jiang, Dixing Xu, and Jinjun Liang. *A Deep Reinforcement Learning Framework for the Financial Portfolio Management Problem*. 2017. arXiv: 1706.10059 [q-fin.CP].

[3] Timothy P. Lillicrap et al. *Continuous control with deep reinforcement learning*. 2019. arXiv: 1509.02971 [cs.LG].

[4] Coinbase. *Exchange Fees*. https://help.coinbase.com/en/exchange/trading-and-funding/exchange-fees. 2023.