Action Exploration in Portfolio Optimization with Reinforcement Learning

Caio de Souza Barbosa Costa[1], **Anna Helena Reali Costa**[1]

[1]Escola Politécnica – Universidade de São Paulo (USP)
São Paulo – SP – Brazil

`{caio326,anna.reali}@usp.br`

Abstract. *In portfolio optimization, an agent continuously rebalances the assets of a financial portfolio to maximize its long-term value. With advancements in artificial intelligence, several machine learning methods have been employed to develop agents capable of effectively managing portfolios. Among these, reinforcement learning agents have achieved significant success, particularly after the introduction of a specialized policy gradient algorithm that is currently the state-of-the-art training algorithm of the research field. However, the full-exploitation characteristic of the algorithm hinders the agent's exploration ability – an essential aspect of reinforcement learning – resulting in the generation of sub-optimal strategies that may even reduce the final portfolio value. To overcome this challenge, this paper explores the integration of noise functions to improve exploration in the agent's action space. Three distinct noise formulations adapted to the portfolio optimization task are evaluated through experiments in the Brazilian market. The results indicate that these noise-driven exploration strategies effectively mitigate the risk of sub-optimal policy generation and significantly improve overall portfolio performance.*

1. Introduction

Since the introduction of the modern portfolio theory [Markowitz 1952], several financial institutions apply mathematical and computational models to perform the portfolio optimization task [Gunjan and Bhattacharyya 2022]. In this task, the asset allocation within a financial portfolio is periodically adjusted in response to market conditions, which may include price trends, indicators, news, and other relevant data. This approach aims to maximize the expected return by removing human emotional biases from the decision-making process. By doing so, it enables the implementation of pragmatic, data-driven strategies that can more objectively navigate market volatility [Cartea et al. 2015].

Among the possible computational models, the ones based on reinforcement learning (RL) [Sutton and Barto 2018] are specially suitable to the portfolio optimization task [Felizardo et al. 2022]. In such approach, an agent interacts with an environment that mimics the dynamics of the financial market and the effects of the agent's investments. The interaction is periodic, so that the training process simulates the constant rebalancing of the portfolio over time, and the environment provides to the agent, at each time step, a reward value which is proportional to the obtained profit. Finally, RL algorithms maximize the expected return of the reward function and, consequently, the expected profit.

The RL formulation achieved great results specially after the introduction of a training algorithm in [Jiang et al. 2017]. This policy gradient algorithm has been well accepted in the research field for its ability to be used with modern neural network architectures such as convolutional ones [Shi et al. 2019], graph neural networks [Soleymani and Paquet 2021, Shi et al. 2022] and transformers [Xu et al. 2020]. Its success is related to the fact that it was specifically developed to train agents that optimize financial portfolios and, according to [Liang et al. 2018], it has shown to be faster, more efficient and achieve better results than other modern RL algorithms such as DDPG (*Deep Deterministic Policy Gradient*) [Lillicrap et al. 2015] and PPO (*Proximal Policy Optimization*) [Schulman et al. 2017]. However, the algorithm is heavily focused on exploitation, which limits the agent's ability to effectively explore alternative actions. As a result, it often gets stuck in local maxima during training, leading to suboptimal performance in volatile markets.

To address this issue, this paper explores the incorporation of random noise into the RL agent's policy to enhance exploration capabilities, similar to those introduced in the DDPG algorithm [Lillicrap et al. 2015]. Therefore, the contributions of this article are:

1. Introduction of three noise functions that can be applied to the agent's action and adheres to the constraints of the portfolio optimization problem;
2. Analysis of the effectiveness of the introduced noise functions in the training process and its results in the Brazilian financial market.

This paper is organized as follows. Section 2 introduces the mathematical definition of the portfolio optimization problem and Section 3 explains how reinforcement learning can be applied to solve it. Section 4 details the policy gradient algorithm introduced in [Jiang et al. 2017], while Section 5 points the main consequences of this algorithm, specially its full-exploitation weakness. Section 6 introduces a few action noise functions that can be applied in order to simulate exploration during training and Section 7 presents the experiments conducted in order to observe the effects of the noise-driven exploration in the training process. Finally, Section 8 concludes this paper.

2. Problem Definition

In the portfolio optimization task, an agent constantly rebalances a financial portfolio composed of n predefined assets by defining, at each time step t, a *portfolio vector* or *weights vector* $\vec{W_t}$, which contains the ratio of remaining cash in the portfolio ($\vec{W_{t,0}}$) and the rate of the invested value in each asset of the wallet ($\vec{W_{t,i}}$, in which $i \in \{1, 2, ..., n\}$). Therefore, $\vec{W_t} \in \mathbb{R}^{n+1}$ and the following constraints must be respected:

$$0 \leq \vec{W_{t,i}} \leq 1, \qquad \sum_{i=0}^{n} \vec{W_{t,i}} = 1. \tag{1}$$

At each time step t, there is a *price vector* $\vec{P_t}$ associated with the price of every asset in the portfolio vector. Thus, $\vec{P_t} \in \mathbb{R}^{n+1}$ and $\vec{P_{t,i}} > 0$, in which $i \in \{0, 1, ..., n\}$. Similar to the portfolio vector, the first element of the price vector refers to the remaining cash in the portfolio and, since, in this work, uninvested cash is considered risk-free and the prices of every other asset is calculated with relation to this one, $\vec{P_{t,0}} = 1$.

Due to the volatility of the market, the prices of the assets change and the weights vector $\vec{W_t^f}$ in the end of the time step t is probably different than the one in the beginning $\vec{W_t}$. This variation can be calculated through the equation:

$$\vec{W_t^f} = \frac{(\vec{P_t} \oslash \vec{P_{t-1}}) \odot \vec{W_t}}{(\vec{P_t} \oslash \vec{P_{t-1}}) \cdot \vec{W_t}}, \tag{2}$$

in which $\oslash$ is the element-wise division, $\odot$ is the element-wise multiplication and $\cdot$ denotes the dot product of vectors.

Finally, it is possible to calculate the value of the portfolio V_t based on its previous value with the equations:

$$V_t^f = V_t\Big(\vec{W_t} \cdot (\vec{P_t} \oslash \vec{P_{t-1}})\Big), \qquad\qquad V_{t+1} = \mu_{t+1} V_t^f, \tag{3}$$

in which V_t^f is the value of the portfolio at the end of time step t and μ_{t+1} is the *transaction remainder factor*, a value between 0 and 1 introduced in [Jiang et al. 2017] that simulates the application of transaction costs in the portfolio rebalancing from W_t^f to W_{t+1}.

In the beginning of the task, no money is invested ($\vec{W_0} = [1, 0, 0, ..., 0]$) and the objective is to find a sequence of portfolio vectors $[\vec{W_1}, \vec{W_2}, ..., \vec{W_T}]$ that maximizes the final value of the portfolio V_T^f.

3. Reinforcement Learning Approach

This section explains the reinforcement learning approach introduced by [Jiang et al. 2017] and used in this paper. Firstly, it is important to highlight that this formulation considers that there is no slippage in the market, so that trading orders issued are immediately completed, and that the actions of the agent do not impact the market. These considerations are very adherent to the real world when the portfolio is composed of high trading volume assets.

The reinforcement learning solution makes use of an agent which constantly interacts with the trading environment. In this paper, this environment is a simulation based on [Costa and Costa 2023] that implements the mathematical formulation presented in Section 2. The environment provides the agent observations O_t about the current state of the market and this observation is utilized by the agent so it can define an internal state S_t which is used as input in the agent's policy of actions π. The policy can be seen as a deterministic function $\pi : S \rightarrow A$ that maps a state S_t into an action A_t that is performed in the environment. Finally, after the action is performed, the environment simulates its effects in the value of the portfolio and provides the agent a new observation O_{t+1} and a reward associated with the effects of the performed action R_{t+1}.

In order to apply this approach in the portfolio optimization task, it is necessary to define the agent's state, action, reward function and policy of actions.

State: The state is composed of a three-dimensional tensor of shape (f, n, t), in which f is the number of features in the state-space, n is number of assets in the portfolio and t represents the size of the state's time window. In this work, the state is built by combining the 50-step time series of closing, high and low prices of each of the 10 assets used in the portfolio and, thus, its shape is $(3, 10, 50)$. All the time series are also normalized by dividing all its items by the last one, similar to the normalization applied in [Shi et al. 2019].

Action: The agent's action is the weights vector introduced in Section 2.

Reward: In this work, the reward provided by the environment is the logarithmic rate of return of the portfolio during a timestep t. This reward function can be calculated with the following equation:

$$r_t = ln\Big(\frac{V_t^f}{V_{t-1}^f}\Big), \tag{4}$$

in which V_t^f is the value of the portfolio at the end of the simulation step. Note that, by using this function, the environment's reward is positive when the agent actions generate profit and negative otherwise.

Policy: The policy of actions can be implemented as any function that maps a state into an action. However, due to the fact that both state and action spaces are continuous, a good approach is to make use of a neural network in order to apply deep reinforcement learning (DRL) algorithms [Sutton and Barto 2018]. Since the state of the agent is composed of multiple price temporal series, the EIIE (*ensemble of identical independent evaluator*) architecture was introduced by [Jiang et al. 2017] to process each time series independently but share the same parameters of a convolutional neural network. Additionally, the EIIE architecture is capable of dealing with transaction costs by making use of the last performed action in the decision making. Due to its simplicity and fast training time, this is the architecture used in this work.

4. Full-Exploitation Policy Gradient

According to [Liang et al. 2018], the best approach to train a portfolio optimization agent with reinforcement learning is to utilize the policy gradient algorithm introduced in [Jiang et al. 2017]. In this algorithm, the agent interacts with the environment and save its experiences (the current state, the action performed and the price variation) in a replay buffer ordered in time. After filling the replay buffer, the agent is trained for a number of *training steps* following the steps below:

1. The agent retrieves a sequential batch of experiences from the replay buffer;
2. The agent's actions related to the batch of data are determined;
3. The logarithm of the profit obtained in the batch of sequential data is calculated;
4. The parameters of the neural network are updated to maximize that profit value.
5. The replay buffer is updated.

It's important to highlight that batches with overlapping temporal data are considered different, so the batches with data from the intervals $[T_1, T_2]$ and $[T_1 + 1, T_2 + 1]$, for example, are equally valuable in the training process. Additionally, [Jiang et al. 2017] argues that newer experiences are more relevant to train the agent, therefore, the sequential batch is chosen following a geometric distribution [Ross 2014] that favors more recent data.

Algorithm 1 details the policy gradient algorithm. It is noticeable that the policy is defined as a neural network that receives the current agent state and the last performed action W_{t-1}. As in [Jiang et al. 2017], in order to provide W_{t-1} to the policy, a structure called *portfolio vector memory* saves all the actions performed by the agent ordered in time during an episode.

Algorithm 1 Policy gradient for portfolio optimization

```
1: Initializes policy neural network $\phi(\vec{s}, \vec{W}|\theta^{\phi})$.
2: Initializes replay buffer R.
3: Resets the environment and receives initial state $\vec{s_0}$.
4: for $step = 1$ until episode_size do                    ▷ Filling the replay buffer
5:     Defines action $\vec{W_t} = \phi(\vec{s_t}, \vec{W_{t-1}}|\theta^{\phi})$.
6:     Executes action in the environment and observes $r_t$ e $\vec{s_{t+1}}$.
7:     Saves experience ($\vec{s_t}$, $\vec{W_{t-1}}$, $\vec{P_t} \oslash \vec{P_{t-1}}$) in R.
8: end for
9: for $step = 1$ until num_steps do                       ▷ Training the agent
10:    Samples a sequential batch of data (from $T_1$ to $T_2$).
11:    Generates batched action $\vec{W_t} = \phi(\vec{s_t}, \vec{W_{t-1}}|\theta^{\phi})$.
12:    Calculates batched $\mu_t = 1 - c\sum_{i=1}^{N} |\vec{W_t}(i) - \vec{W_{t-1}}(i)|$, in which $c$ is the commission fee rate and $N$ is the size of portfolio vector.
13:    Updates the parameters of the policy using gradient ascent:
           $$\nabla_{\theta^{\phi}} J = \nabla_{\theta^{\phi}} \sum_{t=T_1}^{T_2} \frac{ln(\mu_t(\vec{W_t} \cdot (\vec{P_t} \oslash \vec{P_{t-1}}))}{T_2 - T_1 + 1}.$$
14:    Updates replay buffer.
15: end for
```

Finally, a benefit of this algorithm is that it can be easily used in *online training*, since new experiences can be saved in the replay buffer as the agent interacts with the market and a few training steps can be run between portfolio rebalances considering the new experiences so that the algorithm makes use of new market data to keep its policy of actions updated. In this work, this strategy is used when testing the agent in order to consider the effects of online learning in the final performance of the learned strategy.

5. Consequences of the Full-Exploitation Approach

In this section, we discuss the consequences of the approach introduced in Section 4.

5.1. Sub-optimal Policies

As it can be seen in algorithm 1, even though the policy gradient algorithm achieves higher performance than other approaches [Liang et al. 2018], it lacks one of the most important features of the reinforcement learning formulation: the exploration-exploitation trade-off [Sutton and Barto 2018]. In this trade-off, the agent must balance its decision-making during training in order to also explore the action space to avoid learning sub-optimal policies of actions. Since the policy gradient algorithm is full-exploitation, no exploration is performed and the agent does not consistently learns an optimal policy. This problem is specially present when the target market is volatile, such as the Brazilian one, and the agent can even learn trading strategies that makes him lose money.

To empirically show this problem, we have conducted an experiment in the Brazilian market [1]. 50 reinforcement learning agents are created in order to optimize a portfolio composed of 10 high-volume assets (VALE3, PETR4, ITUB4, BBDC4, BBAS3, RENT3, LREN3, PRIO3, WEGE3, ABEV3). The agents are trained using historical daily high, low and close prices time series[2] from 2011/01/01 to 2019/12/31 (2233 trading days) and they are tested in the period of 2020/01/01 to 2020/12/31 (248 trading days). Note that a very volatile interval of the market was chosen as test period (in 2020, the beginning of the COVID-19 pandemic has considerably hit the Brazilian economy) in order to highlight the weakness of the training algorithm. Table 1 contains the hyper-parameters used to train the agents.

Table 1. Hyper-parameters used in the training procedure.

Hyper-parameter	**Value**	**Description**
Learning rate	0.00005	The step size used in the gradient ascent.
Batch size	200	Size of the sequence of experiences used in the optimization.
Sample bias	0.002	Probability considered in the geometric distribution used when sampling the batches of experiences.
Number of training steps	300000	Number of training steps used to train the agent.
Time window size	50	Size (in steps) of the time window considered in the state representation.
Comission rate	0.0025	Rate of comission fee applied when rebalancing the portfolio.
Initial portfolio value	100000	Initial value of the portfolio in BRL.

Table 2 demonstrates the results of the experiment. The FAPV is a metric of profit and can be calculated by dividing the final value of the portfolio by its initial value. Therefore, an FAPV smaller than 1.0 indicates that the portfolio lost value, which denotes that the optimization process failed. As it can be seen in the results of the experiment, 34% of the runs can be categorized as a failed portfolio optimization, showing that the training algorithm is not reliable. Additionaly, only 1 run was able to achieve FAPV bigger than 2.0, which indicates that the algorithm does not consistently learns the optimal policy.

Table 2. Results of 50 runs with the full-exploitation algorithm.

	Value
Mean FAPV of runs	1.15
Runs with FAPV bigger than 1.0	33
Runs with FAPV bigger than 1.5	5
Runs with FAPV bigger than 2.0	1

5.2. Heterogeneous Portfolio Vectors

Another characteristic of this algorithm is the fact that after several training steps, the agent tends to learn policies that generate very heterogeneous portfolio vectors, i.e. vec-

[1] Repository with code and data: `https://github.com/CaioSBC/noise_portfolio`.
[2] The data was retrieved from `https://finance.yahoo.com/`.

tors allocating the majority of the portfolio value in one asset. This issue was first described in [Shi et al. 2022] and a common workaround is to use a softmax with temperature layer [Hinton et al. 2015] in the output of the policy neural network.

Empirically, this behavior can be observed with a simple experiment in which a training process in the Brazilian market with the same data and hyper-parameters of the experiment in Section 5.1 is conducted, excluding the number of training steps, which was limited to 65000 in order to simplify the process. At each training step, a heterogeneity index is calculated using the batch of generated actions in the training step through the following equation:

$$heterogeneity = \frac{\sum_{t=T_1}^{T_2} \Big(max(\vec{W_t}) - min(\vec{W_t}) \Big)}{T_2 - T_1 + 1}, \tag{5}$$

in which $max(\vec{W_t})$ and $min(\vec{W_t})$ are, respectively, the maximum and minimum values of the portfolio actions defined by the policy in the batch of data from simulation steps T_1 to T_2. Therefore, a batch of completely homogeneous $\vec{W}$, such as $[0.25, 0.25, 0.25, 0.25]$, contains a heterogeneity index equal to zero, and a batch of completely heterogeneous $\vec{W}$, such as $[0, 1, 0, 0]$, has an index equal to one.

Figure 1 shows the heterogeneity index at each training step. The blue transparent and noisy time series denotes the real heterogeneity index calculated using equation 5 and the smoothed opaque curve is a moving average of the transparent one, highlighting the asymptotic nature of the index. Analyzing the graph, it is noticeable that, after training using algorithm 1, the agent tends to perform heterogeneous actions.

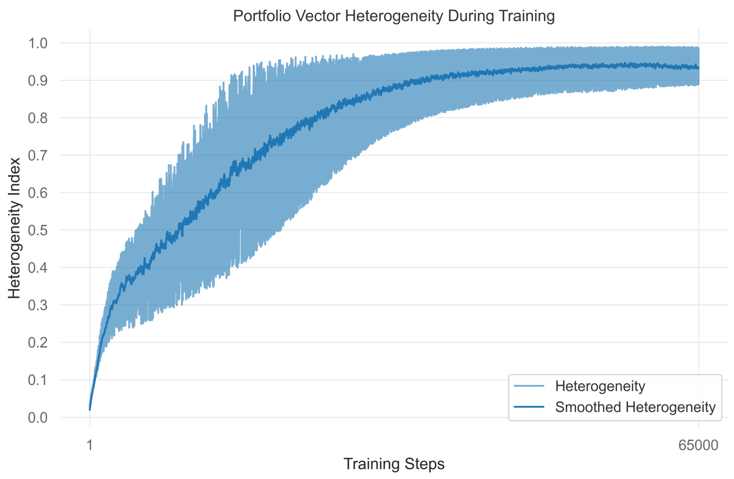

Figure 1. Heterogeneity index during the training process. The smoothed curve is generated by a moving average of 100 past values.

6. Exploration with Action Noise

A possible solution to mitigate the issue introduced in Section 5.1 is to apply a noise function $r : \vec{W} \rightarrow \vec{W}$ to the action generated by the agent's policy in order to add a stochastic step to the training process. The core idea is that given an input state $\vec{s_t}$, the agent's performed action $\vec{W}_t' = r(\vec{W}_t)$ is slightly different than the agent's intended action $\vec{W}_t = \phi(\vec{s_t}, \vec{W}_{t-1})$ so that different gradients are calculated in the optimization process and the policy neural network is able to avoid local maxima and overfitting.

It is important to highlight that this idea is highly inspired by the action noise introduced in the DDPG algorithm [Lillicrap et al. 2015], so it considers that, for a given state $\vec{s_t}$ and last portfolio vector $\vec{W}_{t-1}$, close actions performed leads to close outcomes. This hypothesis make sense in our use-case since similar portfolio strategies generates similar profit. Additionally, one should note that the random function $r : \vec{W} \to \vec{W}$ must generate an action which complies with the constraints specified in equation 1. In this section, we introduce a few random functions and discuss their applicability.

6.1. Constant logarithmic noise

The first possible noise function simply applies a noise that is a sample of a normal distribution with mean 0 and standard deviation proportional to a hyper-parameter ε. The bigger the value of ε, the bigger the standard deviation of the normal distribution, meaning that a bigger random noise is applied to the action. Finally, in order to adhere to the constraints in equation 1, the noise is added in the logarithmic domain and a *softmax* function is applied. The equations of the constant logarithmic noise are expressed below:

$$\Delta\vec{W}_t \sim \mathcal{N}\left(0, \left(\varepsilon|ln(10^{-7})|\right)^2\right), \qquad \vec{W}_t' = softmax\left(ln(\vec{W}_t) + \Delta\vec{W}_t\right). \tag{6}$$

6.2. Variable Logarithmic Noise

This is a variation of the noise function introduced in Section 6.1. The main distinction between the two is the fact that the variable logarithmic noise makes use of a normal distribution whose standard deviation is dependent on the maximum value of the absolute value of the logarithm of $\vec{W}_t$. Equation 7 shows how to calculate this noise function.

$$\begin{cases} \Delta\vec{W}_t \sim \mathcal{N}\left(0, \left(\varepsilon \max_w \left(ln(\vec{W}_t)^{abs}\right)\right)^2\right), \text{ in which } \vec{X}^{abs} := \left(|x_{i,j}|\right)_{(i,j)}, \\ \vec{W}_t' = softmax\left(ln(\vec{W}_t) + \Delta\vec{W}_t\right). \end{cases} \tag{7}$$

The difference between both approaches is that the constant variable noise is also effective in homogeneous actions while the variable logarithmic noise only affects portfolio vectors who are partially heterogeneous. Figure 2 demonstrates this difference, showing that the constant logarithmic noise modifies a homogeneous portfolio vector more effectively than the variable one. It is also noticeable that the peak of effectiveness of both noise functions happens when the heterogeneity index is near 0.5. Considering the behavior outlined in Section 5.2, it means that both functions apply more noise when the agent is in the middle of the learning process and, thus, starting the policy neural network convergence.

6.3. Dirichlet Noise

This noise function takes advantage of the properties of the Dirichlet distribution [Gelman et al. 2015], which generates samples of vectors that adheres to the constraints

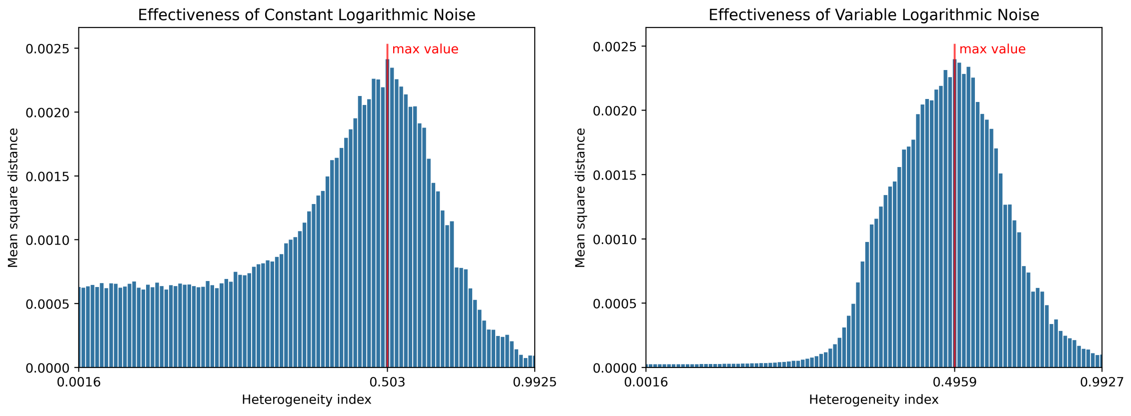

Figure 2. Comparison between constant and variable logarithmic noise effectiveness on portfolio vectors with different heterogeneity indexes ($\varepsilon = 0.1$).

introduced in equation 1. Usually, this distribution is parameterized by a vector with different parameters, but in this work, we consider that all the parameters in the distribution input vector are equal to $\alpha \in \mathbb{R}^+$. The implication of this approach is that, the bigger the α parameter, the more homogeneous the distribution samples are, and vice-versa.

Equation 8 shows how to generate a noisy portfolio vector using the Dirichlet noise and the effect of the α parameter can be seen in figure 3: the smaller the parameter, the more effective the noise function is to modify homogeneous portfolio vectors. Finally, considering the consequence of the full-exploitation training algorithm discussed in Section 5.2, it is noticeable that this noise function is dominant in the end of the training process, since its peak of effectiveness is related to high heterogeneity indexes.

$$\vec{W_t}^{rand} \sim Dir(\alpha), \qquad \vec{W_t'} = (1-\varepsilon)\vec{W_t} + \varepsilon\vec{W_t}^{rand}. \tag{8}$$

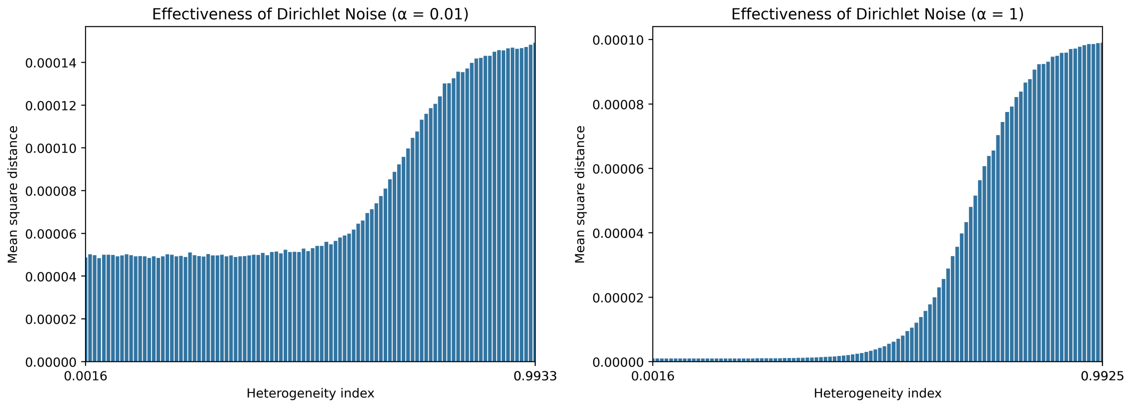

Figure 3. Effectiveness of different values of α in the Dirichlet noise function applied to vectors of different heterogeneity indexes ($\varepsilon = 0.1$).

7. Effects of the Action Noise

In order to verify if the effects of the noise functions introduced in Section 6, an experiment[3] is conducted: for each noise function and different values of ε (0, 0.001, 0.005,

[3]Repository with code and data: `https://github.com/CaioSBC/noise_portfolio`.

0.01, 0.05, 0.1, 0.25), 50 reinforcement learning agents are trained to optimize a financial portfolio identical to the one in the experiment conducted in Section 5.1 (VALE3, PETR4, ITUB4, BBDC4, BBAS3, RENT3,LREN3, PRIO3, WEGE3, ABEV3) using the same hyper-parameters introduced in table 1. The agents are also trained using the same datasets used in that experiment.

Tables 3, 4 and 5 present the results of the experiments conducted using constant logarithmic noise, variable logarithmic noise and Dirichlet noise, respectively. Notably, all noise functions yield better performance compared to experiments without noise ($\varepsilon = 0$), with improvements in mean FAPV observed in several results. However, it is evident that the logarithmic noises produce more consistent results since it is possible to identify a trend: bigger ε values usually lead to gains in performance. The Dirichlet noise, on the other hand does not present an evident pattern, which might indicate that finding the optimal value for the ε parameter can be difficult when using that noise function.

Table 3. Results of 50 runs with the constant logarithmic noise.

Epsilon	**Mean FAPV of runs**	**Runs with FAPV bigger than 1.0**	**Runs with FAPV bigger than 1.5**	**Runs with FAPV bigger than 2.0**
0	1.15 ± 0.05	33	5	1
0.001	1.25 ± 0.06	36	10	3
0.005	1.24 ± 0.07	35	9	3
0.01	1.35 ± 0.07	41	16	4
0.05	1.27 ± 0.06	37	10	4
0.1	1.27 ± 0.06	41	8	4
0.25	1.43 ± 0.06	45	19	5

Table 4. Results of 50 runs with the variable logarithmic noise.

Epsilon	**Mean FAPV of runs**	**Runs with FAPV bigger than 1.0**	**Runs with FAPV bigger than 1.5**	**Runs with FAPV bigger than 2.0**
0	1.15 ± 0.05	33	5	1
0.001	1.33 ± 0.07	40	12	4
0.005	1.23 ± 0.05	35	14	1
0.01	1.22 ± 0.05	36	10	4
0.05	1.31 ± 0.07	39	12	3
0.1	1.37 ± 0.07	43	16	4
0.25	1.47 ± 0.07	46	17	7

The application of action noise also demonstrates to be effective in mitigating the sub-optimal policies issue presented in Section 5.1, specially when using higher ε values in the logarithmic approaches. The fact that the action noise exploration could reduce the number of simulations in which the agent loses portfolio value from 34% to 8% (in the best-case scenario) indicates that this type of exploration can be effectively used to improve the performance of the policy gradient algorithm.

Finally, since the performance gap between the constant and variable logarithmic approach is minimal and the main difference between these noise functions is that the former applies more noise to homogeneous portfolio vectors and considering the asymptotic

Table 5. Results of 50 runs with the Dirichlet noise ($\alpha = 0.01$).

Epsilon	Mean FAPV of runs	Runs with FAPV bigger than 1.0	Runs with FAPV bigger than 1.5	Runs with FAPV bigger than 2.0
0	1.15 ± 0.05	33	5	1
0.001	1.29 ± 0.06	39	10	5
0.005	1.22 ± 0.05	40	11	1
0.01	1.41 ± 0.08	45	14	8
0.05	1.20 ± 0.05	37	5	1
0.1	1.21 ± 0.05	40	10	1
0.25	1.19 ± 0.05	35	8	3

behavior described in Section 5.2, the results indicate that the application of noise effects in the beginning of the training process is irrelevant. Additionally, because the Dirichlet noise is more effective in heterogeneous portfolio vectors, its inconsistent results suggests that noisy actions do not seem to improve the training process when applied in the end of it. Therefore, the experiments of this study concludes that the best approach to implement noise-driven exploration to the algorithm introduced in Section 1 is to apply noise functions in the middle of the policy neural network convergence.

8. Conclusions

The policy gradient algorithm introduced in [Jiang et al. 2017] is currently the state-of-the-art training method for reinforcement learning agents optimizing financial portfolios. However, its full-exploitation strategy compromises one of the core strengths of reinforcement learning: the agent's ability to explore and develop a generalizing policy. To address this limitation, this study conducts extensive tests with this algorithm and empirically demonstrates that it tends to generate sub-optimal action policies, which can even lead to a decline in portfolio value.

This paper then explores the idea of simulating agent exploration by applying a noise function to the actions generated by the agent's policy, similar to the approach in [Lillicrap et al. 2015]. Three action noise formulations that adhere to the constraints of the portfolio optimization action space are introduced and their performances are evaluated with experiments in the Brazilian market. The results demonstrate that this exploration approach effectively mitigate the problem of sub-optimal policies and improves the performance of the portfolio.

Finally, this article concludes that the most effective noise functions are those that introduce larger noise during the middle phase of the neural network's convergence. Therefore, for this specific algorithm, adding noise at the beginning and/or end of the training process should be avoided.

Acknowledgments

The authors would like to thank the *Programa de Bolsas Itaú* (PBI) of the *Centro de Ciência de Dados* (C^{2}D) at Escola Politécnica at USP, supported by *Itaú Unibanco S.A.*, and the Brazilian National Council for Scientific and Technological Development (CNPq Grant N. 312360/2023-1).

References

Cartea, Á., Jaimungal, S., and Penalva, J. (2015). *Algorithmic and High-Frequency Trading*. Cambridge University Press.

Costa, C. d. S. B. and Costa, A. H. R. (2023). POE: A General Portfolio Optimization Environment for FinRL. In *Anais Do Brazilian Workshop on Artificial Intelligence in Finance (BWAIF)*, pages 132–143. SBC.

Felizardo, L. K., Paiva, F. C. L., Costa, A. H. R., and Del-Moral-Hernandez, E. (2022). Reinforcement Learning Applied to Trading Systems: A Survey.

Gelman, A., Carlin, J. B., Stern, H. S., Dunson, D. B., Vehtari, A., and Rubin, D. B. (2015). *Bayesian Data Analysis*. Chapman and Hall/CRC, New York, 3 edition.

Gunjan, A. and Bhattacharyya, S. (2022). A brief review of portfolio optimization techniques. *Artificial Intelligence Review*.

Hinton, G., Vinyals, O., and Dean, J. (2015). Distilling the Knowledge in a Neural Network.

Jiang, Z., Xu, D., and Liang, J. (2017). A Deep Reinforcement Learning Framework for the Financial Portfolio Management Problem.

Liang, Z., Chen, H., Zhu, J., Jiang, K., and Li, Y. (2018). Adversarial Deep Reinforcement Learning in Portfolio Management.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2015). Continuous control with deep reinforcement learning.

Markowitz, H. (1952). Portfolio Selection. *The Journal of Finance*, 7(1):77–91.

Ross, S. (2014). Introduction to Probability Models. In *Introduction to Probability Models (Eleventh Edition)*, page iii. Academic Press, Boston.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal Policy Optimization Algorithms.

Shi, S., Li, J., Li, G., and Pan, P. (2019). A Multi-Scale Temporal Feature Aggregation Convolutional Neural Network for Portfolio Management. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pages 1613–1622, Beijing China. ACM.

Shi, S., Li, J., Li, G., Pan, P., Chen, Q., and Sun, Q. (2022). GPM: A graph convolutional network based reinforcement learning framework for portfolio management. *Neurocomputing*, 498:14–27.

Soleymani, F. and Paquet, E. (2021). Deep graph convolutional reinforcement learning for financial portfolio management – DeepPocket. *Expert Systems with Applications*, 182:115127.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA.

Xu, K., Zhang, Y., Ye, D., Zhao, P., and Tan, M. (2020). Relation-Aware Transformer for Portfolio Policy Learning. In *Twenty-Ninth International Joint Conference on Artificial Intelligence*, volume 5, pages 4647–4653.