SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master’s Thesis in Data Engineering and Analytics

A Generalized Framework for Applications of DDPG in Portfolio Optimization

Ramsundar Govindarajan

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master’s Thesis in Data Engineering and Analytics

A Generalized Framework for Applications of DDPG in Portfolio Optimization

Ein verallgemeinerter Rahmen für Anwendungen von DDPG in der Portfoliooptimierung

Author:	Ramsundar Govindarajan
Supervisor:	Prof. Dr. Rudi Zagst, Prof. Dr. Martin Bichler
Advisor:	Michel Kschonnek
Submission Date:	April 15, 2023

I confirm that this master's thesis in data engineering and analytics is my own work and I have documented all sources and material used.

Munich, April 15, 2023 Ramsundar Govindarajan

Acknowledgments

I would like to express my sincere gratitude to Prof. Dr. Rudi Zagst from the Chair of Financial Mathematics and Prof. Dr. Martin Bichler from the Chair of Decision Sciences and Systems for giving me the chance to complete my thesis in the inter-disciplinary field of informatics and finance.

I am also extremely thankful to Michel Kschonnek for giving me constant support and providing me with innovative ideas. Your valuable feedback and engaging discussions were instrumental in the completion of this thesis.

Finally I would also like to extend my thanks to AllianzGI for making this project possible.

Abstract

We begin by stating the portfolio optimization problem and some of the traditional approaches used for solving it. We also mention a previous work in Reinforcement Learning (RL) and state our own project outline on how we are going to expand the work. We then start off with a short primer on RL and in particular look at an "Actor-Critic" technique called Deep Determinisitic Policy Gradients (DDPG), which can be used to maximize rewards in a continuous setting. We state our experimental set up and also discuss the different critic and the actor functions, we can employ in our set up. Afterwards, we then present the first version of our proposed algorithm, DDPGFunctions. We provide a brief commentary of the system architecture of our framework, the modular components, hyper tuning framework, and the tools for tracking experiments and graphing. Understanding some potential problems with the vanilla approach we try to speed up and improve the accuracy of our results and performance in subsequent versions of DDPGFunctions - DDPGShockBuffer and DDPGEstimates. We compare our results of all these approaches against a variety of environment settings and hyper parameter configurations. In the final part of the thesis, we consider certain aspects for future work and discuss some limitations with our current approach.

Contents

1 Portfolio Optimization Problem

In this chapter we begin by stating the Portfolio optimization problem. We then discuss some of the ways the portfolio optimization problem had been solved traditionally. We discuss some of the problems with respect to these approaches. We also discuss in brief some of the work done by our group in previous works. Finally we outline our approach to solve this class of problems.

1.1 Definition of the portfolio Optimization Problem

We recap in this section the portfolio optimization problem as stated in an earlier work [1]. Consider a finite time horizon $T > 0$ and a complete, filtered probability space $(\Omega, \mathcal{F}_T, \mathbb{F} = (\mathcal{F}_t)_{t\in[0,T]}, Q)$ where the filtration $\mathbb{F}$ is generated by a Wiener process [2] $W = (W(t))_{t\in[0,T]}$.

Consider a market model, $\mathcal{M}$ in which we have 2 assets, a risky asset which follows the Stochastic Differential Equation (SDE)

$$dP_1(t) = P_1(t)(\mu dt + \sigma dW(t)) \tag{1.1}$$

and a riskless asset which follows the SDE

$$dP_0(t) = P_0(t)(r_c\ dt). \tag{1.2}$$

Here, $W(t)$ is the Wiener process we described earlier and the parameters μ,σ and r_c are positive constants.

Let us assume that an agent in the market model specified by $\mathcal{M}$, trades in these 2 assets. This means the agent needs to choose a portfolio process, $\pi : [0, T] \times (0, \infty) \to \mathbb{R}$, which maps the current time and the agent's current wealth (t, v) to the proportion of the agent's wealth invested in the risky asset. We can state then that the evolution of the agent's wealth,$V^{v_0,\pi}$, in $\mathcal{M}$ satisfies the SDE

$$\begin{aligned} dV^{v_0,\pi}(t) &= V^{v_0,\pi}(t)([r_c + (\mu - r_c)\pi(t, V^{v_0,\pi}(t))]dt + \pi(t, V^{v_0,\pi}(t))\sigma dW(t)) \\ V^{v_o,\pi}(0) &= v_0. \end{aligned} \tag{1.3}$$

Due to the Markovity of the setting, we assume that this relative portfolio process is a function of the current time and wealth $(t, V^{v_0,\pi}(t))$. All π satisfying some additional measurability and integrability conditions (see equation (2.3) in [3] for details) will be called admissible and are collected in the set Λ.

We can find a semi-explicit solution for the SDE stated in (1.3) as

$$V^{v_0,\pi}(t) = v_0 \exp\left(\int_0^t r_c + (\mu - r_c)\pi(s, V^{v_0,\pi}(s)) - \frac{1}{2}(\sigma\pi(s, V^{v_0,\pi}(s)))^2 ds + \int_0^t \pi(s, V^{v_0,\pi}(s))\sigma dW(s)\right) \tag{1.4}$$

Further, we assume that the agent is risk averse and derives utility from his terminal wealth at maturity T. Assume the utility function U is concave and an increasing function such that $U : [0, \infty) \to \mathbb{R} \cup \{-\infty\}$. The portfolio optimization problem is then defined as

$$(\mathbf{P}) \begin{cases} \Phi(v_0) = \underset{\pi \in \Lambda}{sup} \, \mathbb{E}[U(V^{v_0,\pi}(T))] \end{cases} \tag{1.5}$$

And the time-dependent version of the problem can be defined as

$$(\mathbf{P_t}) \begin{cases} \Phi(t, v) = \underset{\pi \in \Lambda}{sup} \, \mathbb{E}[U(V^{v_0,\pi}(T)) | V^{v_0,\pi}(t) = v] \end{cases} \tag{1.6}$$

There are quite a few approaches to solving this original portfolio optimization problem. We are listing some of the methods below.

1.1.1 Martingale Method

The portfolio optimization problem can be decomposed into a static optimization problem and a representation problem. The former involves determining the optimal terminal wealth, while the latter involves determining the corresponding optimal trading strategy. This approach, known as the martingale approach, is well-established in complete underlying financial market models, where it is supported by the Martingale Representation Theorem. To cite specific sources, refer to Chapter 3.4 in [4] or Chapter 7.2 in [5]. However, for incomplete financial market models, such as the Heston model (see [17]), the martingale approach is not directly applicable. To address this, the financial market model can be artificially completed by adding a volatility-dependent derivative written on the risky asset, as discussed in [6], [7], [8], and [9].

1.1.2 Merton's Method

Merton's method of portfolio optimization [10] regards the portfolio problem as a stochastic control problem, where the investor aims to maximize expected utility from consumption and terminal wealth. The method obtains the optimal value function, $\Phi(t, v)$ by solving a partial differential equation called the Hamilton-Jacobi-Bellman (HJB) equation [11] [12]. The HJB equation is derived based on the principle of dynamic programming.

To solve the HJB equation, Merton's method requires an ansatz for the structure of the value function. Once the value function is obtained, the method derives the optimal trading strategy of the investor, which is a function of the portfolio's state variables - time t and wealth v.

1.1.3 Challenges of these approaches

The martingale approach, can be applied to characterize the optimal terminal wealth for a portfolio for many general utility functions and complete models (i.e the static problem is solvable under general assumptions) However obtaining the corresponding trading strategy (i.e solving the representation problem) is substantially more challenging in general settings.

Merton’s method has the advantage of being able to handle incomplete financial models such as those with jump-diffusion processes and stochastic interest rates, but has the drawback of requiring the solution to the associated Hamilton–Jacobi–Bellman PDE. This can be challenging and time consuming especially in more complex settings, where there are no analytical solutions for this PDE.

1.2 Reinforcement Learning for Portfolio Optimization and Trading Strategies

1.2.1 State of the Art

After successes in solving games such as Atari [13], Go, etc and also in its generic applicability to a variety of fields such as Natural Language Processing (NLP) (GPT models) [14], Reinforcement Learning (RL) is gaining considerable attention for its applications in finance. Several works have explored the potential of reinforcement learning techniques for financial portfolio management and trading strategies. We briefly look at some of them here.

One such work is "A deep Reinforcement learning framework for the financial portfolio management problem" by Ying et al. (2017) [15] which presents a deep reinforcement learning framework for portfolio management. They used the Deep Deterministic Policy Gradient (DDPG) algorithm, and their results showed that the proposed framework can learn effective portfolio management strategies in terms of metrics such as Sharpe ratio, portfolio returns and maximum draw down.

Another work in this field is "Application of deep reinforcement learning in stock trading strategies and stock forecasting" by Zhang et al. (2020) [16]. They proposed a deep reinforcement learning-based trading strategy that integrates technical analysis and fundamental analysis for stock trading. Their experimental results demonstrated that the proposed model can achieve better performance than other trading models in terms of the metrics such as Sharpe ratio, portfolio returns and maximum draw down.

In "Market Making via Reinforcement Learning" by Huang et al. (2018),[17] the authors presented a market making algorithm that uses reinforcement learning to learn an optimal trading strategy. The algorithm is trained on a limit order book simulator and evaluated on a real-world dataset. The results showed that the algorithm can learn an effective market making strategy in terms of metrics such as effective bid-ask spread, participation rate, slippage ratio, and other portfolio metrics such as trading volume, Sharpe ratio, portfolio returns and maximum draw down etc..

Finally, "Deep Hedging" by Buehler et al. (2019) [18] proposed a new method for hedging options in continuous time using deep learning. They demonstrated that the deep hedging approach can achieve better hedging performance than traditional methods.

1.2.2 Advantages of Reinforcement Learning Techniques

Some of the key areas where reinforcement learning techniques would fare better than traditional ways such as Merton's method and the martingale method described above are listed.

- **Flexibility**: Reinforcement learning techniques can be applied to a wider range of optimization problems, without requiring assumptions on the utility functions such as concavity or differentiability.
- **Model-free**: Reinforcement learning techniques do not require an explicit model of the market dynamics or assumptions about the distribution of returns. This can be useful when the market model is complex and lacks analytical tractability, but can be used to sample market data efficiently.

1.3 Previous Work and Project Outline

Inspired by these advantages, previous projects in our group [19] and [20], worked on DDPG based approaches to solve this portfolio optimization problem using large neural networks to represent the "unknown" critic and actor. In that work, they obtained estimates for the Q-value function and the optimal allocation for logarithmic and power utility functions, which recovered the main characteristics of the true Q-value function and the true optimal allocation.

However, despite achieving a reasonable estimation for the optimal allocation with both utility functions, a clear asymptotic convergence to the true optimal allocation could not be observed. Furthermore, the previous work used Tensorflow's [21] implementation of DDPG and certain key parameters such as the magnitude of target-updates decay and (mini-) batch sizes increase during DDPG's run-time could not be configured. The algorithm's run time also exploded when the number of time steps in the time discretization increased.

1.3.1 Project outline

Considering the drawbacks from the previous projects, we identified the following areas of improvement, which constitute the main goals in our project.

- **Specifying different function classes instead of neural networks for both Q-value and a-value function:** The motivation for moving away from an agnostic function and coming back to giving a specific structure to the functions (similar to the ansatz in Merton's method) is to incorporate known structural properties about the optimization problem into the reinforcement learning algorithm.

- **Build modular components:** To address the configurability problems with the previous work, we want to develop a modularized testing framework that can be utilized for different portfolio optimization problems by applying DDPG while experimenting with various specifications, such as and not limited to:
  - Utility function (reward)
  - Financial market model (environment)
  - Parametrization of actor-function in DDPG
  - Parametrization of critic-function in DDPG
- **Tuning hyper-parameters of DDPG:** The framework should include a stand-alone implementation of DDPG so that changes to DDPG itself can be part of the modularization process.
- **Capabilities to analyze convergence speed and accuracy**
- We aim to provide **detailed documentation** for the modularized testing framework so that this can be extended in future projects.

Overall, this thesis aims to develop a flexible and modular testing framework for portfolio optimization that can be adapted to various optimization problems. The performance of the DDPG algorithm in portfolio optimization will be tested under different utility functions and market models, and the results will be analyzed and compared to evaluate its accuracy and convergence speed. The framework will be fully documented for the benefit of future users.

2 Reinforcement learning and DDPG

In this chapter we begin by giving a very short introduction of reinforcement learning (RL) and provide a taxonomy of the various flavors of RL. We then drill down to a specific branch in RL (model free, Q-Learning) that we plan to deploy for the portfolio optmization problem. We discuss these classes briefly before we move to explain the Deep Determinisitic Policy Gradients (DDPG) algorithm - the foundation for our solution. In the final part, we explain the different components of DDPG.

2.1 Introduction to Reinforcement Learning

Reinforcement learning (RL) is a field of machine learning that studies decision making in an environment, where an agent interacts with its surroundings to achieve a goal. The agent receives rewards based on the state it is in and the actions it takes. The aim is to maximize the cumulative reward over time by learning a function that maps optimal states to actions. The entire lifecycle is shown in Figure 2.1 [22].

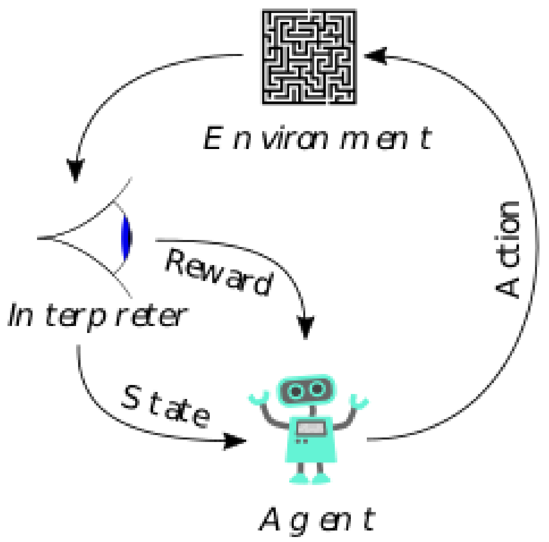


Figure 2.1: Reinforcement learning

There are many flavors of RL algorithms, each targeted at specific types of problems. A representative taxonomy of RL algorithms is provided in Figure 2.2.

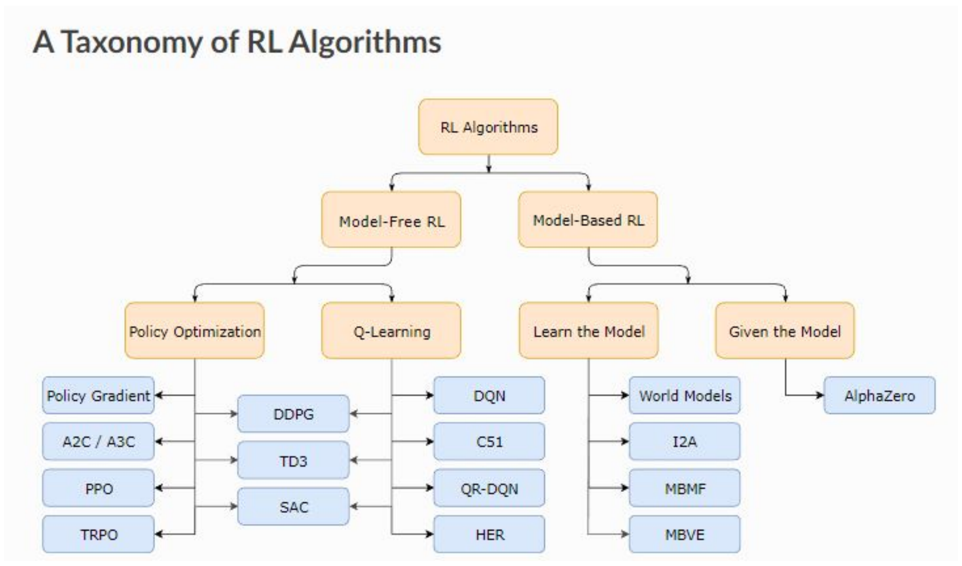


Figure 2.2: Taxanomy of RL algorithms

2.1.1 Model-Free vs Model-Based RL

As can be seen in Figure 2.2, a major classification in RL algorithms is related to the model of the environment, depending on whether or not, the model is known to the agent a priori. By model, we mean the state transition functions on taking any permissible action in the environment and also the rewards the agent obtains by taking up such an action [23].

When the model is known to the agent, the problem is easier to solve as the agent can explicitly decide what action to take depending on the possible rewards on taking up that action. In-sample accuracy improves and training time reduces substantially when the environment can be modeled [24].

However in many settings, the ground truth model of the environment is not available to the agent. In such cases, the agent can try to build a model based on the experience of the agent. However such models are biased towards the training sample and quite often such agents perform terribly in real-life scenarios [25].

Model-free methods on the other hand are algorithms that do not use a model of the environment, meaning they do not have a function that predicts state transitions and rewards. Instead, these methods rely solely on the experience of the agent through trial-and-error interactions with the environment. Model-free methods are easier to implement and tune compared to model-based methods, which require the agent to learn a model of the environment. Some popular model-free RL algorithms include Q-Learning, SARSA, and actor-critic networks. However, model-free methods are less sample efficient compared to model-based methods because they do not allow the agent to plan ahead. But generally they are more robust in handling real life scenarios.

2.2 Q-Learning

One of the most popular methods for solving RL problems is Q-Learning [26], which is a model-free, off-policy algorithm. The basic idea behind Q-Learning is to estimate the action-value function $Q(s, a)$, also refered to as "critic", which represents the expected future reward starting from state s, taking action a and acting optimally afterwards. The optimal action $a^*(s)$ is then computed as

$$a^*(s) = \arg\max_{a \in A} Q(s, a), \tag{2.1}$$

where A is the set of admissible actions.

We start off with an estimate of the Q function and then iteratively update the estimate by using the Bellman optimality equation [27]

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r(s, a, s') + \max_{a' \in A} Q(s', a') - Q(s, a) \right], \tag{2.2}$$

where α is the learning rate, $r(s, a, s')$ is the reward obtained after taking action a in state s and s' is the next state.

Q-Learning has some favorable properties [23] such as:

- **Model-Free**: One of the key advantages of Q-Learning is that it is model free. So it is very flexible and can be applied to solve a wide range of problems.
- **Off-Policy**: Q-Learning is an off-policy learner. An off-policy learner learns the value of the optimal action independently of the agent's actions. This leads to more stable results, especially when the scenarios change rapidly over time.
- **Easy Implementation**: Q-Learning is a relatively simple algorithm to implement, making it a good starting point for understanding reinforcement learning.
- **Good Exploration-Exploitation Trade-off**: Q-Learning balances exploration and exploitation by updating its estimates based on observed rewards. By tuning the learning rate, we can configure the exploration/exploitation trade-off at different times as we proceed into the experiment.

Q-Learning has been applied to a wide range of problems, including robot control, recommendation systems, game playing, and finance. The algorithm has been shown to be effective in many applications, but it can suffer from slow convergence and instability when applied to problems with large state spaces and sparse rewards.

One of the challenges in implementing Q-Learning is dealing with the fact that the Q-value function can have high variance. To mitigate this issue, the algorithm can be combined with function approximation techniques, such as neural networks, to approximate the value function.

2.2.1 Actor - Critic Algorithm

A variation of Q-Learning is the actor-critic algorithm [28], which uses separate actor, a^ϕ and critic, Q^θ with parameters ϕ and θ to estimate the optimal action and the Q-value function, respectively. In other words the central underlying assumption is that

$$Q(s,a) = Q^\theta(s,a) \text{ and } a^*(s) = a^\phi(s),$$

for suitable parameterized functions Q^θ and a^ϕ with parameters θ and ϕ. The actor maps states to actions, while the critic maps states and actions to a scalar value, representing the expected reward. The actor is updated based on the gradients of the Q-value function with respect to the actions, while the critic is updated using the standard Q-Learning rule.

2.3 DDPG Basics

Deep Deterministic Policy Gradient (DDPG) is an algorithm which concurrently learns a Q-value function and an optimal action function. It uses off-policy data and the Bellman equation to learn the Q-function, and uses the Q-function to learn the policy. DDPG can only be used for environments with continuous action spaces and can be thought of as protoypical deep Q-Learning for continuous action spaces.

The DDPG approach is closely connected to Q-Learning, and is motivated by the same relation between actor a^* and the critic Q

$$a^*(s) = \underset{a \in A}{\text{argmax}}\, Q(s,a). \tag{2.3}$$

DDPG combines the process of learning an estimate for $Q(s,a)$ and an estimate for $a(s)$ in every update step and is tailored specifically for environments with continuous action spaces. This specialization of DDPG lies in how it calculates the maximum value of $Q(s,a)$ over all actions a, i.e $\max_{a \in A} Q(s,a)$.

Finding the maximum Q-value in a scenario with a finite number of discrete actions is straightforward because we can easily evaluate the Q-values for each action and compare them, giving us the optimal action right away. However, when the action space is continuous, it's impossible to evaluate all possible actions, making the optimization problem much more complex. Using a traditional optimization algorithm to compute $\max_{a \in A} Q(s,a)$ would be computationally demanding and would have to be executed every time the agent needs to take an action, which is not practical.

Since the action space is continuous, it is assumed that the function $Q^\theta(s,a)$ is differentiable with respect to the action argument. This enables us to implement an efficient learning rule for the optimal action $a^\phi(s)$ that leverages this property. As a result, instead of running a costly optimization routine every time we need to calculate $\max_{a \in A} Q^\theta(s,a)$, we can approximate it as $\max_{a \in A} Q^\theta(s,a) \approx Q^\theta(s, a^\phi(s))$.

2.3.1 Algorithm

Let us begin by stating the Bellman equation describing the optimal action-value function [27]

$$Q(s,a) = \mathop{\mathbb{E}}_{s' \sim P}\left[r(s,a,s') + (1-d)\max_{a'} Q(s',a')\right], \tag{2.4}$$

where $s' \sim P$ is shorthand for saying that the next state, s', is sampled by the environment from a distribution $P(\cdot|s,a)$ and d indicates whether the state s' is terminal or not.

To learn an approximation of $Q(s,a)$, and $a(s)$ we use a parametrized function $Q^{\theta}(s,a)$ and $a^{\phi}(s)$ with parameters θ and ϕ, and transitions (s,a,r,s',d) according to a distribution $\mathcal{D}$. We can evaluate the approximation's quality by calculating the mean-squared Bellman error (MSBE) as follows:

$$\begin{aligned} L(\theta,\phi,\mathcal{D}) &= \mathop{\mathbb{E}}_{(s,a,r,s',d)\sim\mathcal{D}}\left[\left(Q^{\theta}(s,a) - \left(r(s,a,s') + (1-d)\max\nolimits_{a'} Q^{\theta}(s',a')\right)\right)^2\right] \\ &\approx \mathop{\mathbb{E}}_{(s,a,r,s',d)\sim\mathcal{D}}\left[\left(Q^{\theta}(s,a) - \left(r(s,a,s') + (1-d)Q^{\theta}(s',a^{\phi}(s'))\right)\right)^2\right] \end{aligned} \tag{2.5}$$

The Q-Learning algorithms for function approximators, including DQN and its variants as well as DDPG, aim to minimize the MSBE loss function. These algorithms employ two key strategies and have a unique characteristic, which is specific to DDPG.

Replay Buffers

The use of an experience replay buffer is a common feature in algorithms for training deep neural networks to approximate the Q-value function, $Q(s,a)$. The replay buffer, which is a set of past experiences, must be large enough to include a diverse range of experiences, but it is important to strike a balance between having too much and too little data. Too much data may slow down the learning process, while using only the most recent data could lead to overfitting and performance issues.

It's worth noting that DDPG is an off-policy algorithm, meaning it can use experiences from outdated actions. This is possible because the Bellman equation holds for all possible transitions, regardless of how the actions were selected or what occurs after a transition. Any experiences that have been collected can be used when fitting a Q-function approximation through minimization of the mean-squared Bellman error.

Target Parameters

Several Q-Learning algorithms make use of target parameters. The term

$$r(s,a,s') + (1-d)\max_{a'} Q^{\theta}(s',a') \approx r(s,a,s') + (1-d)Q^{\theta}(s',a^{\phi}(s')) \tag{2.6}$$

in the equation (2.5) is called the target, because when we minimize the MSBE loss, we are trying to move the Q-value function close to this target. Q-Learning algorithms incorporate the use of target parameters to stabilize the training process. The target parameters, denoted as θ_{trg} and ϕ_{trg}, are kept close to the actual parameters, θ and ϕ, but with a time delay. The target parameters are used in the mean-squared Bellman error (MSBE) loss function, which is the quantity that the Q-function is trying to minimize.

To avoid instability in MSBE minimization caused by the target parameters dependence during training, the target parameters are updated periodically. In DQN algorithms, this is done by copying the actual parameters to the target every certain number of steps. In DDPG algorithms, the target parameters are obtained by weighted averaging the actual parameters and the existing target parameters with a hyperparameter τ.

$$\begin{aligned} \theta_{\mathrm{trg}} &\leftarrow (1-\tau)\theta_{\mathrm{trg}} + \tau\theta \\ \phi_{\mathrm{trg}} &\leftarrow (1-\tau)\phi_{\mathrm{trg}} + \tau\phi, \end{aligned} \tag{2.7}$$

where τ is a hyperparameter between 0 and 1 (usually close to 0).

2.3.2 Exploration vs. Exploitation

To train the optimal allocation in DDPG, the algorithm employs an off-policy approach. To increase the exploration of previously untested actions, noise is added to the actions during training. The original DDPG paper recommends using time-correlated Ornstein-Uhlenbeck (OU) noise, but recent findings suggest that using uncorrelated, mean-zero Gaussian noise is just as effective and is simpler. It's possible to reduce the noise scale during training to improve the quality of the training data, but this is not a common practice in implementations and the noise scale is typically kept constant.

3 DDPG Setup and Actor-Critic

Before we present our version, DDPGFunctions, in this chapter we consider the broad set up for our DDPG solution. We specify the tuple - input state, action, rewards and future states in the context of portfolio optimization. In the remainder of the chapter we dive deep into the actor, critic function classes we have used in our experiments for log and power utility functions.

3.1 DDPG Setup

3.1.1 Recap

Let us first restate the portfolio optimization problem below.

$$(\mathbf{P_t}) \begin{cases} \Phi(t,V) = \underset{\pi\in\Lambda}{sup}\, \mathbb{E}[U(V^{v_0,\pi}(T))|V^{v_0,\pi}(t) = V] \end{cases} \tag{3.1}$$

The formulation of Q-learning is considered throughout this project, and DDPG in particular, rely on the fact that the control-process for the dynamic optimization problem is a discrete sequence of clearly separable actions [20]. As $(\mathbf{P}_t)$ is formulated as dynamic optimization problem with continuous-time controls in equation (1.6), we need to formulate an appropriate time-discretization version of it.

We restrict the investor's ability to change his relative portfolio allocation π only at a discrete series of time-points $0 = t_0 < t_1 < .. < t_n = T \quad \text{with } t_i = i\frac{T}{n} = i\Delta t \quad$ for i =0,...,n.

The set of admissible and discretized portfolio processes π can then be characterized as

$$\Lambda^{\Delta t} = \Big\{\pi^{\Delta t} = (\pi_i)_{i=0,...,(n-1)} | \pi_i = \pi(t_i, \cdot) : (0,\infty) \to \mathbb{R}, \pi^{\Delta t} \in \Lambda, i = 0, ..., (n-1)\Big\}.$$

The discretized version of $(\mathbf{P_t})$ can now be defined as

$$(\mathbf{P}_{t_i}^{\Delta t}) \begin{cases} \Phi^{\Delta t}(t_i, v) = \underset{\pi\in\Lambda^{\Delta t}}{sup}\, \mathbb{E}[U(V^{v_0,\pi}(T))|V^{v_0,\pi}(t_i) = v], \end{cases} \tag{3.2}$$

where $\pi^{\Delta t}$ is shortened as π for ease of exposition.

3.1.2 Setup

Given this discretized version, $(\mathbf{P}_{t_i}^{\Delta t})$ of P_t we can embed $(\mathbf{P}_{t_i}^{\Delta t})$ into a deep Q-learning framework with:

- The state space $S = \{t_1, .., T\} \times (0, \infty)$, where the first component is time and the second component represents investor's wealth.
- The action space defined as the admissible values for the relative portfolio process, i.e. $A = \mathbb{R}$.
- The action sequences defined as the discretized relative portfolio processes, i.e. $a = \pi$ for $\pi \in \Lambda^{\Delta t}$
- The wealth updates according to a wealth update process in the continuous space as specified below.

$$\begin{aligned} v_{i+1} = V^{v_0,\pi}(t_{i+1}) = \underbrace{V^{v_0,\pi}(t_i)}_{=:v_i} \exp(&\textstyle\int_{t_i}^{t_{i+1}} r_c + (\mu - r_c)\pi(s, V^{v_0,\pi}(s)) - \frac{1}{2}(\sigma\pi(s, V^{v_0,\pi}(s))^2 ds \\ &+ \textstyle\int_{t_i}^{t_{i+1}} \pi(s, V^{v_0,\pi}(s))\sigma dW(s)) \\ &= v_i \exp\left(r_c + (\mu - r_c)\pi(t_i, v_i) - \frac{1}{2}\left(\sigma\pi(t_i, v_i)\right)^2 \Delta t + \pi((t_i, v_i))\sigma\Delta W_{t_{i+1}}\right) \end{aligned} \tag{3.3}$$

where $\Delta W_{t_{i+1}} \backsim \mathcal{N}(0, \Delta t)$

Using the risky asset log returns, $\Delta P_{t_{i+1}} = ln\left(\frac{P_1(t_{i+1})}{P_1(t_i)}\right)$, we may alternatively write

$$v_{i+1} = v_i \exp\left((1 - \pi(t_i, v_{i+1})r_c\Delta t + \pi(t_i, v_i)\Delta P_{t_{i+1}} + \frac{1}{2}\sigma^2\pi(t_i, v_i)(1 - \pi(t_i, v_i))\Delta t\right)$$

- The reward function defined as

$$r(s, a, s') = r\left((t_i, v_i), \pi(t_i, v_i), (t_i + \Delta t, v_{i+1})\right) = \begin{cases} 0, \text{ if } t_i + \Delta t \neq T \\ U(v_{i+1}), \text{ if } t_i + \Delta t = T \end{cases} \tag{3.4}$$

For setting up the actor and critic, we use specific function classes for log and power utility functions. For these functions, the true critic and the actor can be computed in closed form. We present the results in the following section.

3.2 Log Utility Function

For the log utility function $U(v) = log(v)$, the optimal action a^* (correspondingly the optimal allocation π^*) (see Remark 3.1 in [1]) is

$$a^*(t, v) = \pi^* = \frac{\mu - r_c}{\sigma^2}. \tag{3.5}$$

The optimal action is independent of time and wealth. Hence, rather than learning μ, r, σ directly, we may parametrize $a^* \approx a^\phi$ as

$$a^\phi(t, v) = \pi^* = \phi \quad \text{for some } \phi \in \mathbb{R}. \tag{3.6}$$

The value function Φ (see Remark 3.1 in [1]) is given as

$$\Phi(t,v) = \log(v) + \left[r_c + \frac{1}{2}\left(\frac{\mu - r_c}{\sigma}\right)^2 (T-t) \right]. \tag{3.7}$$

The Q-Value function can then be derived (see Lemma 3.4 in [1]) as

$$Q(t_i,v,a) = \log(v) + (r_c + (\mu - r_c)a - \tfrac{1}{2}\sigma^2 a^2)\Delta t + (r_c + \tfrac{1}{2}(\tfrac{\mu-r_c}{\sigma})^2)(T - t_{i+1}) \tag{3.8}$$

Again, rather than learning μ, r_c, σ directly, we can identify Q as a quadratic function in a with parameters $\theta_0, \theta_1, \theta_2,$ and θ_T such that

$$Q(t_i,v,a) = \log(v) + (\theta_0 + \theta_1 a + \theta_2 a^2)\Delta t + \theta_T(T - t_{i+1}). \tag{3.9}$$

3.3 Power Utility Function

For the power utility function $U(v) = \frac{1}{v}v^b$, the optimal action a^* (correspondingly the optimal allocation π^*) (see Remark 3.1 in [1]) is

$$a^*(t,v) = \pi^* = \frac{\mu - r_c}{(1-b)\sigma^2} \tag{3.10}$$

The optimal action is independent of time and wealth. Hence rather than learning μ, r, σ directly, we may parametrize $a^* \approx a^\phi$ as

$$a^\phi(t,v) = \pi^* = \phi \quad \text{for some } \phi \in \mathbb{R} \tag{3.11}$$

The value function Φ (see Remark 3.1 in [1]) is given as

$$\Phi(t,v) = \frac{1}{b}v^b \exp\left(\left(br_c + \frac{1}{2}\left(\frac{\mu - r_c}{\sigma}\right)^2 \frac{b}{1-b} \right)(T-t) \right). \tag{3.12}$$

The Q- Value function can be derived (see Lemma 3.5 in [1]) as

$$\begin{aligned} Q(t_i,v,a) = \frac{1}{b}v^b \exp\left(b\left[r_c + \frac{(\mu - r_c)^2}{2(1-b)\sigma^2} \right](T - t_{i+1}) \right) \\ \exp\left(\left[br_c + b(\mu - r_c)a + \frac{1}{2}b(b-1)\sigma^2 a^2 \right]\Delta t \right) \end{aligned} \tag{3.13}$$

Again, rather than learning μ, r_c, σ directly, we can identify Q as an exponentially quadratic function in a, with parameters $\theta_0, \theta_1, \theta_2,$ and θ_T such that

$$Q(t_i,v,a) = \tfrac{1}{b}v^b \exp\left((\theta_0 + \theta_1 a + \theta_2 a^2)\Delta t + \theta_T(T - t_{i+1}) \right). \tag{3.14}$$

Case	**Utility Function**	**Actor**	**Learnable Parameters**	**Critic**	**Learnable parameters**
(i)	Log	$\frac{\mu-r_c}{\sigma^2}$	None	(3.8)	μ and σ
(ii)	Log	$\frac{\mu-r_c}{\sigma^2}$	μ and σ	(3.8)	μ and σ
(iii)	Log	ϕ	ϕ	(3.9)	$\theta_0, \theta_1, \theta_2$ and θ_T
(i)	Power	$\frac{1}{1-b}\frac{\mu-r_c}{\sigma^2}$	None	(3.13)	μ and σ
(ii)	Power	$\frac{1}{1-b}\frac{\mu-r_c}{\sigma^2}$	μ and σ	(3.13)	μ and σ
(iii)	Power	ϕ	ϕ	(3.14)	$\theta_0, \theta_1, \theta_2$ and θ_T

Table 3.1: Actor critic functions

3.4 Function classes for Actor and Critic

Based on the Q-value and a-value functions that we have defined for the utility functions, we can build different configurations of actor and critic. We summarize some of the possible configurations in Table 3.1.

We can use the previously defined parametrizations of actor and critic derived for log and power utility functions in three natural ways:

- (i) Learn the market parameters μ and σ through the calibration of the critic and pass them on directly to the actor

  (ii) Learn the market parameters μ and σ simultaneously through the calibration of the actor and critic

  (iii) Learn the structural parameters $\phi, \theta_0, \theta_1, \theta_2,$ and θ_T simultaneously through the calibration of the actor and critic
- Alternatively one could still consider choosing more generic function classes, such as neural networks, in such a way that structural properties of the optimization problem are still retained. This can be very useful as many utility functions settings are complex and explicit solutions for Q functions may not be found. However we might still expect some structural form of the final solution.

4 DDPGFunctions

In this chapter, we present a version of DDPG, which is implemented and set up from scratch with customizable functions for the actor and critic. We detail the pseudocode of our first version of our algorithm called DDPGFunctions and discuss the 4 main components of the algorithm

- Event loop
- Environment
- Replay Buffer
- DDPG with actor and critic

We then summarize the methodology by going over some of the features of the algorithm.

4.1 Algorithm

There are 4 major components in the algorithm. To describe them completely, we first go over the main event loop, where the experiments progress. We then describe the individual components in the event loop and drill deep into each one of the components to understand the whole picture.

4.1.1 Event Loop

This is the main driver of the algorithm where the agent interacts with the environment in an episodic way. At the beginning of every episode, the agent begins with a wealth v_0 , and is invested in a basket of a risky and riskless asset. Based on the action learned by the agent, the allocations change over time between the 2 assets. The agent (i.e the investor) obtains a reward r at the end of an episode which ends at time T. The goal of a whole experiment is to maximize the expected utility of the agents wealth at time T. The pseudo code of what happens in an episode is described below.

Most of the pseudocode is very self explanatory. At line number 12, one can see that the optimal action for the state is selected. Then, based on a policy, a certain level of noise is added to the optimal action. This controls the exploration part of the DDPG algorithm. There is a replay buffer that records all the experiences in line 14 (again described more in Section 4.1.3). The DDPG's learning method, "learn", then updates the optimal Q and A values for both the actual and the target network respectively. We finally record metrics to be used for tracking and visualizing results.

Algorithm 1 Event loop

```
Require: $total\_episodes \geq 100$
 1: $ddpg \leftarrow DDPG$
 2: $actor \leftarrow Actor$
 3: $critic \leftarrow Critic$
 4: $buffer \leftarrow ReplayBuffer$
 5: $env \leftarrow Enviroment$
 6: $settings \leftarrow Settings$
 7: $ep \leftarrow 0$
 8: $noiseFactor \leftarrow 1$
 9: while $ep <= total\_episodes$ do
10:     $currentState \leftarrow env.reset$
11:     while TRUE do
12:         $action \leftarrow actor.\phi^{actual}(currentState)$
13:         $action \leftarrow Policy(action, settings.NoiseScale, settings.Factor)$        ▷ See Algorithm 2
14:         $nextState, reward, isDone \leftarrow env.step(currentState, action)$
15:         $buffer.record(nextState, reward, isDone)$
16:         $ddpg.learn()$
17:         if decayTau then
18:             $ddpg.updateTau((total\_episodes - ep)/total\_episodes)$
19:         end if
20:         if isDone then
21:             BREAK
22:         end if
23:         $currentState \leftarrow nextState$
24:         Record metrics
25:     end while
26: end while
```

Algorithm 2 Policy for optimal action

```
Require:
    $action, scale, factor \leftarrow Action, Scale, Factor$

 1: $noise \leftarrow Noise\_object$        ▷ Could be OU or Gaussian process (config based)
 2: $policy\_action = action + scale * factor * noise$
 3: return $policy\_action$
```

4.1.2 Environment

The environment itself is a custom discrete-time Black-Scholes environment with one risky-asset and bank account. The environment simulates the evolution of the investor's portfolio according to a discrete version of the wealth SDE. The interesting function 'step' is explained in the following pseudocode.

Algorithm 3 Environment Step

```
Require: $State, action$
1: $v \leftarrow State.GetWealth()$
2: $t \leftarrow State.GetTime()$
3: $\Delta W \leftarrow$ Generate standard normal variable
4: $\Delta P \leftarrow (\mu - 0.5\sigma^2)\Delta t + \sigma\sqrt{\Delta t}\Delta W$
5: $v \leftarrow v \exp\left((1-action)r_c\Delta t + action\Delta P + \frac{1}{2}action(1-action)\sigma^2\Delta t\right)$
6: $t \leftarrow t + \Delta t$
7: $done \leftarrow False$
8: if t = T then
9:     $reward \leftarrow Utility(v)$
10:    $done \leftarrow True$
11: else
12:    $reward \leftarrow 0$
13: end if
14: return $(t, v), reward, done$
```

At line 4, one can see the log returns of the risky asset being generated. The wealth of the portfolio is then updated at line 5 , (see wealth dynamics, [5] [Inv. Strategies script (by Prof. Zagst) Theorem 2.18]. The reward is only obtained at the end of the episode. For all other time steps, the reward is 0. The utility function can either be a log or power utility function in our experiments.

4.1.3 Replay Buffer

The replay buffer records experiences in a preset container of fixed size and evicts experiences when the buffer becomes full. The DDPG component samples random experiences from this buffer and learns the optimal Q-and a-value functions. Each observation has to be independent of each other and the replay buffer provides a way of sampling independent observations. Otherwise, sampling the last N observations will make the observations highly correlated to each other and gradient descent would not work on those scenarios.

The replay buffer can support numerous eviction policies. Some common policies used are

- FIFO - Oldest experiences are evicted out in this policy.
- MRU - Most recently used experiences can be evicted out. This would give a chance to sample from unused observations by evicting out already used observations.

4.1.4 DDPG

The DDPG module is invoked by the event loop to update the value of Q and A parameters on every step. In DDPGFunctions, the main steps of the update function are detailed in the following pseudocode.

DDPG as we have established is an off policy algorithm - in that sense, the policy that is being learned at every iteration is not the policy used to make decisions to traverse to the next state in terms of portfolio allocation. This can be seen in the way there are 2 sets of parameters presented in the algorithm 4 - actual and target. The target is in fact here, a slow moving version of the actual parameters that are both being updated at every step - τ being the factor that controls this learning. Another implementation specific comment on the algorithm is that since we do not necessarily use neural networks, we had to customize the backpropagation step of gradients by exposing an API in both the Q-value function and the a-value function, that would transmit the trainable variables. The implementation can then be generic to include any parametrized function(even a neural network) so long as we can get a list of trainable variables.

An highlight of our implementation, is that the different modules in the update step are loosely coupled to each other - which helps us to experiment with different configurations. The critic and actor are both external to DDPG and can be the different implementations we have talked about in the Chapter 3. Also, we provide hooks to have a custom actor function that can be fed in by the critic or invoke any conventional actor that can be trained.

Algorithm 4 DDPG Update

```
1:  $state, action, reward, next\_state \leftarrow ReplayBuffer.getBatch$
2:  $actor \leftarrow Actor$
3:  $critic \leftarrow Critic$
4:
5:  ******* Update Critic *****
6:
7:  $action^{target} \leftarrow actor.\phi^{target}(next\_state)$
8:  $Q^{target}(state, action) \leftarrow reward + critic.Q\theta^{target}(next\_state, action^{target})$
9:  $Q^{actual}(state, action) \leftarrow critic.Q\theta^{actual}(state, action)$
10: $criticLoss \leftarrow (Q^{target}(state, action) - Q^{actual}(state, action))^2$
11: $criticLossGradient \leftarrow Gradient(criticLoss, critic.Trainablevariables)$
12: $ApplyGradients(criticLossGradient, critic.Trainablevariables)$
13:
14: ******* Update Actor *****
15:
16: if $actor.needsGradientUpdate$ then
17:     $action^{actual} \leftarrow actor.\phi^{actual}(state)$
18:     $optimalCriticValue \leftarrow critic.Q\theta^{actual}(state, action^{actual})$
19:     $actorLoss \leftarrow \sum_N optimalCriticValue$          ▷ N is the minibatch size
20:     $actorLossGradient \leftarrow Gradient(actorcLoss, actor.Trainablevariables)$
21:     $ApplyGradients(actorLossGradient, actor.Trainablevariables)$
22: else
23:     $actor.applyCustomUpdate()$     ▷ When parameters have to be passed into a non learnable actor
24: end if
25:
26: ******* Update Target Actor *****
27:
28: $action^{target}, action^{actual} \leftarrow actor.getAllVariables$
29: for $a^{target} \in action^{target}$ and $a^{actual} \in action^{actual}$ do
30:     $a^{target} = \tau a^{actual} + (1 - \tau) a^{target}$
31: end for
32:
33: ******* Update Target Critic *****
34:
35: $critic^{target}, critic^{actual} \leftarrow critic.getAllVariables$
36: for $c^{target} \in critic^{target}$ and $c^{actual} \in critic^{actual}$ do
37:     $c^{target} = \tau c^{actual} + (1 - \tau) c^{target}$
38: end for
```

4.2 DDPGFunctions - Features

Our DDPGFunctions algorithm builds on top of the original DDPG solution with neural networks. We present a detailed analysis of the results of the algorithm in a later chapter. However in this brief section we comment on the features of our algorithm and the problems it is expected to solve.

- **Faster simulations**: This is one of our key expectations/motivation of our algorithm. Having conventional deep layered neural network would lead to exploding run times. The problem becomes practically intractable with marginal additional complexity to the environmental set up. For example, having a 100-step simulation for 2 or more assets would by itself take weeks to converge to stable results [19]. Since our functions are very simple, and the number of tunable parameters is not more than 5, convergence should be observed more rapidly.

- **Accurate convergence**: In most of the experiments we conducted, the problem setting is relatively simple, where the critic and the actor parametrizations are given almost explicitly the problems they are expected to solve. Thus we expect our simulations to converge to values very close to the exact results.

- **Building custom deep networks**: We can still use the algorithm to build deep neural networks in settings where we cannot explicitly (analytically) specify the function for Q-and a- value functions. In these complex settings, we can still start off with a general form, which includes properties which the function are known or expected to have. Then, we use deep neural networks for the parameters for that function. We can exploit the structural characteristics of the function while also including the advantages of having a neural network for modeling unknown properties.

5 DDPG Shock Buffer

In this chapter we analyze some of the problems behind DDPGFunctions described in Chapter 4. We then discuss the reasons why this could happen and ways to mitigate the problems. Afterwards, we present our 2nd version of our DDPG algorithm - DDPG Shock Buffer and explain why in this algorithm, some of the problems in DDPGFunctions get alleviated. We finally explore some hyper parameter constructs with respect to DDPG Shock Buffer and how they impact the simulations.

5.1 DDPG Functions - Problems

We noticed that while DDPG functions provided good results (we discuss them in detail in chapter 8), sometimes we noticed the gradients explode in certain configurations.

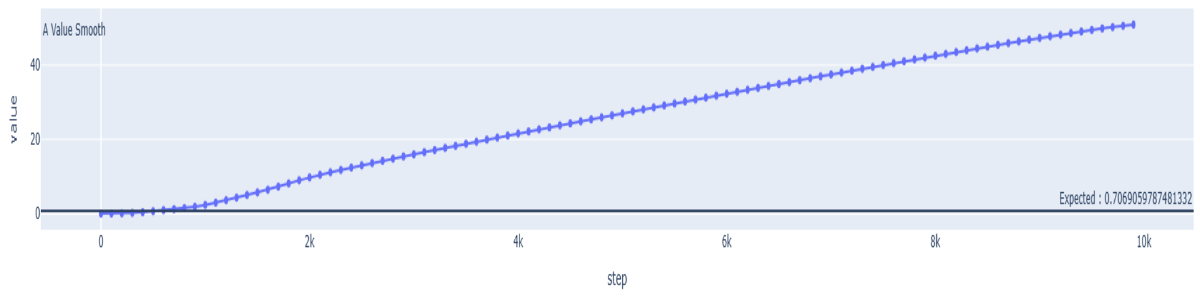


Figure 5.1: A-value (a^{ϕ}) during training

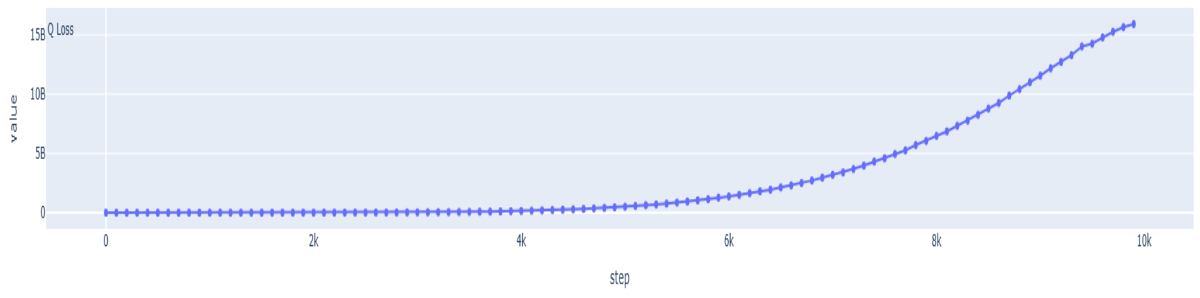


Figure 5.2: Approximate MSBE or "Q-loss" (see (2.5)) during training

In Figures 5.1 and 5.2, we can see that both MSBE and a-value explode after around 800 training steps. We saw a non trivial number of experiments (37/1404) , around 2.6%

where the gradients exploded while testing on different market parameters. In addition, we observed that for around 2.5% of experiments the relative error between the learned portfolio allocation and the optimal allocation was more than 100%.

Further, we noticed that in many experiments, the variance of the learned allocation was very high during training.

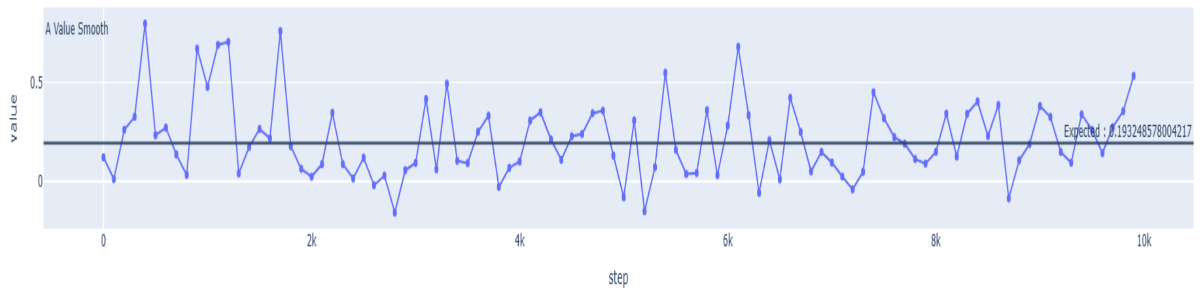


Figure 5.3: Example of high variance of a^{ϕ} during training

In addition to both these issues, we also wanted to explore and obtain more accurate approximation a^{ϕ} for the optimal portfolio π^* .

5.2 Key Idea

To understand what could be a key area of improvement let us restate the Bellman optimality equation defined in Chapter 1

$$\begin{aligned} Q^*(s,a) &= \mathop{\mathbb{E}}_{s'\sim P}\left[r(s,a,s') + (1-d)\max_{a'} Q^*(s',a')\right] \\ \iff 0 &= \left(\mathop{\mathbb{E}}_{s'\sim P}\left[Q^*(s,a) - r(s,a,s') - (1-d)\max_{a'} Q^*(s',a')\right]\right)^2 \end{aligned} \tag{5.1}$$

The mean-squared Bellman error (MSBE) function, which tells us roughly how closely Q^{θ} comes to satisfying the Bellman equation is defined as :

$$L(\theta,\phi,\mathcal{D}) = \mathop{\mathbb{E}}_{(s,a,r,s',d)\sim\mathcal{D}}\left[\left(Q^{\theta}(s,a) - \left(r(s,a,s') + (1-d)\max_{a'} Q^{\theta}(s',a')\right)\right)^2\right] \tag{5.2}$$

In the mini batch setting which takes place in the update step of the DDPG, the expectation in Equation (5.2) is approximated, rather than the expectation in (5.1). Specifically DDPG uses the estimate

$$\begin{aligned} &\mathop{\mathbb{E}}_{(s,a,r,s',d)\sim\mathcal{D}}\left[\left(r(s,a,s') + (1-d)\max_{a'} Q^{\theta}(s',a') - Q^{\theta}(s,a)\right)^2\right] \\ \approx &\frac{1}{|B|}\sum_{(s,a,r,s',d)\sim B}\left(r(s,a,s') + (1-d)Q^{\theta_{trg}}(s',a^{\phi_{trg}}(s')) - Q^{\theta}(s,a)\right)^2. \end{aligned} \tag{5.3}$$

where B is the mini batch and $Q^{\theta_{trg}}$ and $a^{\phi_{trg}}$ are target networks for critic and actor respectively. However, the term s' is only 1 realization of the state after taking the action a, in state s. Hence, we regard the expression inside the expectation in Equation (5.2) as a Monte-Carlo estimate for (5.1) where just *one* realization of s' is used for each state-action pair (s,a). This is bound to make the simulation unstable and may lead to the exploding gradients problem as we saw in Figure 5.1 We aim to mitigate this issue by obtaining more accurate estimates for the expectation in (5.1) without requiring more samples. We contend that this should solve our exploding gradient problem and also improve our results in terms of accuracy.

5.3 Algorithm

One way of obtaining a better estimate of the expectation in the Equation (5.1), is to create two separate buffers R_a and R_p. Only the observed state-action pairs $(s,a) = (t,v,a)$ are stored in R_a, whereas the observed one-period log-returns of the risky assets (or "shocks"), ΔP, are stored in R_p. For given $(t,v,a) \in R_a$ and log-return $\Delta P \in R_p$, we can determine a realization of the consecutive state $s' = (t + \Delta t, V^u)$ and reward r by defining

$$\begin{aligned} V^u &= V^u(v,a,\Delta P) = V \exp\left((1-a)r_c\Delta t + a\Delta P) + \tfrac{1}{2}a(1-a)\sigma^2\Delta t\right) \\ r(s,a,s') &= r(t_i,V^u) = \begin{cases} 0, \text{ if } t_i + \Delta t \neq T \\ U(V^u), \text{ if } t_i + \Delta t = T. \end{cases} \end{aligned} \tag{5.4}$$

In particular, as the shocks $\Delta P \in R_p$ are generated independently from R_a we can sample a mini-batch $B_p \subset R_p$ to obtain a better estimate of (5.1) for any $(s,a) = (t,v,a) \in R_a$:

$$\begin{aligned} &\left(\mathbb{E}_{s'\sim P}\left[Q(s,a) - (r(s,a,s') + (1-d)\max_{a'} Q(s',a'))\right]\right)^2 \\ &\approx \left(Q(t,v,a) - \left(\frac{1}{|B_p|}\sum_{\Delta P \in B_p} r(t,V^u) + \mathbb{1}_{\{t+\Delta t \neq T\}} \max_{a'}(Q(t+\Delta t, V^u, a'))\right)\right)^2 \end{aligned} \tag{5.5}$$

Using (5.5) and mini-batches $B_a \subset R_a$ and $B_p \subset R_p$, improve the update rule of the actor and critic parameters θ and ϕ to

$$\begin{aligned} &\theta \leftarrow argmin_{\theta'} \frac{1}{|B_a|}\sum_{(t,v,a)\in B_a}\left(Q^{\theta'}(t,v,a) - \frac{1}{|B_p|}\sum_{\Delta P\in B_p} r(t,V^u) + \mathbb{1}_{\{t+\Delta t\neq T\}} Q^{\theta_{trg}}(t+\Delta t, V^u, a^{\phi_{trg}}(t+\Delta t, V^u))\right)^2 \\ &\phi \leftarrow argmax_{\phi'} \frac{1}{|B_a|}\sum_{(t,v,a)\in B_a} Q^{\theta}(t,v,a^{\phi'}(t,v)). \end{aligned} \tag{5.6}$$

The whole procedure can be explained in the following pseudo code

Algorithm 5 DDPG Shock Buffer Update

```
1: $state_i, action_i \leftarrow ReplayBuffer.getBatch$
```

```
 2: $wealth_i, time_i \leftarrow state_i$
 3: $dP \leftarrow ReplayBuffer.getShockBatch$
 4: $actor \leftarrow Actor$
 5: $critic \leftarrow Critic$
 6:
 7: ******* Update Critic *****
 8:
 9: $wealth_{i+1} \leftarrow wealth_i * env.wealthGridUpdate(action, dP)$    ▷ This is a big update with $wealth_{i+1}$ dimensions being (state_buffer_size X shock_buffer_size)
10: $time_{i+1} \leftarrow time_i + \Delta t$
11: $time_{i+1} \leftarrow RepeatAcrossShockBufferSize(time_{i+1})$    ▷ In these lines we adjust the dimension from $1 \times |m|$ to $n \times m$, where n is state_buffer_size and m is shock_buffer_size
12: $state_{i+1} \leftarrow (wealth_{i+1}, time_{i+1})$    ▷ Dimension: (state_buffer_size X shock_buffer_size)
13: $action_{i+1} \leftarrow aN.\Phi^{target}(state_{i+1})$
14: if $t_{i+1} = T$ then
15:     $reward_{i+1} \leftarrow env.U_2(wealth_{i+1})$
16: else
17:     $reward_{i+1} \leftarrow 0$
18: end if
19: $Q^{target}(state_{i+1}, action_{i+1}) \leftarrow reward_{i+1} + critic.Q\theta^{target}(state_{i+1}, action_{i+1})$    ▷ Dimension: (state_buffer_size X shock_buffer_size)
20:
21: ******* Arrive back at the modified Bellman optimality Equation *****
22:
23: $Q^{target}(state_i, action_i) \leftarrow getMeanAcrossShockBatch(Q^{target})$    ▷ Dimension: state_buffer_size
24: $Q^{actual}(state_i, action_i) \leftarrow critic.Q\theta^{actual}(state_i, action_i)$
25: $criticLoss \leftarrow (Q^{target}(state_i, action_i) - Q^{actual}(state_i, action_i))^2$
26: $criticLossGradient \leftarrow Gradient(criticLoss, critic.Trainablevariables)$
27: $ApplyGradients(criticLossGradient, critic.Trainablevariables)$
28:
29: ******* Update Actor *****
30:
31: if $actor.needsGradientUpdate$ then
32:     $action_i^{actual} \leftarrow actor.\phi^{actual}(state_i)$
33:     $optimalCriticValue \leftarrow critic.Q\theta^{actual}(state_i, action_i)$
34:     $actorLoss \leftarrow \sum_N optimalCriticValue$    ▷ N is the minibatch size
35:     $actorLossGradient \leftarrow Gradient(actorcLoss, actor.Trainablevariables)$
36:     $ApplyGradients(actorLossGradient, actor.Trainablevariables)$
37: else
38:     $actor.applyCustomUpdate()$    ▷ When parameters have to be passed into a non
```

```
    learnable actor
39: end if
40:
41: ******* Update Target Actor *****
42:
43: $action_i^{target}, action_i^{actual} \leftarrow actor.getAllVariables$
44: for $a^{target} \in action_i^{target}$ and $a^{actual} \in action_i^{actual}$ do
45:     $a^{target} = \tau * a^{actual} + (1-\tau)a^{target}$
46: end for
47:
48: ******* Update Target Critic *****
49:
50: $critic^{target}, critic^{actual} \leftarrow critic.getAllVariables$
51: for $c^{target} \in critic^{target}$ and $c^{actual} \in critic^{actual}$ do
52:     $c^{target} = \tau * c^{actual} + (1-\tau)c^{target}$
53: end for
```

The main difference between Algorithm 4 and Algorithm 5 is in the update of the critic actor function. All the other parts of the algorithm are identical.

The most interesting aspects are at line 9 where instead of 1 realization of s', we have a shock_buffer size of realizations of s'. Afterwards we compute the next optimal action for all these states s' at line 13 and the corresponding rewards at lines 15 and 17. We then compute the expected reward and expected Q value for the next (state,action) tuple by taking a mean over the shock buffer. We use the expected reward and the expected Q value to compute the Bellman optimality equation.

In the pseudo code at line 9, we define the wealth update in the environment module as

Algorithm 6 DDPG Wealth Update Shock Buffer

```
1: $\mathbf{\Delta P} \leftarrow$ "Shock" returns sampled from replay buffer$R_p$    ▷ Size : N_s x 1
2: $\mathbf{a} \leftarrow$ Actions sampled from replay buffer $R_a$    ▷ Size: N_b x 1
3: $r, \Delta t \leftarrow$ Risk free rate, discretized time step
4: return $\exp\left\{(\mathbb{1}_{N_b} - \mathbf{a})\mathbb{1}'_{N_s} r\Delta t + \mathbf{a}\mathbf{\Delta P}' + \frac{1}{2}\mathbf{a}(\mathbb{1}_{N_b} - \mathbf{a})\mathbb{1}'_{N_s}\sigma^2\Delta t\right\}$    ▷ $\mathbb{1}$ is a vector of all one(s) of dimension defined in subscript.
```

The environment step function is very similar to the one discussed in Chapter 4, the only difference being the log return of the "shock" that is also generated is stored in a separate buffer every time a transition happens.

5.3.1 Hyper Parameter Considerations

In this very brief section, we consider the effect of the size of the shock buffer and the batch size for the shock buffer. Since the batch size of the shock buffer multiplies with the batch size of the state transitions, setting a high value for the shock buffer batch size, will increase the training times considerably. At the same time setting a low value for the shock buffer batch size will result in the same problems discussed in the original version explained in Chapter 4. Hence a trade off experimentation is needed to set the optimal size of the shock buffer batch sizes.

The shock buffer size itself can be completely independent of the state buffer size. Older observations need not be discarded if the model parameters of the environment are static over time. However, in real-life scenarios, one can also think that the distribution of the log shock returns will change over time. So, having a policy like FIFO to discard the older shock observations makes sense in such settings.

6 DDPG Estimates

In this chapter we try to analyze some of the problems behind DDPG Shock Buffer described in Chapter 5. We then discuss the reasons why this could happen and ways to mitigate the problem. We then present our third version of the DDPG algorithm - DDPG Estimates and explain why in this algorithm, some of the problems in DDPG Functions get alleviated. We finally explore some hyper parameter constructs with respect to DDPG Estimates and how they can impact the simulations.

6.1 DDPG Shock Buffer - Problems

One considerable improvement of DDPG Shock Buffer over DDPG Functions was in making the simulations stable. We noticed no experiment that exploded in the calculations in that scenario (multiple market parameters for experiments up to 10 time steps). Also the number of experiments where the relative error rate was more than 100% was around 2.06% which was lower than DDPG Functions (we noted that such bad simulations happened when the optimal allocations were close to 0).

While the above results looked promising, we noticed that the learned optimal action a^{ϕ} still exhibits a high variance during training when using Shock Buffer, see Figure 6.1 .

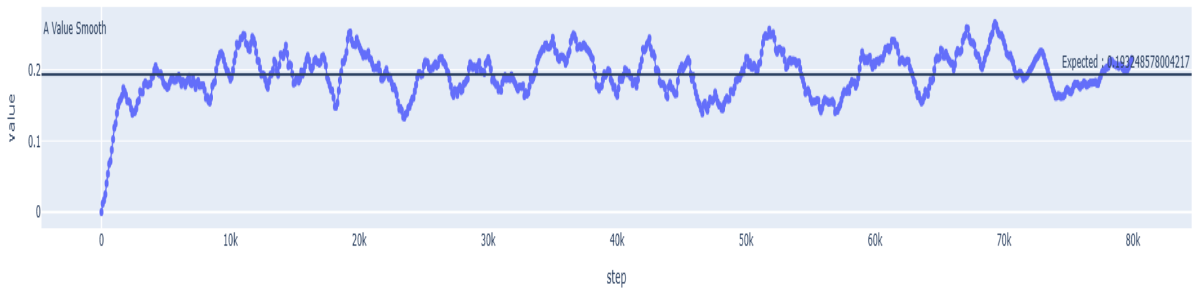


Figure 6.1: Example of high variance of a^{ϕ} during training of Shock Buffer

The other problem we faced (which also happened always with DDPG Functions) was that as the number of time steps increased, the probability that the gradients explode increased significantly. This is illustrated in Figure 6.2.

We noticed that the error rates could further be improved. The median accuracy we obtained for a batch of experiments was around 92%.

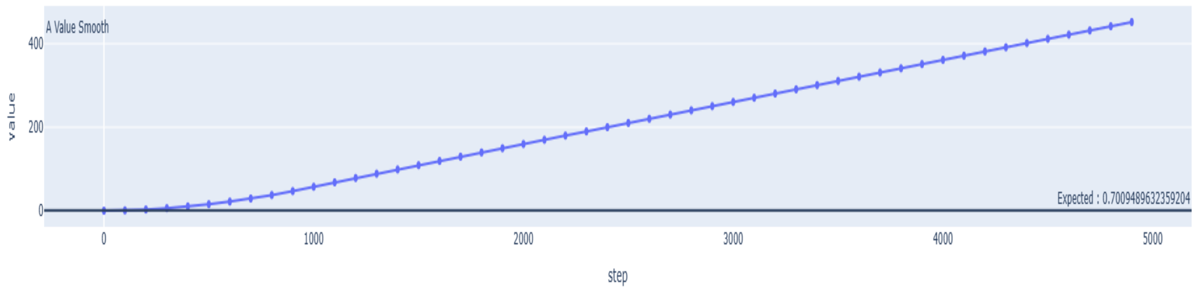


Figure 6.2: A-value (a^{ϕ}) exploding during training - Shock Buffer

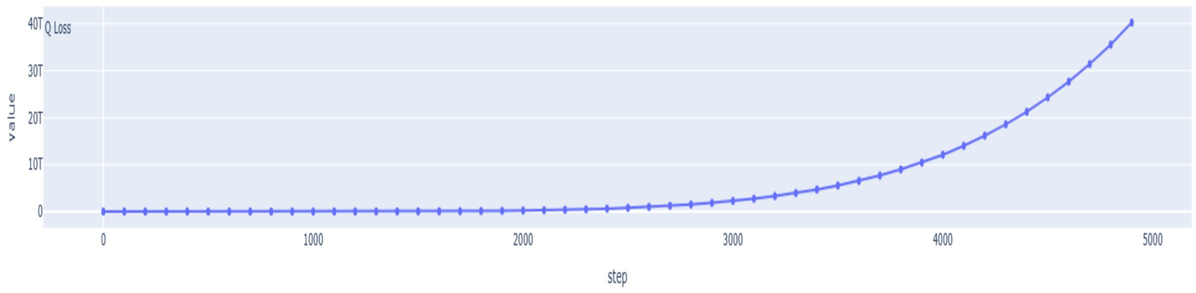


Figure 6.3: Approximate MSBE or "Q-loss" (see (2.5)) exploding during training

6.2 Key Idea

Let us first look at some possible shortcomings in the formulation of DDPG Shock Buffer. In DDPG Shock Buffer we strive to obtain a better estimate of the expectation defined in equation (5.1). In DDPG Shock Buffer, this is achieved by generating multiple realizations of s' on every update step by sampling from $m = |B_p|$ realizations of the past shock returns we have experienced and stored in R_p.

We now consider another way of obtaining a better estimate of the expectation defined in equation (5.1). by using the information of the distribution of the log returns dP. Let us restate the probability density function for a standard normal distribution.

$$\varphi(x) = \frac{1}{\sqrt{(2\pi)}} e^{\frac{-x^2}{2}} \tag{6.1}$$

Let $\beta > 0, m \in \mathbb{N}$ and let $-\beta = z_{-n} < z_{-(n-1)} < ... < z_0 = 0 < z_1 < ... < z_n = \beta$ be a partition of $[-\beta, \beta]$ for large β. Then for any (regular) $f : \mathbb{R} \to \mathbb{R}$ and $Z \sim \mathcal{N}(0,1)$

$$\mathbb{E}[f(Z)] = \int_{\mathbb{R}} f(z)\varphi(z)dz \approx \int_{-\beta}^{\beta} f(z)\varphi(z)dz \approx \sum_{i=-(m-1)}^{m} f(z_i)\varphi(z_i)(z_i - z_{i-1}) \tag{6.2}$$

We can then use this idea in our framework while computing our target Q values. In our framework, the next state (i.e the next wealth) given the state $s_i = (t_i, v_i)$ and action a_i can be obtained as:

$$\begin{aligned} v_{i+1} &= v_i \exp\left((1-a_i)r_c\Delta t + a_i\Delta P) + \tfrac{1}{2}a_i(1-a_i)\sigma^2\Delta t\right) \\ &=: V^u(v_i, a_i, z), \\ \text{with, } \Delta P &= (\mu - \tfrac{1}{2}\sigma^2)\Delta t + \sigma z\sqrt{\Delta t} \end{aligned} \tag{6.3}$$

We can then use (6.3) and (6.2) to obtain an approximate Bellman equation.

Define again the reward function $r(s,a,s')$ as

$$r(s,a,s') = r(t_i, V^u) = \begin{cases} 0, \text{ if } t_i + \Delta t \neq T \\ U(V^u), \text{ if } t_i + \Delta t = T. \end{cases} \tag{6.4}$$

Then the Bellman equation becomes

$$\begin{aligned} Q(t,v,a) &= \mathbb{E}[r(t, V^u(v,a,z))] + \mathbb{1}_{\{t+\Delta t\neq T\}}\mathbb{E}[\max_{a'\in A} Q(t+\Delta t, V^u(v,a,z), a')] \\ &= \int_{\mathbb{R}} r(t, V^u(v,a,z))\varphi(z)dz + \mathbb{1}_{\{t+\Delta t\neq T\}} \int_{\mathbb{R}} \max_{a'\in A} Q(t+\Delta t, V^u(v,a,z), a')\varphi(z)dz \\ &\approx \sum_{i=-(m-1)}^{m} [r(t, V^u(v,a,z_i)) + \mathbb{1}_{\{t+\Delta t\neq T\}} \max_{a'\in A} Q(t+\Delta t, V^u(v,a,z_i), a')]\varphi(z_i)(z_i - z_{i-1}) \end{aligned} \tag{6.5}$$

Thus, the core idea is to use the equation derived in (6.5) in the target Q in the critic loss update step. This can be further seen in Algorithm 7.

6.3 Algorithm

The procedure can be explained in the following pseudo code

Algorithm 7 Set Initial Values

```
1: $N \leftarrow Intervals$
2: $z \leftarrow GetPartition(z)$
3: $dP \leftarrow (\mu - \frac{1}{2}\sigma^2)\Delta t + \sigma z\sqrt{\Delta t}$ for all $z$        ▷ Dimension : N
4: $pdf \leftarrow \varphi(z) = \frac{1}{\sqrt{2\pi}}e^{-\frac{1}{2}z^2}$        ▷ Dimension : N
5: $diff_i \leftarrow z_i - z_{i-1}$ for all $i$ except $i = 0$, $diff_0 \leftarrow 0$        ▷ Dimension : N
6: $dP \leftarrow RepeatAcrossStateBufferSize(dP)$        ▷ Dimension : state_buffer_size X N
7: $pdf \leftarrow RepeatAcrossStateBufferSize(pdf)$        ▷ Dimension : state_buffer_size X N
8: $diff \leftarrow RepeatAcrossStateBufferSize(diff)$        ▷ Dimension : state_buffer_size X N
```

Now the DDPG update step is explained below.

Algorithm 8 DDPG Estimate Update

```
1: $state_i, action_i \leftarrow ReplayBuffer.getBatch$
2: $wealth_i, time_i \leftarrow state_i$
3: $actor \leftarrow Actor$
4: $critic \leftarrow Critic$
5:
6: ******* Update Critic *****
7:
8: $wealth_{i+1} \leftarrow wealth_i * env.wealthGridUpdate(action, dP)$    ▷ This is a big update with $wealth_{i+1}$ dimensions being (state_buffer_size X N)
9: $time_{i+1} \leftarrow time_i + \Delta t$
10: $time_{i+1} \leftarrow RepeatAcrossNcolumns(time_{i+1})$    ▷ Tile the next time across all partitions so the dimension lines up with $wealth_{i+1}$
11: $state_{i+1} \leftarrow (wealth_{i+1}, time_{i+1})$    ▷ Dimension: (state_buffer_size X N)
12: $action_{i+1} \leftarrow aN.\phi^{target}(state_{i+1})$    ▷ Dimension: (state_buffer_size X N)
13: if $t_{i+1} = T$ then
14:     $reward_{i+1} \leftarrow env.U_2(wealth_{i+1})$
15: else
16:     $reward_{i+1} \leftarrow 0$
17: end if
18: $Q^{target}(state_{i+1}, action_{i+1}) \leftarrow (reward_{i+1} + critic.Q\theta^{target}(state_{i+1}, action_{i+1})).pdf.diff$    ▷ Dimension: (state_buffer_size X shock_buffer_size) , pdf and diff are defined in algorithm 7
19:
20: ******* Arrive back at the modified Bellman optimality Equation *****
21:
22: $Q^{target}(state_i, action_i) \leftarrow getSumAcrossPartitionAxes(Q^{target})$    ▷ Dimension: state_buffer_size
23: $Q^{actual}(state_i, action_i) \leftarrow critic.Q\theta^{actual}(state_i, action_i)$
24: $criticLoss \leftarrow (Q^{target}(state_i, action_i) - Q^{actual}(state_i, action_i))^2$
25: $criticLossGradient \leftarrow Gradient(criticLoss, critic.Trainablevariables)$
26: $ApplyGradients(criticLossGradient, critic.Trainablevariables)$
27:
28: ******* Update Actor *****
29:
30: if $actor.needsGradientUpdate$ then
31:     $action_i^{actual} \leftarrow actor.\phi^{actual}(state_i)$
32:     $optimalCriticValue \leftarrow critic.Q\theta^{actual}(state_i, action_i)$
33:     $actorLoss \leftarrow \sum_N optimalCriticValue$    ▷ N is the minibatch size
34:     $actorLossGradient \leftarrow Gradient(actorcLoss, actor.Trainablevariables)$
35:     $ApplyGradients(actorLossGradient, actor.Trainablevariables)$
```

```
36: else
37:     $actor.applyCustomUpdate()$    ▷ When parameters have to be passed into a non learnable actor
38: end if
39:
40: ******* Update Target Actor *****
41:
42: $action_i^{target}, action_i^{actual} \leftarrow actor.getAllVariables$
43: for $a^{target} \in action_i^{target}$ and $a^{actual} \in action_i^{actual}$ do
44:     $a^{target} = \tau * a^{actual} + (1-\tau)a^{target}$
45: end for
46:
47: ******* Update Target Critic *****
48:
49: $critic^{target}, critic^{actual} \leftarrow critic.getAllVariables$
50: for $c^{target} \in critic^{target}$ and $c^{actual} \in critic^{actual}$ do
51:     $c^{target} = \tau * c^{actual} + (1-\tau)c^{target}$
52: end for
```

The main difference between Algorithm 4 and Algorithm 8 is in the update of the critic actor function. All the other parts of the algorithm are identical.

The most interesting aspects are at line 8 where instead of 1 realization of s', we have m realizations of s', one realization for each grid point, z_i. We then compute the next optimal action for all these states s' at line 12 and the corresponding rewards at line 14 and 16.

In the pseudo code at line 8, we use the wealth update from Algorithm 6. The environment step function is identical to the one discussed in Chapter 4.

6.3.1 Generating Intervals

While choosing intervals stated in the Algorithm 7 at line 2, it is important that the discretization of the integral using "Riemann" integrals [29] captures the integral approximately well. The naive way of choosing N equally spaced partitions does not account for the changing size of φ. This could in turn lead to large discretization errors in regions where φ is large and thus cause a large error for the approximation of the integral. This can be avoided if we choose the partition in such a way that we have many grid points in regions where φ is large and few grid points in areas where φ is small. One such procedure is described below.

Let Φ be the cumulative distribution function of a standard normal distribution $\mathcal{N}(0,1)$, i.e

$$\Phi(x) = \mathbb{P}[Z \leq x] = \int_{-\infty}^{x} \varphi(z)dz. \tag{6.6}$$

Let $F = \Phi^{-1}$ be the generalised inverse of Φ, i.e the quantile function of $z \sim \mathcal{N}(0,1)$ such that

$$X = F(\Phi(x)) = F(\mathbb{P}(Z \leq x)) \quad \forall x \in \mathbb{R} \tag{6.7}$$

For $m \in \mathbb{N}$, define a partition of [0,1] as

$$y_j = \frac{1}{2}(1 + \frac{j}{m+1}) = \frac{m+1+j}{2(m+1)} \in (0,1) \text{ for } -m \leq j \leq m \tag{6.8}$$

Finally, we define $z_{-m} < ... < z_m$ as

$$z_j = F(y_j). \tag{6.9}$$

7 System Architecture, Design and utility frameworks

In this chapter, we describe some highlights in terms of the architecture of the setup. We start off by summarizing all the components we had described in Chapters 4, 5 and 6 with an architecture diagram and then a class diagram showing the compositional and hierarchical structure of the different classes we designed. We then proceed to give some comments about the architecture in broadly 4 areas

- **Modular components of the setup**. The different components of the set up that can be 'plug and played' are.
  - Actor
  - Critic
  - Replay Buffer
  - DDPG Flavors
- **Tracking , Visualization**. Combining good tracking tools such as MLFlow, we also have implemented a visualization framework based on dash [30], to see the list of results and export the data for charting, and a tailor made list of plot functions that can be extended to further projects that extend this idea.
- **Deployment** - We present our Fastapi[31] based server, from where experiments can be spawned directly from a webserver that can be hosted from any location.
- **Hyper parameter network**. We present a customizable tuning framework from where families of configurations can be launched from a configuration file.

7.1 Architecture

There are 2 different kinds of components in the architecture specified in Figure 7.1 - the core DDPG components and the driver components.

The driver components broadly specify the different ways the project can be run. They are

- **run_all** : This is the traditional version of the project , a command line version from where the project can be run. The command line version takes a configuration file as input which contains the parameters of the experiments along with the hyper

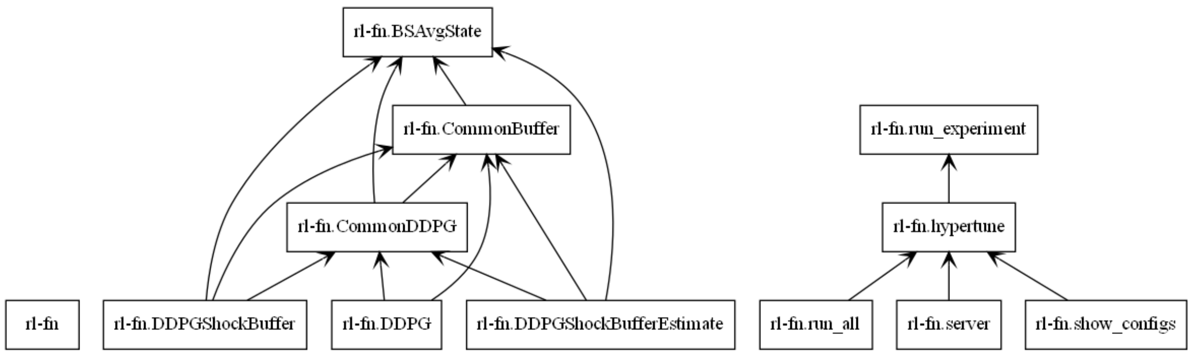


Figure 7.1: Architecture of the different components in the project

parameters, that have to be tuned and calls the **hypertune** component, which would spawn individual configurations for each value of a hyper parameter. The generated config would then be used by the **run_experiment** module which would run individual experiments using the core components of the project.

- **server** : This is almost equivalent to the command line version in terms of the functionality but this is used while the project is hosted on a server, serving Restful APIs [32] from a Flask framework [33]

- **show_configs** : This version is used to only generate all the configurations, corresponding to all the tunable hyper parameters and would not run the individual experiments. This version is ideal to inspect the possible configurations an experiment will be tuned for.

The core components encompass the different DDPG versions **DDPG**, **DDPG ShockBuffer** and **DDPG Estimates** (shown as DDPGShockBufferEstimate in Figure 7.1). The common functionalities for all these components are inherited in a *base class*, **CommonDDPG**. All the 3 versions use the same replay buffer **CommonBuffer**, which is already customized to store the environment variables , extra variables such as log-returns (specific only to Shock buffer) and can be easily extended to store other parameters in any newer setting. The **BSAvgState** is the environment component that is used across all the other components and progresses the episodes of the experiments. The environment currently supports a Black-Scholes process and the utility functions - log and power. Besides the component can be extended to newer utility functions by overriding a utility interface in the environment class.

The class diagram for these components is displayed in Figure 7.2.

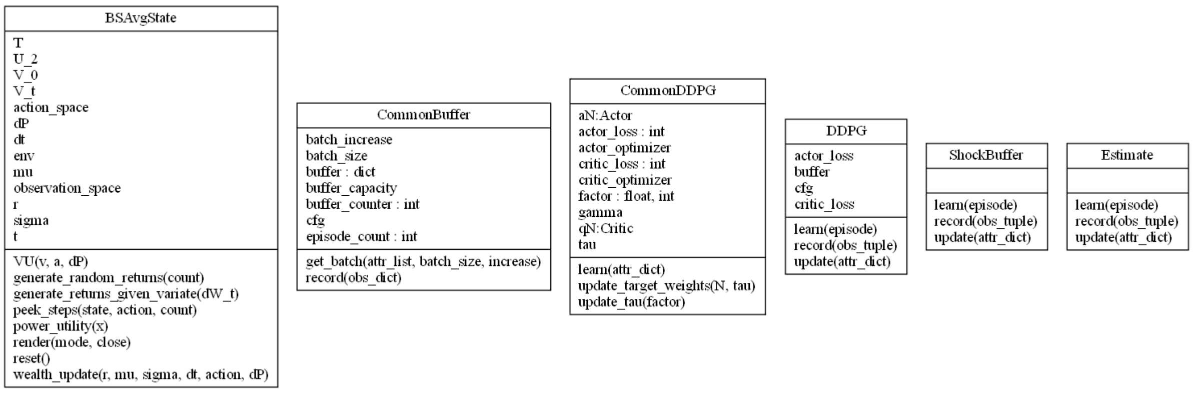


Figure 7.2: Class diagram of DDPG components

Besides, we have a number of classes for the critic and the actor as well. There are 2 types of Q (and A) classes we use depending on how the critic and actor are defined. They are shown in Figure 7.3.

Q_DDPG_Update corresponds to cases (ii) and (iii) defined in Table 3.1, while Q_Custom_Update corresponds to case (i) defined in the same table.The difference between these 2 classes is that the Q_Custom_Update has an extra method update_weight which needs to be customized for case (i).

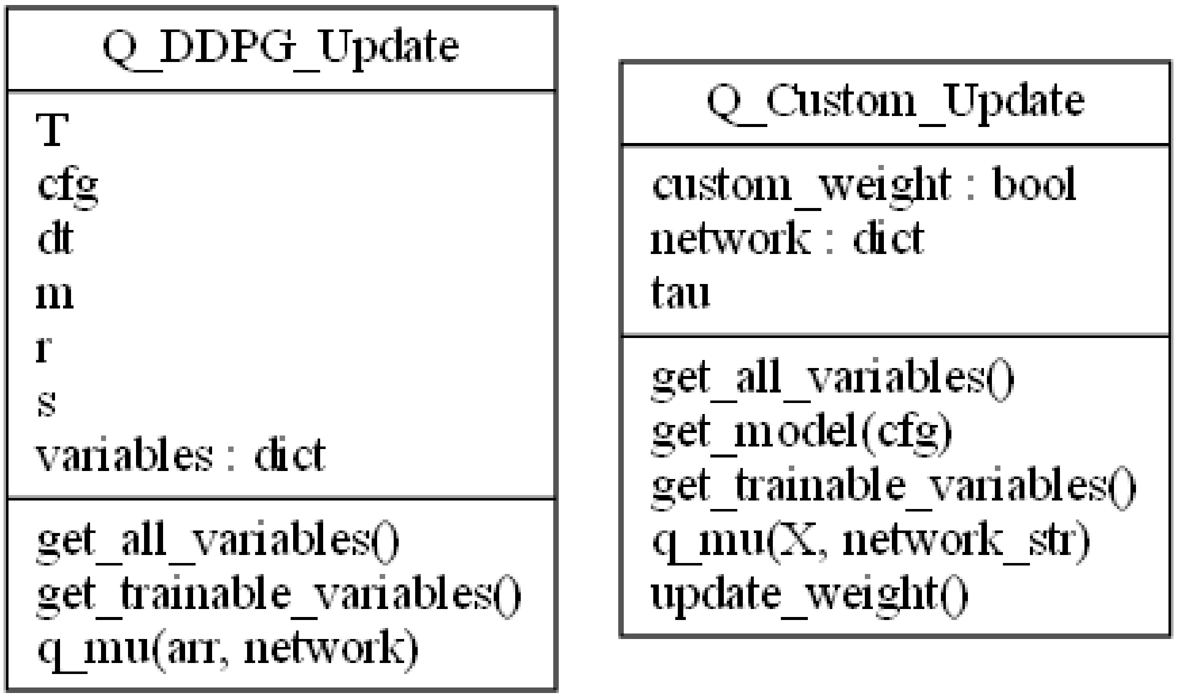


Figure 7.3: Class diagram of critic

7.2 Modular Components

At the heart of every experiment run is a configuration which we take as input. A sample configuration is described below.

Listing 7.1: Example Configuration

```
{
 "name": "Experiment - Decaying tau and batch size in DDPG Shock Buffer
    version",
 "env": {
  "name": "BlackScholes-V2",
  "mu": 0.09,
  "sigma": 0.3,
  "r": 0,
  "T": 1,
  "dt": 0.2,
  "V_0": 1.0,
  "U_2": "pow",
  "b": -1
 },

"general_settings": {
  "max_episodes": 5000,
  "max_steps": 100,
  "batch_size": 1024,
  "batch_size_increase": "linear"
},
"ddpg": {
  "type" :
    {
   "name": "DDPGShockBufferEstimate",
    "m": 20
    },
  "gamma": 1,
  "noise_decay": 1,
  "noise_scale": 1,
  "tau": 0.005,
  "tau_decay": "linear",
  "buffer_len": 20000,
  "q": {
    "name": "q_pow_utparametric",
    "lr": 0.005,
    "variables": [
      0.1,
      0.2,
      0.1,
      0.1
    ]
  },
  "a": {
    "name": "a_pow_ut1",
    "lr": 0.001,
    "variables": {
      "m": 0.1
    }
  }
}
```

As one can see in the Listing 7.1, broadly there are 3 modules:

- **Environment (env)**: This module takes the environment parameters, and a utility function as input. This includes r,μ,σ, the time step interval Δt, T, initial wealth v_0 and the utility function U.

  The parameters are themselves quite flexible and new parameters can be added and existing parameters can be removed easily. For example one can set up a Heston model [34] instead of a Black Scholes Model by defining the evolution of the volatility explained by the equations

$$\begin{aligned} dS_t &= (r + \eta z_t)S_t dt + \sqrt{z_t} S_t dW(t) \\ dz_t &= \kappa(\theta - z_t)dt + \sigma\sqrt{z_t} dW_z(t). \end{aligned} \tag{7.1}$$

In terms of our modular setup, we would have to redefine the step function as specified in the Algorithm 3 and then add the following variables :

- η - Market price of the risk driver
- κ - Mean reversion speed of volatility
- θ - Long-run average volatility
- σ - Volatility of volatility
- ρ - Correlation factor

The other components of the DDPG can just ’flow’ in as they were implemented.

- **DDPG**: The DDPG module itself can be replaced by different flavors / implementation. What we described in Algorithm 4 is only 1 version for our DDPG. As we have seen in Chapters 5 and 6, any DDPG model can be plugged in so long as it includes the following APIs
  - Learn - The function that is meant to update the Q-value and a-value functions.
  - Record - The API that stores all the experience into a replay buffer.

  Besides the APIs that DDPG should support, there are a bunch of hyper parameters that DDPG operates on, that are also listed in the sample configuration.

- **Actor(a) and Critic (Q)** The different actor and critic algorithms that were discussed in detail in Chapter 3 are implemented in these modules. There is no restriction or a sense of constraint between the critic and the actor functions themselves. Any set of interesting modules can be used and experimented in the existing set up. We however require that these modules support the following APIs.
  - The critic should implement a function - Q_mu(s,a) that returns the Q value of the state-action tuple, (s,a).
  - The actor should implement a function Mu(s) that returns the (suggested) optimal action for state ’s’.
  - The function TrainableVariables() for both actor and critic, that provides the list of variables to be trained using stochastic gradient descent in the context of DDPG.
  - The function AllVariables() - A tuple of both trainable variables and the ’target’ network variables that are not trained but just updated.

7.3 Data Visualization, Experiment Tracking, Deployment and Plotting

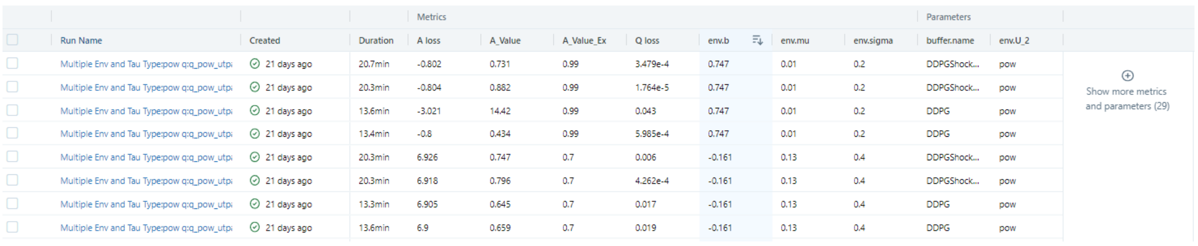
These are some of the downstream activities of our experiments but they are crucial to gain insights from the experiments we were conducting. Since the functions we chose for the actor and critic were quite small in terms of parameters compared to a multi-layered neural network, most of our experiments ran much faster compared to traditional DDPG problems.

Therefore we were able to run a vast number of experiments, tuning different hyper parameters, building many versions of the core DDPG algorithm itself and also tracking the accuracy of our experiments in different environmental conditions - namely under different expected return and volatility of the risky assets.

Manually keeping track of so many experiments was hard and we were able to achieve better tracking by integrating MLFlow [35] into our experiments.

7.3.1 Experiment Tracking and Visualization

Here we present a dashboard that can be constructed from running many configurations in a particular experiment.



			Metrics							Parameters	
Run Name	Created	Duration	A loss	A_Value	A_Value_Ex	Q loss	env.b	env.mu	env.sigma	buffer.name	env.U_2
Multiple Env and Tau Type:pow q:q_pow_utp:	21 days ago	20.7min	-0.802	0.731	0.99	3.479e-4	0.747	0.01	0.2	DDPGShock...	pow
Multiple Env and Tau Type:pow q:q_pow_utp:	21 days ago	20.3min	-0.804	0.882	0.99	1.764e-5	0.747	0.01	0.2	DDPGShock...	pow
Multiple Env and Tau Type:pow q:q_pow_utp:	21 days ago	13.6min	-3.021	14.42	0.99	0.043	0.747	0.01	0.2	DDPG	pow
Multiple Env and Tau Type:pow q:q_pow_utp:	21 days ago	13.4min	-0.8	0.434	0.99	5.985e-4	0.747	0.01	0.2	DDPG	pow
Multiple Env and Tau Type:pow q:q_pow_utp:	21 days ago	20.3min	6.926	0.747	0.7	0.006	-0.161	0.13	0.4	DDPGShock...	pow
Multiple Env and Tau Type:pow q:q_pow_utp:	21 days ago	20.3min	6.918	0.796	0.7	4.262e-4	-0.161	0.13	0.4	DDPGShock...	pow
Multiple Env and Tau Type:pow q:q_pow_utp:	21 days ago	13.3min	6.905	0.645	0.7	0.017	-0.161	0.13	0.4	DDPG	pow
Multiple Env and Tau Type:pow q:q_pow_utp:	21 days ago	13.6min	6.9	0.659	0.7	0.019	-0.161	0.13	0.4	DDPG	pow

Figure 7.4: Sample dashboard showing the different runs of an experiment. Note A_Value_Ex is the theoretical optimal action

The dashboard presents metrics, and parameters that can be logged during an experiment. Besides the metrics themselves, one can also log artifacts such as the configuration file mentioned in Listing 7.1.

Within a run one can also see the progression of an experiment. The following plot shows the multi series of learned a-value, optimal a-Value, Q losses and a losses (i.e the maximization objective in the actor update) as we progress within an experiment.

Further, we also generated a customizable framework to create even more filters for our dashboards that MLFlow did not support. Our customizable framework is based on dash [30] and a sample screen shot of that is found in Figure 7.6. We also hosted this framework at https://ddpg-po-dash1.herokuapp.com/ so that it is accessible everywhere. The difference between the framework is that the dash framework (implemented from scratch in terms of layout), has better filtering capabilities, and is much faster compared to the MLFlow UI. The layout has been improved to consider the specifics of the problem

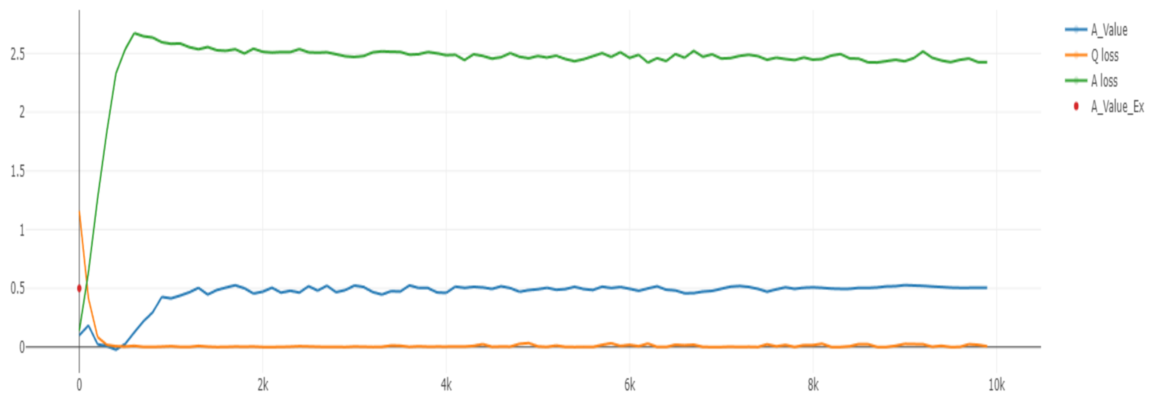
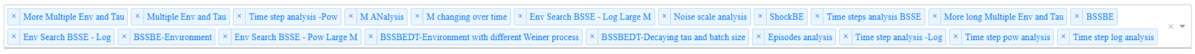


Figure 7.5: Sample plots generated by MLFlow.

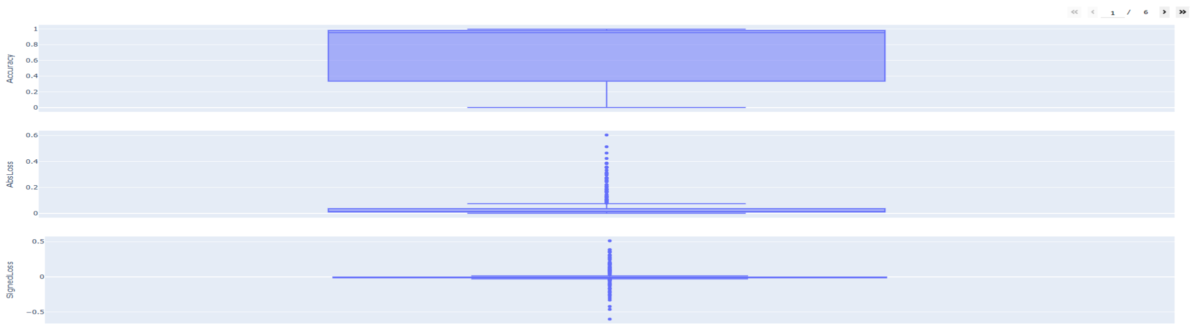
statement. The following features are supported in the implemented dash framework in addition to the ones supported by MLFlow

- Selecting multiple experiments to conduct experiments
- Visualizing summary statistics on the filtered experiments
- Visualizing losses in any particular experiment.
- Exporting the datatable to a comma separated value file.

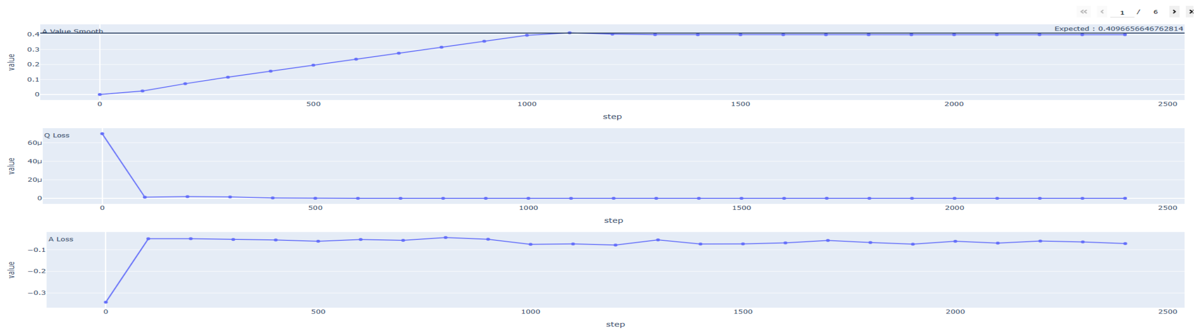
Some screenshots of the application are shown in Figures 7.6 and 7.7



(a) Filter experiments

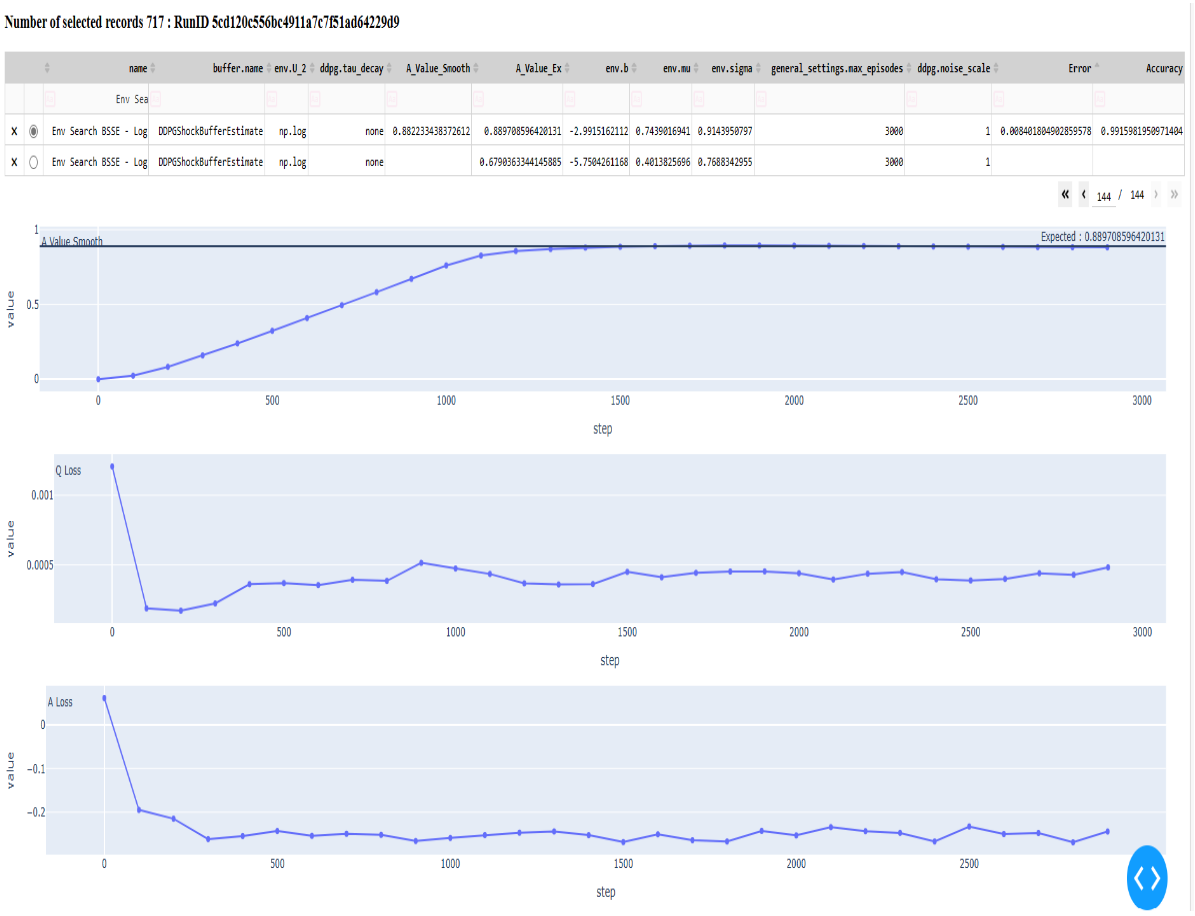


(b) Summary statistics



(c) Loss plots

Figure 7.6: Dashboard screenshots



Number of selected records 717 : RunID 5cd120c556bc4911a7c7f51ad64229d9

		name	buffer.name	env.U_2	ddpg.tau_decay	A_Value_Smooth	A_Value_Ex	env.b	env.mu	env.sigma	general_settings.max_episodes	ddpg.noise_scale	Error	Accuracy
		Env Sea												
X	◉	Env Search BSSE - Log	DDPGShockBufferEstimate	np.log	none	0.882233438372612	0.889708596420131	-2.9915162112	0.7439016941	0.9143950797	3000	1	0.008401804902859578	0.9915981950971404
X	○	Env Search BSSE - Log	DDPGShockBufferEstimate	np.log	none		0.6790363344145885	-5.7504261168	0.4013825696	0.7688342955	3000	1		

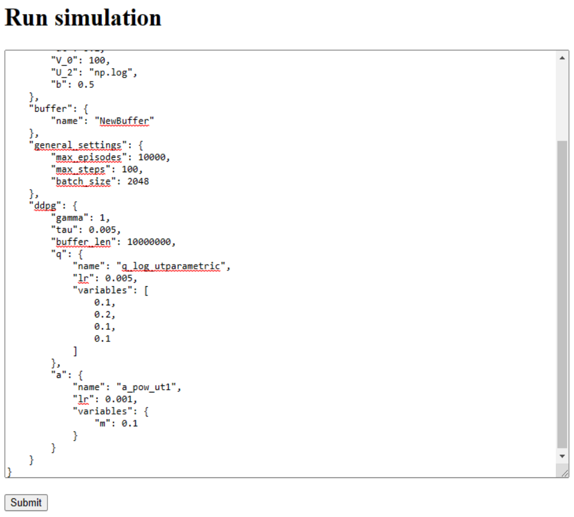
« ‹ 144 / 144 › »

Figure 7.7: Sample plots generated by MLFlow - Dashframework

7.3.2 Deployment

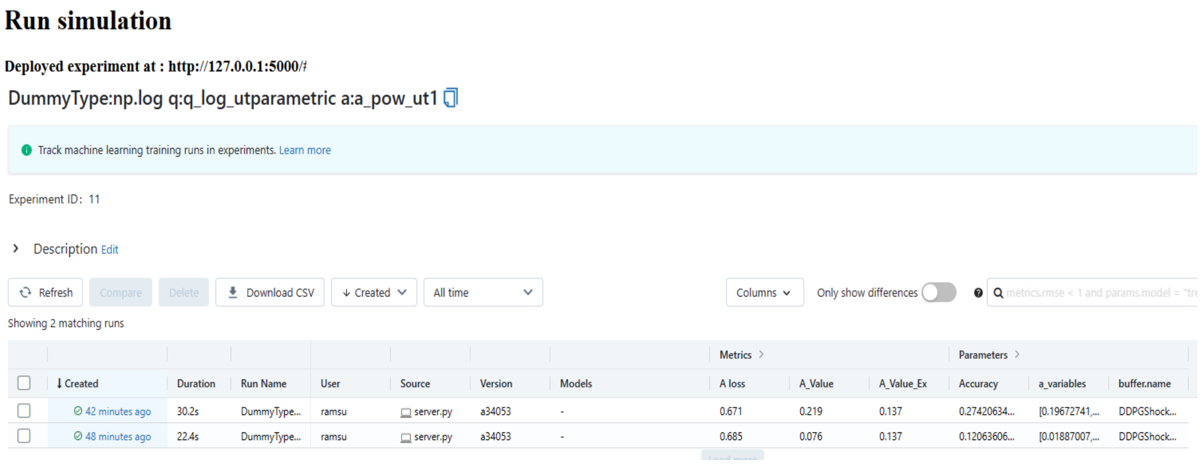
To deploy the experiments, we implemented a framework based on the fastapi [31] package. A Fastapi server is spawned using uvicorn - which is an ASGI web server implementation for Python [33]. One can deploy a simple experiment or even a configuration of networks using the framework described in the next section 7.4.

We show an experiment spawned and the resulting feedback in Figures 7.8 and 7.9.


Run simulation

```
        "V_0": 100,
        "U_2": "np.log",
        "b": 0.5
    },
    "buffer": {
        "name": "NewBuffer"
    },
    "general_settings": {
        "max_episodes": 10000,
        "max_steps": 100,
        "batch_size": 2048
    },
    "ddpg": {
        "gamma": 1,
        "tau": 0.005,
        "buffer_len": 10000000,
        "q": {
            "name": "q_log_utparametric",
            "lr": 0.005,
            "variables": [
                0.1,
                0.2,
                0.1,
                0.1
            ]
        },
        "a": {
            "name": "a_pow_ut1",
            "lr": 0.001,
            "variables": {
                "m": 0.1
            }
        }
    }
}
```

Figure 7.8: Deployment step


Run simulation

Deployed experiment at : http://127.0.0.1:5000/#

DummyType:np.log q:q_log_utparametric a:a_pow_ut1

Track machine learning training runs in experiments. Learn more

Experiment ID: 11

Description Edit

Showing 2 matching runs

								Metrics			Parameters		
	Created	Duration	Run Name	User	Source	Version	Models	A loss	A_Value	A_Value_Ex	Accuracy	a_variables	buffer.name
	42 minutes ago	30.2s	DummyType...	ramsu	server.py	a34053	-	0.671	0.219	0.137	0.27420634...	[0.19672741,...	DDPGShock...
	48 minutes ago	22.4s	DummyType...	ramsu	server.py	a34053	-	0.685	0.076	0.137	0.12063606...	[0.01887007,...	DDPGShock...

Figure 7.9: Deployment pipeline

7.4 Hyperparameter Tuning Framework

One of the key requirements of the project is to conduct experiments across a large set of parameters. Some of the parameters could be environmental settings, such as the model parameters of the environment, the number of time steps to conduct an experiment, and the utility functions we need to try.

In addition, the hyper parameters of the algorithm itself could be a number of settings. For example the parametric functions for the critic and actor, the learning rates used for the optimizing critic and actor loss, the target network learning rate τ among many others can all be changed leading to an exponential number of configurations.

However in this section, we talk about a framework we implemented - a configuration based approach to iterate through the different configurations of the optimization. Every base configuration specified in Listing 7.1 can be tuned with a tuning dictionary. A sample tuning is provided in Listing 7.2.

Listing 7.2: Tuning dictionary

```
1  "tune": {
2   "buffer.name": {
3    "list": [ "DDPGShockBuffer","DDPG" ]
4   },
5   "env.dt" :
6   {
7    "list" : [ 0.01,0.02,0.1,0.2]
8   },
9   "ddpg.max_episodes" :
10  {
11     "low" : 2000, "high" : 20000, "step" : 100
12  },
13  "env.V_0"
14  {
15   "set" : "np.random.rand()*2" ,"size" : 10
16  },
17  "group":[
18   {
19     "env.mu": 0.019536,"env.sigma": 0.377183
20   },
21   {
22     "env.b": -8.381621,"env.sigma": 0.57196
23   }
24  ],
25  "group_2" :
26  [
27   {
28   "ddpg.q.name" : "q_log_utparametric", "env.U_2" : "np.log"
29   },
30   {
31     "ddpg.q.name" : "q_pow_utparametric","env.U_2" : "pow"
32   } ]}
```

In general every parameter specified in the configuration can be tuned. For example in the dictionary specified in Listing 7.2, one of the ways the experiment can be iterated is over the time step, env.dt [See line 7]. Some of the ways the parameter can be tuned are specified below.

- **list** - By specifying list, we iterate over a set of specified values. Continuing on the example 7.2 env.dt is iterated over the list 0.01,0.02,0.1,0.2 .
- **range** - By specifying the keyword,"low", and the optional parameters, "high" (default

1) and "step" (default 1) we specify a range [low,high,step] where we do a grid search for the hyper parameter from low to high incremented at each iteration by step.

- **set** - Set helps to evaluate an expression for the hyper parameter. The expression is any valid python based expression that returns a value compliant to the variable that it is being assigned to. In the Listing 7.2, we set $env.V_0$ the initial wealth by calling the expression "np.random.rand()*2" , 10 times.

- **group*** - While tuning hyper parameters, one can specify the level of granularity to iterate over by introducing "group" elements. If we do not have this element and tune over n hyper parameters, then the total number of configurations generated would be $\prod_{i=1}^{n} C_i$ where C_i is the number of values to be tested for a hyper parameter i. However, it makes sense to bunch a set of hyper parameter configurations together and iterate over another set of hyper parameters, along with them.

  For example, consider that we have 2 utility functions to be tested - power and log. And let the corresponding Q-value functions to be experimented are $Q^{\theta_i^p}$ and $Q^{\theta_j^l}$ respectively (where $1 \leq i \leq N_p$ and $1 \leq j \leq N_l$; N_p and N_l are positive integers). It would make sense to create a group with the power utility function and the different Q-value functions $Q^{\theta_i^p}$, and then create another group with the log utility function and the different Q-value functions $Q^{\theta_j^l}$, instead of iterating over all possible configurations. This is possible with the "group" construct. Group elements should start with the prefix "group". The number of possible configurations would be $\prod_{i=1}^{n} G_i$ where G_i is the number of configurations for a group G_i. Note that $\prod_{i=1}^{n} G_i \leq \prod_{j=1}^{N} C_j$. An element stated explicitly (as in the examples for list, set and range) is a group of one element trivially. Again within a group, one could recursively tune for list, set and range as discussed before and even have nested group variables inside it.

8 Results

In this chapter, we summarize our results comparing different DDPG algorithms specified before in Chapters 4 , 5, and 6.

We first provide an overall view of the results across all the configurations we have tested. We then drill down to the performance of the algorithm under different market conditions. Some of the metrics we tested in this case are

- Utility functions of wealth for the investor.
- Mean return (μ) and volatility (σ) of the risky asset.
- Risk aversion factor b of the investor in the case of power utility.
- Number of time steps (time discretization) in an episode

We proceed then to do a hyper parameter search on the tunable parameters in the different algorithms and then make inferences on the optimal hyper parameter configuration to improve our performance. Some of the hyper parameters we considered are

- τ - target learning rate
- Noise scale to the exploration part in DDPG.
- Number of episodes
- Batch size
- Number of grid points on DDPG Estimate
- Batch size $|B_p| = m$, for the log returns in DDPG Shock Buffer

We found out that in particular, Shock Buffer (Chapter 5) and Estimates (Chapter 6) were robust and as we increased the number of sampled log returns of the shock m and the number of partitions (also m) respectively, the accuracy also significantly increased. We repeated the experiments by increasing the number of time steps and across different model parameters for the robust version and found out that the results were stable.

8.1 Performance Analysis

We define four benchmarks to summarize the performance analysis of our experiments.

- **Accuracy** as defined by the following equations

$$Error_{actual} = \frac{|\pi^* - a^{\phi}|}{\pi^*} \tag{8.1}$$

$$Error_{Normalized} = \begin{cases} Error_{actual} & Error_{actual} \leq 1 \\ 1 & otherwise \end{cases} \tag{8.2}$$

$$Accuracy = 1 - Error_{Normalized}. \tag{8.3}$$

- **Signed Loss** as defined by

$$SignedLoss = \pi^* - a^{\phi}. \tag{8.4}$$

- **Absolute Loss** as defined by

$$AbsoluteLoss = |\pi^* - a^{\phi}|. \tag{8.5}$$

- **Variance** as defined by

$$Variance = \frac{1}{100}\Sigma_{i=0}^{99} \left(a^{\phi^{(n-i)}} - \frac{1}{100}\Sigma_{j=0}^{99} a^{\phi^{(n-j)}} \right)^2, \tag{8.6}$$

  where n is the last recent update to the algorithm

Going forward in this chapter, we refer to DDPG Functions, DDPG Shock Buffer and DDPG Estimates as DDPG, Shock Buffer and Estimates, respectively.

Table 8.1: Overview configuration

Environment Parameter	**Sampled values**	**DDPG Parameter**	**Values**	**DDPG Parameter**	**Values**
$\mu \in [0,1]$	[0.07,0.955]	Version	DDPG, Shock Buffer, Estimates	Batch Size	1024
$\sigma \in [0,1]$	[0.1,1.4]	Grid Points	[8,1024]	Batch Size Growth	None and Linear
Δt	[0.01,0.2]	Shock Buffer Size	[8,1024]		
$v_0 \in (0,1]$	[0.1,0.9]	Noise Decay	Linear and None		
Utility	power and log	Noise Scale	[0.1,5]		
$b \in [-10,1)\backslash\{0\}$	[-9.0,0.95]	τ	5.10^{-4}		
T	1	τ decay	Linear and None		
r_c	0	Buffer Length	[10^4, 10^5]		

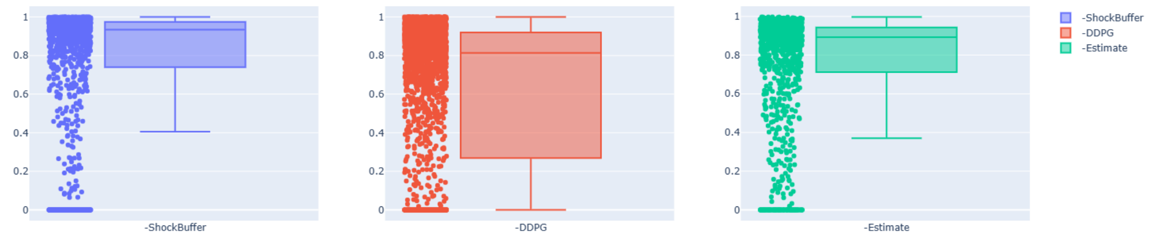
8.1.1 Accuracy and Losses - Overview

We conducted a variety of experiments, with different time steps and environment parameters. The different configuration of these parameters are shown in Table 8.1. In the table, Grid Points is only applicable to DDPG Estimate, and Shock Buffer Size is only applicable to DDPG Shock Buffer.

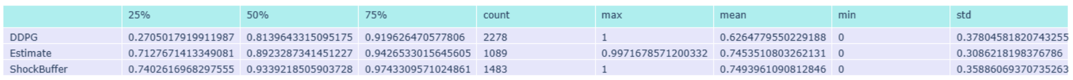
In Figure 8.1, we filtered experiments, with accuracy at least greater than 0% and removed the simulations where gradients exploded. The number of experiments were different in different versions for many reasons. Some experiments (noise scale analysis) were only conducted in 1 version of the solution (DDPG, Shock Buffer or Estimates). We started off with DDPG, and developed the other versions later so we had more experiments on DDPG compared to the rest.

From Figure 8.2, it appears that Shock Buffer and Estimates performed (in terms of accuracy) much better compared to DDPG. All quantiles as well as the mean accuracy are higher.

Comparing Shock Buffer and Estimates, we see that Shock Buffer generally performs better. The standard deviation of accuracy for Shock Buffer is however higher compared to Estimates. In our experiments we noted that Shock Buffer simulations were quite "noisy" in convergence compared to Estimates. The sample variance in the optimal action in the experiments we conducted were lower for Estimates (2.8e-7) compared to Shock buffer

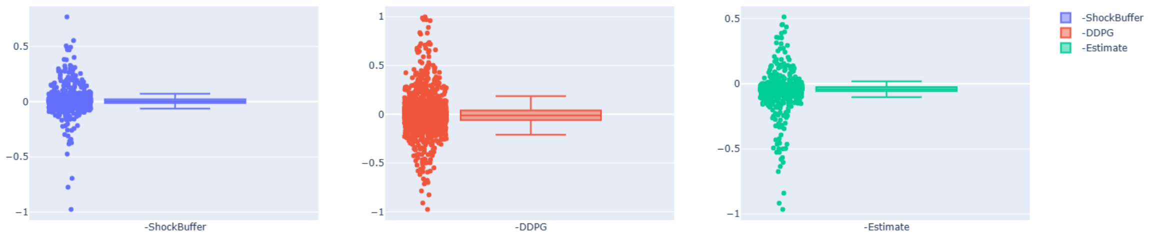


(a) Box plot comparison for all DDPG algorithms - Accuracy

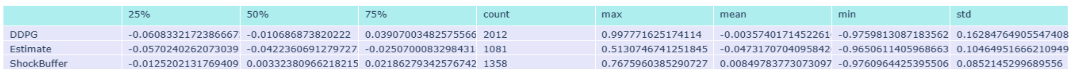


	25%	50%	75%	count	max	mean	min	std
DDPG	0.2705017919911987	0.8139643315095175	0.919626470577806	2278	1	0.6264779550229188	0	0.37804581820743255
Estimate	0.7127671413349081	0.8923287341451227	0.9426533015645605	1089	0.9971678571200332	0.7453510803262131	0	0.3086218198376786
ShockBuffer	0.7402616968297555	0.9339218505903728	0.9743309571024861	1483	1	0.7493961090812846	0	0.35886069370735263

(b) Tabular description of algorithms (accuracy)



(c) Box plot comparison for all DDPG algorithms - signed loss



	25%	50%	75%	count	max	mean	min	std
DDPG	-0.0608332172386667	-0.010686873820222	0.03907003482575566	2012	0.997771625174114	-0.0035740171452261	-0.9759813087183562	0.16284764905547408
Estimate	-0.0570240262073039	-0.0422360691279727	-0.0250700083298431	1081	0.5130746741251845	-0.0473170704095842	-0.9650611405968663	0.10464951666210949
ShockBuffer	-0.0125202131769409	0.00332380966218215	0.02186279342576742	1358	0.7675960385290727	0.00849783773073097	-0.9760964425395506	0.0852145299689556

(d) Tabular description of algorithms (signed loss)

Figure 8.1: Performance analysis

(7.8e-6) at the 50th percentile.

Looking at the box plot of the signed loss in Figure 8.1, we see an interesting pattern for Estimates. The loss is mostly negative. which means the calculated optimal action value is lower than the optimal value. Here it seems like Estimates yields biased values for a^* compared to the other versions which have a median signed loss of approximately 0.

Further, we plotted the mean absolute loss across bins of the expected optimal action constrained between 0 and 1 in Figure 8.2.

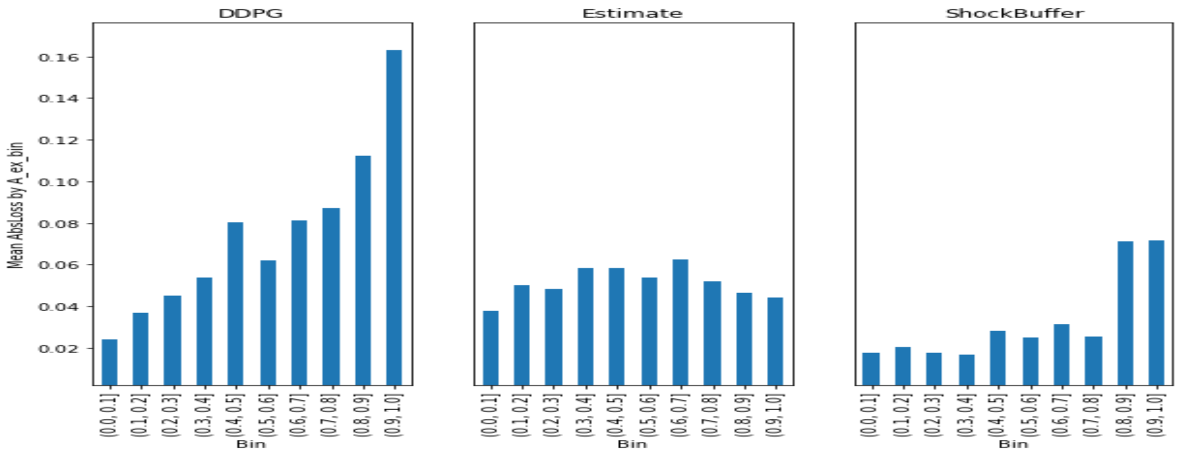


Figure 8.2: Absolute loss analysis

In DDPG and Shock Buffer we see that as the true value of the optimal action, a^* increases, the absolute losses are much higher compared to the other intervals. However for Estimates, the absolute losses seem to be more robust in the interval [0,1].

8.2 Environment search

Next we also tried to do an environment search across different model parameters and across different utility functions. The procedure for selecting the parameters is listed in Algorithm 9

Algorithm 9 Generate environment parameters procedure

```
Require:
    N                                                   ▷ Number of observations

1: $i \leftarrow 0$
2: while $i < N$ do
3:     $\mu \leftarrow UniformDistribution(0,1)$
4:     $\sigma \leftarrow UniformDistribution(0,1)$
5:     $b \leftarrow UniformDistribution(-10,1)$
6:     $a^*_{log} \leftarrow \mu/\sigma^2$
7:     $a^*_{pow} \leftarrow \mu/((1-b)\sigma^2)$
8:     if $a^*_{log} < 1$ and $a^*_{pow} < 1$ then
9:         Add environment parameters $\{\mu,\sigma,b\}$
10:        $i \leftarrow i+1$
11:    end if
12: end while
```

We sampled uniform random variables in the range [0,1] for μ and σ. For b we used a uniform distribution in the range [-10,1]. We also made sure that, expected optimal action, a^* was within the range [0,1]

The configuration for these experiments are shown in Table 8.7

The results are shown in Figures 8.3, 8.4 and 8.5. The lower triangle in these plots is empty because of the way the parameters were chosen with the constraint on optimal action values, a^* being in [0,1].

We see that Shock Buffer and Estimates have better accuracy in most of the regions in the heatmap. We also observe that there are no particular regions in the grid, where the accuracy is markedly different (for any of the algorithms). We do note that when μ is small, the accuracy is lower. However, the expected optimal action for both utility functions are directly proportional to μ. Hence, when the optimal action value is also very small, the accuracy may be significantly impacted even for minor perturbations from the theoretical values.

Table 8.2: Configuration - Environment Analysis

Environment Parameter	**Sampled values**	**DDPG Parameter**	**Values**	**DDPG Parameter**	**Values**
$\mu \in [0,1]$	**[0.07,0.955]**	Version	DDPG, Shock Buffer, Estimates	Batch Size	1024
$\sigma \in [0,1]$	**[0.1,1.0]**	Grid Points	20 (Estimate)	Batch Size Growth	None
Δt	0.2	Shock Buffer Size	8 (Shock Buffer)		
v_0	1	Noise Decay	Linear		
Utility	power and log	Noise Scale	1		
$b \in [-10,1)\backslash\{0\}$	[-9,0.95]	τ	5.10^{-4}		
T	1	τ decay	Linear		
r_c	0	Buffer Length	10^4		

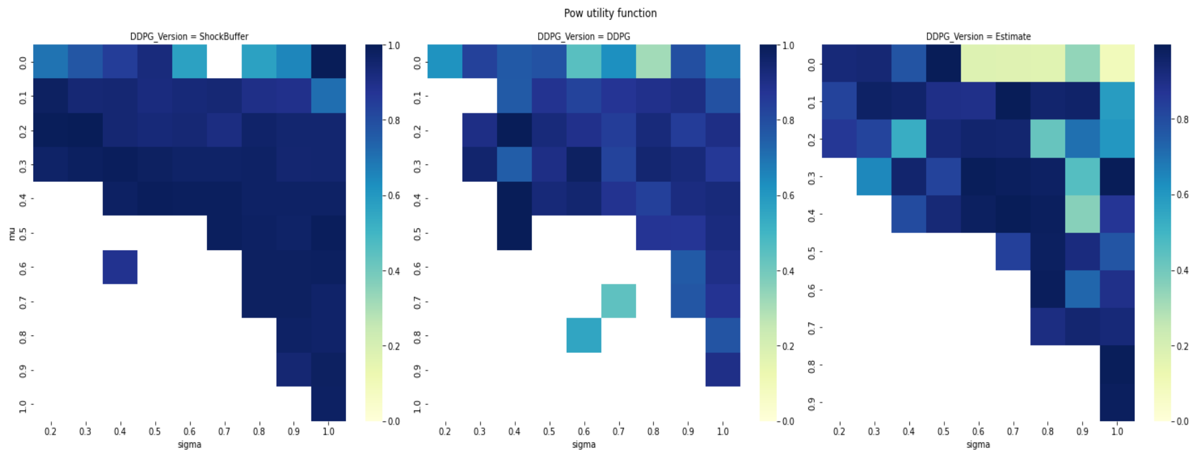


Figure 8.3: Accuracy heat map - Power utility

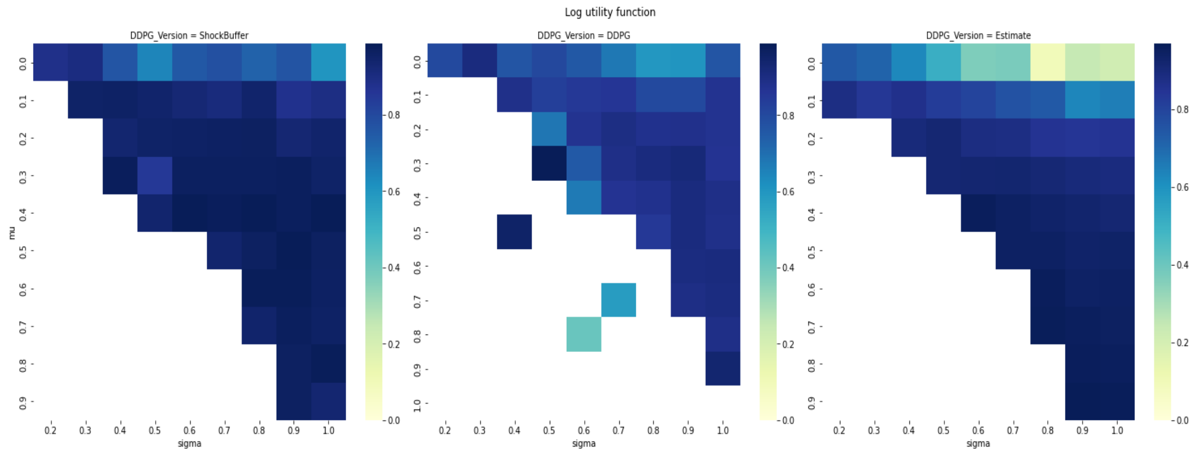


Figure 8.4: Accuracy heat map - Log utility

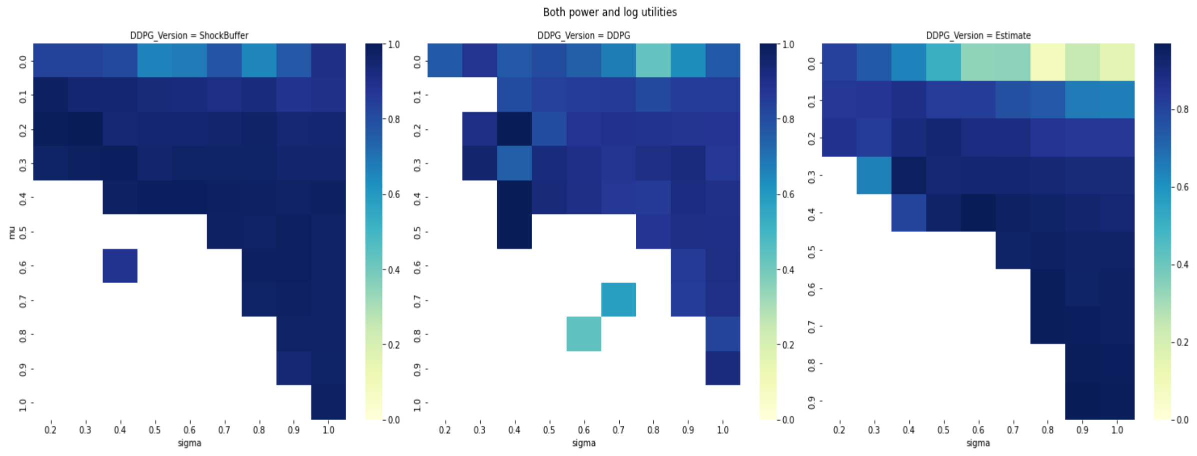


Figure 8.5: Accuracy heat map for both utility functions

Table 8.3: Configuration - Environment Analysis - Risk aversion

Environment Parameter	**Sampled values**	**DDPG Parameter**	**Values**	**DDPG Parameter**	**Values**
$\mu \in [0,1]$	[0.07,0.955]	Version	DDPG, Shock Buffer, Estimates	Batch Size	1024
$\sigma \in [0,1]$	[0.1,1.0]	Grid Points	20 (Estimate)	Batch Size Growth	None
Δt	0.2	Shock Buffer Size	8 (Shock Buffer)		
v_0	1	Noise Decay	Linear		
Utility	power	Noise Scale	1		
$\mathbf{b \in [-10,1) \backslash \{0\}}$	**[-9,0.95]**	τ	5.10^{-4}		
T	1	τ decay	Linear		
r_c	0	Buffer Length	10^4		

8.2.1 Risk Aversion analysis for Power Utility

We also analyzed with a number of values for b, the risk aversion factor for the power utility function. We expect that as b approaches 1, the absolute losses should increase as the true optimal allocation π^* also increases.

The configuration for these experiments are shown in Table 8.3 and the results are shown in Figure 8.6.

8.3 DDPG Hyper Parameter Analysis

8.3.1 τ - Target Parameters Learning Rate

The target network is updated periodically with the parameters from the main network. The rate, τ at which the target network is updated is controlled by a hyperparameter known as the target network learning rate. A low value for τ means that the target network parameters are updated slowly, which can help to stabilize the learning process. On the other hand, a high value for τ means that the target network parameters are updated quickly, which can lead to faster convergence but may also lead to instability in the learning process.

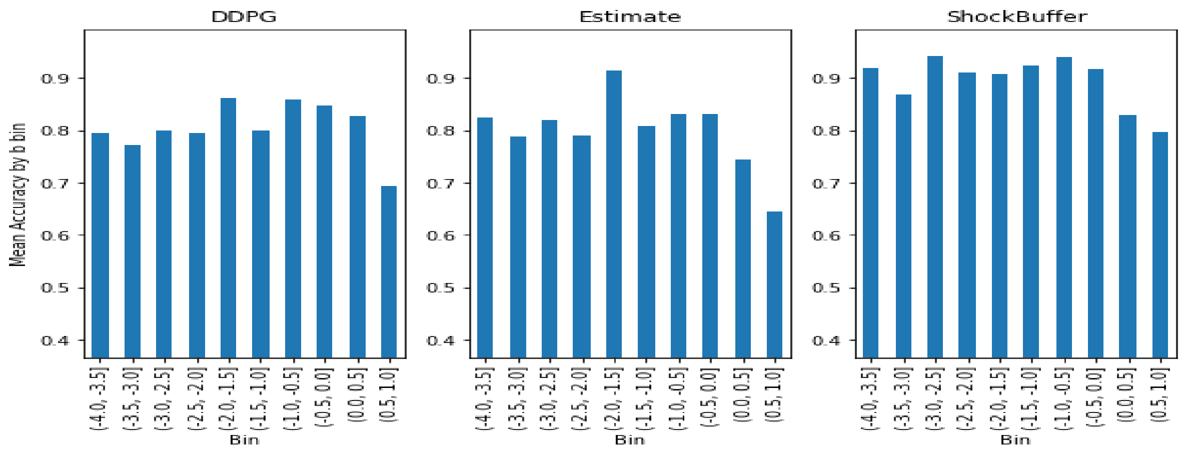


Figure 8.6: Risk aversion analysis

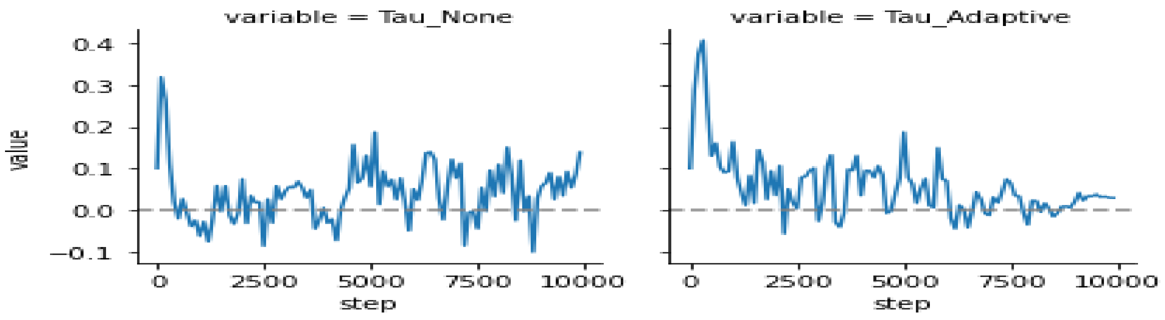


Figure 8.7: τ analysis for 2 similar experiments

Adaptive tau

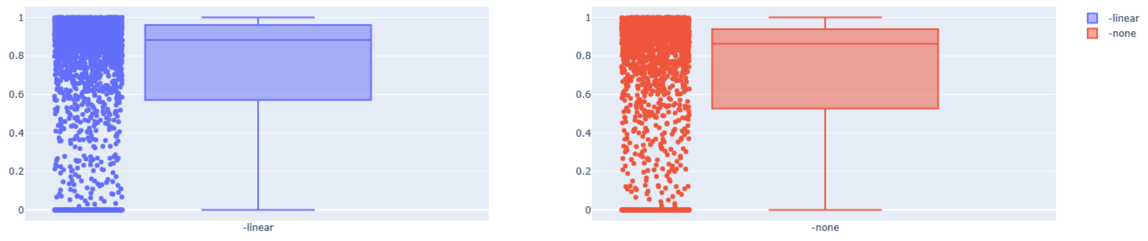
We can set τ to be high for faster convergence in the beginning of the experiments while reducing it gradually over time to stabilize the process. Sample plots from our experiments showing the representative effect is shown in the Figure 8.7. In the Tau_Adaptive case, the initial value of τ is multiplied in by the factor $(TotalEpisodes - currentEpisode)/TotalEpisodes$ before each update

Table 8.4: Configuration - Tau Analysis

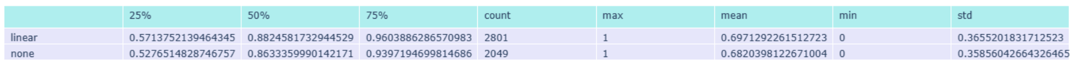
Environment Parameter	**Sampled values**	**DDPG Parameter**	**Values**	**DDPG Parameter**	**Values**
$\mu \in [0,1]$	[0.07,0.955]	Version	DDPG, Shock Buffer, Estimates	Batch Size	1024
$\sigma \in [0,1]$	[0.1,1.0]	Grid Points	20 (Estimate)	Batch Size Growth	None
Δt	0.2	Shock Buffer Size	8 (Shock Buffer)		
v_0	1	Noise Decay	Linear		
Utility	power	Noise Scale	1		
$b \in [-10,1)\backslash\{0\}$	[-10,1)	τ	5.10^{-4}		
		Tau Decay	**None and Linear**		
r_c	0	Buffer Length	10^4		

We tried to conduct experiments for studying the effects of adaptive τ based on the formula we earlier described. We present the configuration of these experiments in Table 8.4 and the results in Figure 8.8

We noted that having an adaptive policy for tau decay has a positive effect in our experiments.



(a) Box plot analysis - τ



	25%	50%	75%	count	max	mean	min	std
linear	0.5713752139464345	0.8824581732944529	0.9603886286570983	2801	1	0.6971292261512723	0	0.3655201831712523
none	0.5276514828746757	0.8633359990142171	0.9397194699814686	2049	1	0.6820398122671004	0	0.35856042664326465

(b) Table analysis - τ

Figure 8.8: τ analysis

8.3.2 Noise Process

In our experiments, to model the exploration policy while arriving at the optimal action we used the Ornstein Uhlenbeck Process [36] as stated in the original paper to generate correlated noise. In the i-th training step, the noise level X_i is updated recursively through the equation

$$X_i = (1 - \kappa\Delta_N)X_{i-1} + \sigma_N\sqrt{\Delta_N}\mathcal{N}, \tag{8.7}$$

for positive constants $\kappa, \Delta_N, \sigma_N$ and $\mathcal{N} \sim \mathcal{N}(0,1)$. In our experiments we set $\sigma_N = 0.2$ (volatility of the noise process). $\kappa = 0.15$ (rate at which X reverts to the mean) and $\Delta_N = 0.2$

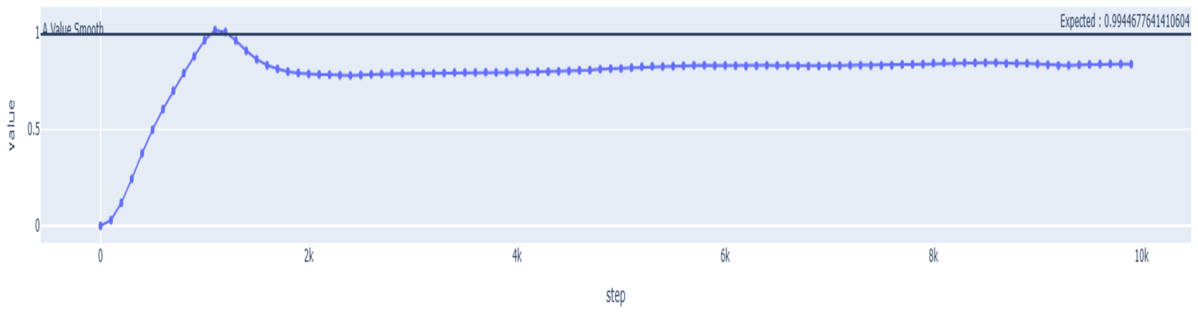
For every call to the noise object specified in Algorithm 1,we update the noise level as per (8.7). Then the noise level is scaled according to an additional hyper parameter. Morever, we have another hyper parameter that decays the whole noise level as we progress on an experiment. Finally we add the resulting noise level to the estimated optimal action, a^{Φ} which we compute at every step. This whole procedure is shown in Algorithm 10.

Algorithm 10 Noise process - Hyper parameter tuning

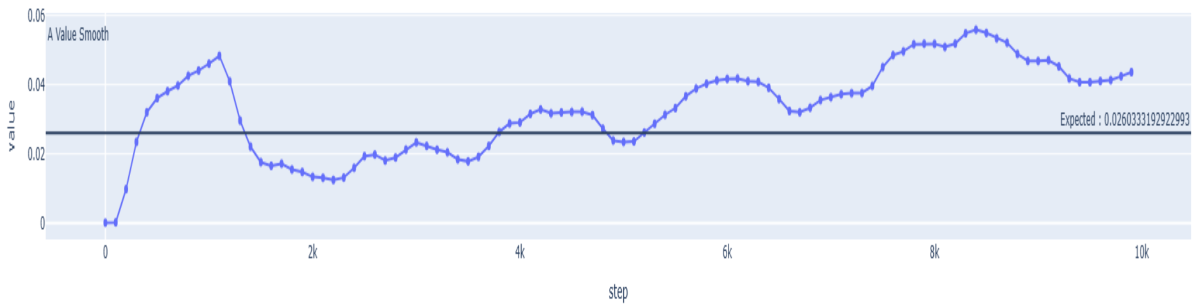
```
Require:
    $currentEpisode, TotalEpisodes$
    $scale, decay$ ▷ scale is a numeric hyperparameter, decay is a boolean hyperparameter to linearly decay the noise level
    $noiseLevel, \theta, \mu, \sigma, \Delta t$    ▷ Parameters of the OU process, $\Delta t$ is the time step
    $currentEpisode, TotalEpisodes$

1: $noiseLevel \leftarrow \big(noiseLevel * (1 - \kappa\Delta_N) + \sigma_N\sqrt{\Delta_N} * StandardNormalVariable()\big)$
2: Store $noiseLevel$
3: $decayFactor \leftarrow 1$
4: if $decay = True$ then
5:     $decayFactor = (TotalEpisodes - currentEpisode)/TotalEpisodes$
6: end if
7: return $decayFactor * scale * noiseLevel$
```

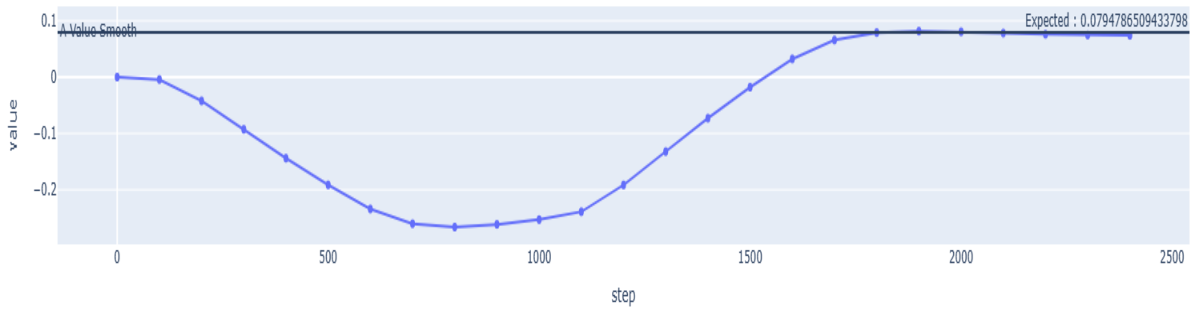
By increasing the *scale* hyper parameter, we can control the exploration part in seeking the optimal action, and by setting *decay* to True, we can achieve better convergence (while also allowing for high exploration in the beginning of the experiment). These effects are illustrated in Figure 8.9.



(a) Noise factor 0.1



(b) Noise factor 5



(c) Adaptive noise

Figure 8.9: Noise scale analysis to illustrate effects on convergence and accuracy

We conducted a batch of experiments to study the effect of noise scale on the performance, the configuration of the experiments is shown in Table 8.5 and the corresponding results are shown in Figure 8.10.

Table 8.5: Configuration - Noise scale Analysis

Environment Parameter	**Sampled values**	**DDPG Parameter**	**Values**	**DDPG Parameter**	**Values**
$\mu \in [0,1]$	[0.02,0.42]	Version	Shock Buffer	Batch Size	1024
$\sigma \in [0,1]$	[0.35,0.98]	Grid Points	-	Batch Size Growth	None
Δt	0.2	Shock Buffer Size	8 (Shock Buffer)		
v_0	1	Noise Decay	Linear		
Utility	power, log	**Noise scale**	**{0.1,1,2,5}**		
$b \in [-10,1)\backslash\{0\}$	[-8.1,0.73]	τ	5.10^{-4}		
T	1	τ decay	Linear		
r_c	0	Buffer Length	10^4		

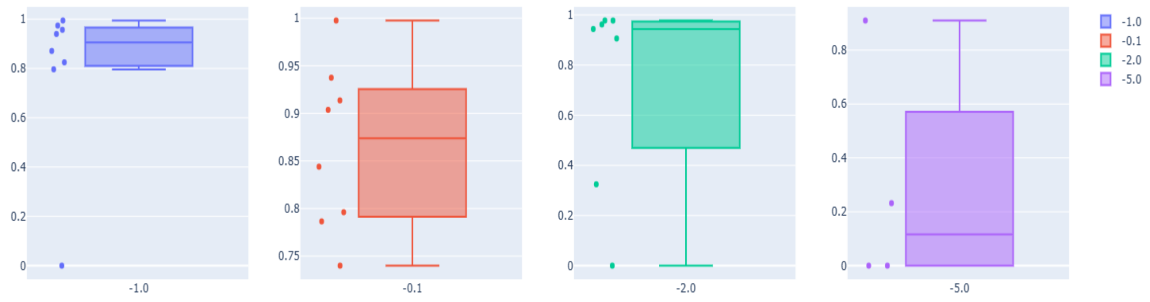
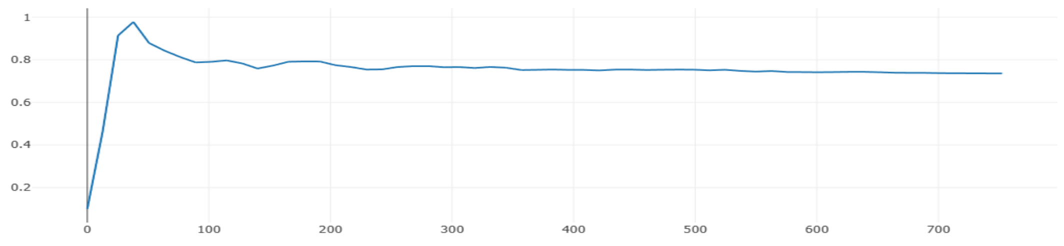
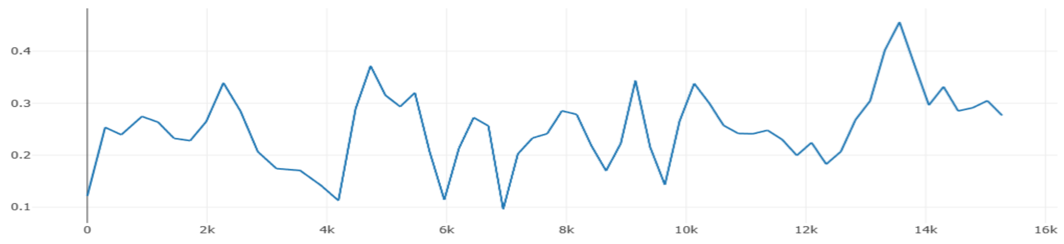


Figure 8.10: Noise scale Analysis

We noted that in our experiments having a higher or lower noise scale to increase/decrease our exploration generally did not improve our performance for the configuration we tested.



(a) Good convergence



(b) No Convergence

Figure 8.11: Episode hyperparamter - to show the solution either converging or not in our experiments

8.3.3 Episodes

Typically, we would expect that as we increase the number of episodes, the algorithm should converge. If we observe that the algorithm does not converge it could be mainly because of 2 reasons in our set up (assuming τ is not decayed).

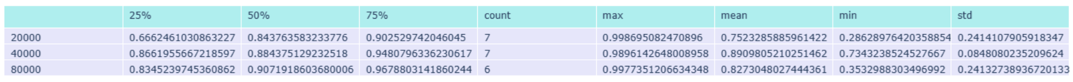
- Approximation for the expectation in the equation (5.1) is too inaccurate (which we discussed in depth in earlier Chapters 5 and 6)
- The Bellman equation cannot be fitted in a stable manner

Table 8.6: Configuration - Number of episodes analysis

Environment Parameter	**Sampled values**	**DDPG Parameter**	**Values**	**DDPG Parameter**	**Values**
$\mu \in [0,1]$	[0.016,0.62]	Version	Shock Buffer and Estimates	Batch Size	1024
$\sigma \in [0,1]$	[0.27,0.98]	Grid Points	20 (Estimates)	Batch Size Growth	None
Δt	0.2	Shock Buffer Size	8 (Shock Buffer)	**Episodes**	$\mathbf{\{2.10^4, 4.10^4, 8.10^4\}}$
v_0	1	Noise Decay	Linear		
Utility	power, log	Noise Scale	1.0		
$b \in [-10,1)\backslash\{0\}$	[-7,0.6]	τ	5.10^{-4}		
T	1	τ decay	Linear		
r_c	0	Buffer Length	10^4		

When we do not decay τ, or noise over time, we see both good convergence and poor convergence in our experiments. This is shown in Figure 8.11 - all other parameters of the 2 experiments except μ and σ were identical and yet the convergence was quite different.

We conducted experiments to analyse the effects of the number of episodes. The configuration of these experiments are provided in Table 8.6 and the results in Figure 8.12.



	25%	50%	75%	count	max	mean	min	std
20000	0.6662461030863227	0.843763583233776	0.902529742046045	7	0.998695082470896	0.7523285885961422	0.28628976420358854	0.2414107905918347
40000	0.8661955667218597	0.884375129232518	0.9480796336230617	7	0.9896142648008958	0.8909805210251462	0.7343238524527667	0.0848080235209624
80000	0.8345239745360862	0.9071918603680006	0.9678803141860244	6	0.9977351206634348	0.8273048027444361	0.3532988303496992	0.24132738936720133

(a) Table analysis - number of episodes

Figure 8.12: Episodes analysis

We see that when we increase the number of episodes, accuracy does increase (albeit only marginally). For Estimates we did not see any improvement as we increase the number of episodes. When we inspected individual plots, we find out that convergence (even if not to the optimal action) happens fairly early during training.

8.4 Robust Simulations

After our initial experiments, we observed 2 key points.

- Increasing the number of episodes only marginally increases the accuracy. We observed the Q loss decreased with episodes but the optimal action was not necessarily maximized.
- Shock Buffer and Estimates performed better in almost all aspects compared to DDPG.

So coming up with more accurate expectations in Equation (5.1) helped a long way for having robust simulations. We used this idea and tried to build better estimates by the following methods.

8.4.1 Estimates

We conducted 2 kinds of experiments.

- We created 2 sets of partitions - coarse and fine over $-\inf, \inf$. We used the fine ones from -0.5 to 0.5 roughly and coarse ones outside of that interval. The reason behind this idea was to build different (and better) partition sets for estimates that would better converge to the expected value.
- *Adaptive partition strategy*: We started off with very fine partitions (around 200) to begin with and then increased over time.

However we noted that both these experiments did not improve the accuracy by much. Then we decided on a brute force solution and merely increased the number of grid points, i.e "m", to a very high number - increased from 20 to 1000. We kept the number of episodes to 2500 as we observed that convergence happens rapidly in Estimates. We also built our solution on GPUs to help us cope with the memory constraints such a configuration would entail.

8.4.2 Shock Buffer

We followed a very similar strategy for Shock Buffer. However, as we observed that Shock Buffer took longer to converge, we increased the number of episodes to 6000, while increasing the mini batch size $m = B_p$ for log returns from 20 to 1000.

Table 8.7: Configuration - M Analysis

Environment Parameter	**Sampled values**	**DDPG Parameter**	**Values**	**DDPG Parameter**	**Values**
$\mu \in [0,1]$	[0.07,0.955]	Version	DDPG, Shock Buffer, Estimates	Batch Size	1024
$\sigma \in [0,1]$	[0.1,1.0]	Grid Points	**1000** (Estimates)	Batch Size Growth	None
Δt	0.2	Shock Buffer Size	**1000** (Shock Buffer)	Episodes	**Estimates: 2500 Shock Buffer: 6000**
v_0	1	Noise Decay	Linear		
Utility	power and log	Noise Scale	1		
$b \in [-10,1)\backslash\{0\}$	[-9.0,0.95]	τ	5.10^{-4}		
T	1	τ decay	Linear		
r_c	0	Buffer Length	10^4		

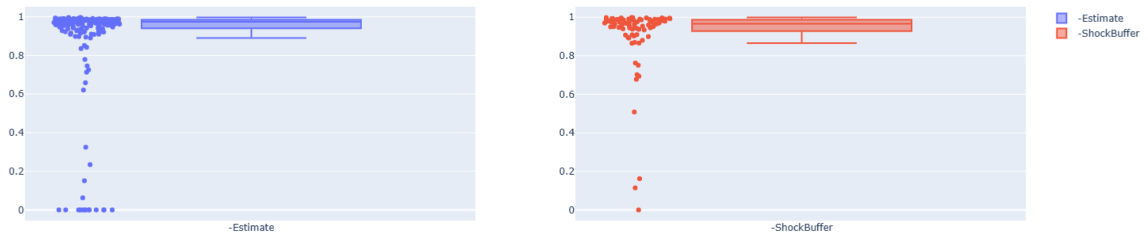
8.4.3 Results

We then ran these configurations on both power and log utility functions and repeated the experiment under different market parameters. We display the configuration of these experiments in Table 8.7 and the results in Figure 8.13.

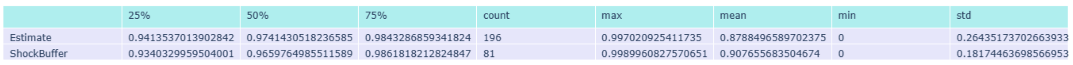
We observed that this version of both these algorithm performed comparatively better, even we decreased Δt from 0.2 to 0.01. The results for $\Delta t = 0.01$ (other configuration parameters are the same) are shown in Figure 8.14

These results are much better than our initial analysis. It does appear that Shock Buffer is better compared to the Estimates. However, we also note that Estimates is much faster (around 3 times faster) than Shock Buffer in the above set up. Further, we notice that as we keep increasing the number of grid points, the accuracy of Estimates also increases. Hence, Estimates, in a sense, can be regarded as equivalent to the Shock Buffer.

Another interesting pattern we found was that Estimates performed much better for log utility functions than for power utility functions. However, we could not find such a pattern for Shock Buffer. This pattern is captured in Figure 8.15 and 8.16



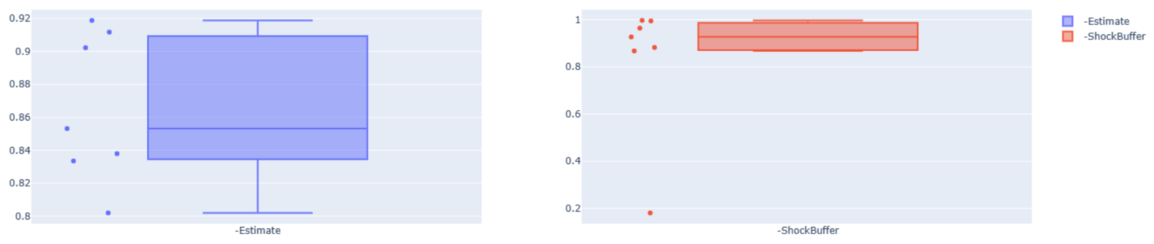
(a) Box plot analysis - m



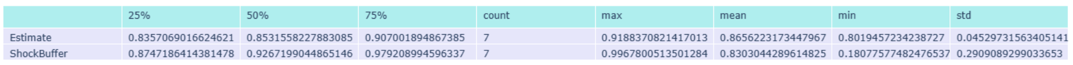
	25%	50%	75%	count	max	mean	min	std
Estimate	0.9413537013902842	0.9741430518236585	0.9843286859341824	196	0.997020925411735	0.8788496589702375	0	0.26435173702663933
ShockBuffer	0.9340329959504001	0.9659764985511589	0.9861818212824847	81	0.9989960827570651	0.907655683504674	0	0.18174463698566953

(b) Table analysis estimates - m

Figure 8.13: m analysis



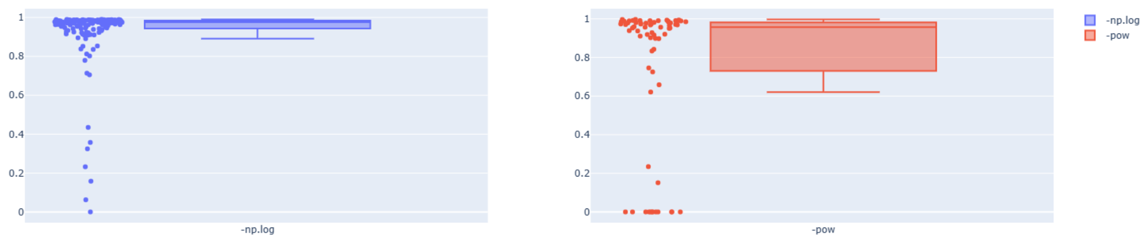
(a) Box plot analysis m



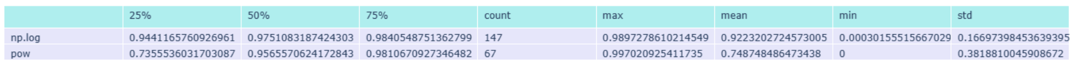
	25%	50%	75%	count	max	mean	min	std
Estimate	0.8357069016624621	0.8531558227883085	0.907001894867385	7	0.9188370821417013	0.8656223173447967	0.8019457234238727	0.04529731563405141
ShockBuffer	0.8747186414381478	0.9267199044865146	0.979208994596337	7	0.9967800513501284	0.8303044289614825	0.18077577482476537	0.2909089299033653

(b) Table analysis estimates - m

Figure 8.14: m analysis - Δt - 0.01



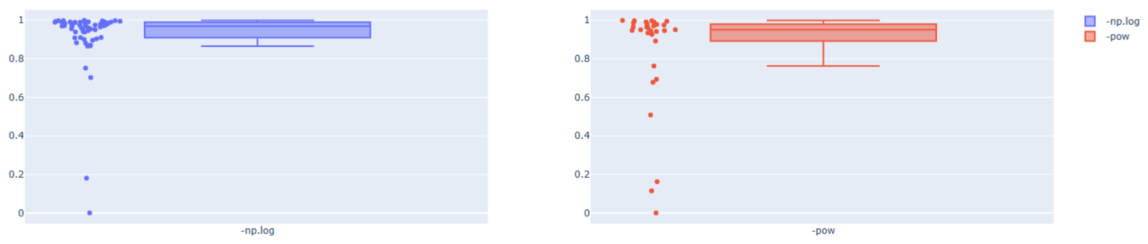
(a) Box plot Analysis - Estimates for log and power utility functions



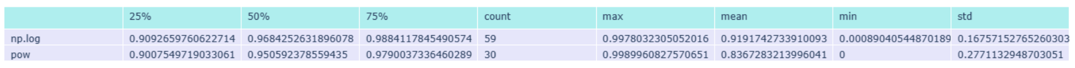
	25%	50%	75%	count	max	mean	min	std
np.log	0.9441165760926961	0.9751083187424303	0.9840548751362799	147	0.9897278610214549	0.9223202724573005	0.00030155515667029	0.16697398453639395
pow	0.7355536031703087	0.9565570624172843	0.9810670927346482	67	0.997020925411735	0.748748486473438	0	0.3818810045908672

(b) Table analysis Estimates for log and power utility functions

Figure 8.15: Utility functions analysis - Estimates



(a) Box plot Analysis - Shock Buffer for log and power utility functions



	25%	50%	75%	count	max	mean	min	std
np.log	0.9092659760622714	0.9684252631896078	0.9884117845490574	59	0.9978032305052016	0.9191742733910093	0.00089040544870189	0.16757152765260303
pow	0.9007549719033061	0.950592378559435	0.9790037336460289	30	0.9989960827570651	0.8367283213996041	0	0.2771132948703051

(b) Table analysis Shock Buffer for log and power utility functions

Figure 8.16: Utility functions analysis - Shock Buffer

9 Conclusion and Future Work

In this final brief chapter, we summarize the results we obtained in the previous Chapter 8, and recommend our best models we observed in our experiments. We finally conclude by listing some of the future work that could be done extending this framework to other class of problems and some improvements that could be made to the existing software architecture.

9.1 Conclusion

Based on the results we obtained in the previous chapter, we noticed that the most important factor for improving the accuracy and robustness of our results was to improve the estimation for the expectation defined in Equation 5.1. This can be achieved by increasing the shock buffer batch size in Shock Buffer and number of grid points in Estimates (commonly referred as m in both the methods). Increasing the number of episodes was the next most important factor that increased the accuracy, especially observed in Shock Buffer and DDPG. Adaptive τ also marginally improved the accuracy in our experiments. These results are further summarized in Table 9.1.

In this sense, we recommend both Shock Buffer, and Estimates as our preferred algorithms. The configuration described in Section 8.4 (for both methods) yielded the most promising results in our experiments. However, we also believe that one could achieve even better accuracy and convergence speed by further increasing the factor m.

Parameter	**Impact**	**DDPG Version**	**Comment**
Stable estimate for expectation (5.1)	High	Shock Buffer and Estimates	Computing stable estimates of expectation in the Bellman equation has a very positive impact on the accuracy.
Number of grid points	High	Estimates	Increasing partitions to find the expected value of future Q values has a very positive effect on accuracy.
Batch size of log returns	High	Shock Buffer	Increasing batch size of log shock returns to find the expected value of future Q values has a very positive effect on accuracy.
Batch size of state, action tuples	Medium	All	Increasing batch size increases accuracy but only up to a level. Adaptive batch size marginally improves accuracy.
τ	Medium	All	Adaptive τ marginally improves accuracy, and convergence.
Noise scale	Low	All	Did not significantly improve accuracy.
Number of episodes	Medium	All	**Estimates**: Did not impact accuracy after a rapid and an early converging level. **Shock Buffer**: Impacted the accuracy considerably.
Model parameters	None	All	Did not markedly notice domains of model parameters that impacted performance or convergence time.
Time discretization Δt	High	All	Accuracy and convergence time is proportional to Δt.

Table 9.1: Summary of results - Qualitative analysis

9.2 Future Work

Our study was very theoretical in which we generated observations from a model distribution. It was also a very simplistic problem with just 1 risky and 1 riskless asset. The utility functions we considered were also designed in such a way that the optimal allocation strategy could be explicitly derived and were independent from the time horizon in the experiments. The model distribution itself was very simple - a Black Scholes model with known mean and variance of the risky asset. The riskless rate was also assumed to be 0.

So considering all the discussed simplifications it is very obvious the tremendous scope and expansion of the current effort.

- **Model Distribution**

  First of all we can build a slightly more complex environment. For instance instead of a Black Scholes environment, we can use a Heston environment [34] where the volatility of the risky asset changes over time. There is also no limit on extending the environment by adding more parameters and increasing the complexity of the environment.

- **Empirical Data**

  We did not use actual returns that were observed in real markets to generate our optimal actions. A problem with using empirical real data is that we have only one realization. We can also build a capital market model and generate complex forward looking simulations of assets based on such a model. The problem with having such complex models is that our actor and critic may no longer be expressive and accurate enough to capture the underlying model.

- **Actor and Critic**

  Our actor and critic were actually parametrized versions of the theoretical values of Q and A values respectively. In essence, we almost supplied the exact answer to our experiments and just caliberated the parameters in those functions using DDPG. There are various improvements that can be done in this setting.

  - **Generic expected forms of the solution**

    In many cases, we cannot find out or find out only numeric solutions of Q and A value functions. They are the real applications of this setting as we can feed in forms of our expected solution without explictly stating the actual form. The challenge then will be to understand if indeed the final solution is the optimal solution and also arriving at the generic form itself may be a challenge.

  - **Generic forms with neural network**

    We can start off with the same set up as in the previous case, but infer the parameters of the generic function using deep neural networks. In addition to the usual performance related challenges, convergence time, simulations needed, architecture of the neural network and other neural network related problems such as regularization, over fitting etc. are some of the other factors one must consider.

  - **Mismatched functions**

    We can deliberately try to give mismatched functions to Q and A value functions and then understand how those functions capture information.

  - **Multi-asset setting**

    We only discussed a simple setting with 1 risky asset in this problem. We can easily extend this to a multi-asset problem with a possibly time-changing correlation matrix between them. Such a setting would further challenge the robustness and the scalability of the developed algorithms.

  - **Utility functions**

    We considered only 2 utility functions which are concave in the problem domain and functions in which the optimal action values were independent of the time axis. We can relax these rigid assumptions and can conduct further studies on many other utility functions.

- **Software Architecture**

  The proposed architecture also could be improved, with better visualization, tracking and deployment suites. Most of what we implemented were custom built and tightly coupled with what we needed. A more thorough study on each of these aspects can be made.

  - **Distributed deployment**

    We did not build any deployment pipelines over the cloud such as having a managed kubernetes cluster. Such a deployment could widely improve our abilities to conduct vast number of experiments over the problem domain

  - **DAG for tensorflow**

    We did not exploit lazy evaluation and constructing directed acyclic graphs in tensorflow to speed up our computation. This would result in a major performance improvement in terms of speed.

  - **Tensorboard**

    We used MLFLow to track our experiments. We could have experimented with other visualization tools such as tensorboard [37] and build better visualization charts.

  - **Testing**

    Our ML model is not integrated with any test suite to generate test cases and check the validity of our experiments. Going forward this could be one crucial piece to quickly prototype and implement new features while conducting regression tests [38] on the validity of existing functionalities.

List of Figures

List of Tables

Bibliography

[1] L. Fernández and M. Kschonnek. *AGI Reinforcement Learning*. Tech. rep. July 2022.

[2] N. Wiener. “Differential space”. In: *Journal of Mathematics and Physics* 2.1 (1923), pp. 131–174.

[3] M. Escobar-Anel, M. Kschonnek, and R. Zagst. “Portfolio optimization: not necessarily concave utility and constraints on wealth and allocation”. In: *Mathematical Methods of Operations Research* 95 (2022), pp. 101–140.

[4] R. Korn. *Optimal Portfolios: Stochastic Models for Optimal Investment and Risk Management in Continuous Time*. World Scientific, 1997.

[5] R. Zagst. *Lecture Notes in Investment Strategies, Winter Term*. Lecture notes. 2019.

[6] N. Branger, C. Schlag, and E. Schneider. “Optimal portfolios when volatility can jump”. In: *Journal of Banking & Finance* 32.6 (2008), pp. 1087–1097.

[7] N. Branger et al. “Optimal portfolios when variances and covariances can jump”. In: *Journal of Economic Dynamics and Control* 85 (2017), pp. 59–88.

[8] M. Escobar, S. Ferrando, and A. Rubtsov. “Dynamic derivative strategies with stochastic interest rates and model uncertainty”. In: *Journal of Economic Dynamics and Control* 86 (2018), pp. 49–71.

[9] J. Liu and J. Pan. “Dynamic derivative strategies”. In: *Journal of Financial Economics* 69.3 (2003), pp. 401–430.

[10] R. C. Merton. “Lifetime portfolio selection under uncertainty: The continuous-time case”. In: *The Review of Economics and Statistics* 51.3 (1969), pp. 247–257.

[11] R. Korn and H. Kraft. “Optimal portfolios and HJB equations with jump diffusion processes”. In: *Mathematical methods of operations research* 54.3 (2001), pp. 427–452.

[12] M. H. A. Davis and A. R. Norman. “Portfolio selection with transaction costs”. In: *Mathematics of operations research* 15.4 (1990), pp. 676–713.

[13] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. “Human-level control through deep reinforcement learning”. In: *Nature* 518.7540 (2015), pp. 529–533.

[14] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. “Language models are few-shot learners”. In: *arXiv preprint arXiv:2005.14165* (2020).

[15] Z. Ying, H. Li, H. Ren, W. Chen, and Y. Wang. “A deep reinforcement learning framework for the financial portfolio management problem”. In: *IEEE Access* 5 (2017), pp. 24084–24094.

[16] Y. Zhang, Z. C. Lipton, M. Li, and X. Li. “Application of deep reinforcement learning in stock trading strategies and stock forecasting”. In: *IEEE Transactions on Neural Networks and Learning Systems* (2020).

[17] K. Huang, N. Vishnoi, X. Huang, P. Richtarik, and J. Zou. “Market making via reinforcement learning”. In: *International Conference on Machine Learning*. PMLR. 2018, pp. 2218–2227.

[18] J. Buehler, L. Gonon, J. Teichmann, and B. Wood. “Deep hedging”. In: *Quantitative Finance* 19.8 (2019), pp. 1271–1291.

[19] A. S. Grill. “Reinforcement Learning for Dynamic Investment Strategies in Continuous Time”. Masterarbeit. Technische Universität München, 2022.

[20] *Reinforcement Learning - An Introduction*. `http://artint.info/html/ArtInt_268.html`. Accessed: 2023-02-04.

[21] Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*. 2015. URL: `https://www.tensorflow.org/`.

[22] W. contributors. *Reinforcement Learning — Wikipedia, The Free Encyclopedia*. `https://en.wikipedia.org/wiki/Reinforcement_learning`. [Online; accessed 25-February-2023]. 2023.

[23] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[24] K. Nguyen, Y. Zheng, M. Wen, and B. Li. “A review of model-free deep reinforcement learning methods”. In: *International Conference on Machine Learning and Cybernetics* (2017), pp. 768–773.

[25] L. P. Kaelbling, M. L. Littman, and A. W. Moore. “Reinforcement learning and control as probabilistic inference”. In: *Journal of artificial intelligence research* 4 (1996), pp. 237–285.

[26] C. J. Watkins and P. Dayan. “Q-learning”. In: *Machine Learning* 8.3-4 (1992), pp. 279–292.

[27] R. Bellman. “Dynamic programming”. In: *Journal of Mathematics and Mechanics* 6.1 (1957), pp. 87–105.

[28] V. R. Konda and J. N. Tsitsiklis. “Actor-critic algorithms”. In: *Advances in neural information processing systems*. 2000, pp. 1008–1014.

[29] B. Riemann. *Gesammelte Mathematische Werke und Wissenschaftlicher Nachlass*. Druck und Verlag von B.G. Teubner, 1887.

[30] P. T. Inc. *Dash - A web application framework for Python*. 2022. URL: `https://plotly.com/dash/`.

[31] S. Tiangolo. *FastAPI*. `https://github.com/tiangolo/fastapi`. 2019.

[32] R. T. Fielding. “Architectural Styles and the Design of Network-based Software Architectures”. PhD thesis. University of California, Irvine, 2000.

[33] S. Tiangolo. *UVicorn*. `https://github.com/encode/uvicorn`. 2017.

[34] S. L. Heston. “A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options”. In: *The Review of Financial Studies* 6.2 (1993), pp. 327–343. DOI: `10.1093/rfs/6.2.327`.

[35] M. Zaharia et al. *MLflow: An Open Source Platform for the Complete Machine Learning Lifecycle*. 2018. URL: `https://mlflow.org/docs/latest/index.html`.

[36] G. E. Uhlenbeck and L. S. Ornstein. “On the Theory of Brownian Motion”. In: *Physical Review* 36.3 (1930), pp. 823–841.

[37] Google. *TensorBoard*. `https://www.tensorflow.org/tensorboard`. Accessed: February 21, 2023. 2016.

[38] W. E. Howden. “A theory of regression test selection”. In: *IEEE Transactions on Software Engineering* 4.3 (1978), pp. 227–238.