



# A Reinforcement Learning Perspective To Exotics and Flows Equities Derivatives.

**A Master's Thesis Submitted to the Faculty of:**

**Science and Engineering of Sorbonne University(SU) and Institut Polytechnique de Paris(IPP)**

**by:**

**Emmanuel Gnabeyeu Mbiada**

**M2 Probability & Finance**

**In partial fulfilment of the requirements for the degree of:**

**MPhil., Msc. in Applied Mathematics and Statistics.**

**Advisors&Jury: Pr. Gilles Pagès, Pr. Mathieu Rosenbaum**

**Internship supervisors: Omar Karkar, Imad Idboufous**

**September 2024**

In addition to completing this thesis, I co-authored the below research paper (pre-publication phase) available here.

# Solving The Dynamic Volatility Fitting Problem: A Deep Reinforcement Learning Approach in Continuous State Action Spaces

Emmanuel M. Gnabeyeu * London, UK.
Omar Karkar † London, UK.
Imad Idboufous ‡ London, UK.*

September 19, 2024

**Abstract**

The volatility fitting is one of the core problems in the equity derivatives business. Through a set of deterministic rules, the degrees of freedom in the implied volatility surface encoding (parametrization, density, diffusion) are defined. Whilst very effective, this approach widespread in the industry is not natively tailored to learn from shifts in market regimes and discover unsuspected optimal behaviors. In this paper, we change the classical paradigm and apply the latest advances in Deep Reinforcement Learning(DRL) to solve the fitting problem. In particular, we show that variants of Deep Deterministic Policy Gradient (DDPG) and Soft Actor Critic (SAC) can achieve at least as-good as standard fitting algorithms. Furthermore, we explain why the reinforcement learning framework is appropriate to handle complex objective functions and is natively adapted for online learning.

**Keywords:** Volatility Fitting, Continuous Action Spaces, Stochastic and Continuous Control, Sequential Decision Making and Deep Reinforcement Learning.

## 1 INTRODUCTION

The implied volatility, which is one of the most important risk factors in the equity business, is usually encoded in a functional form with a limited set of quasi-orthogonal parameters. This functional form, often referred to as volatility parametrization, allows users to describe the dynamic properties (movements and deformations) of the implied volatility surface in an easy way and helps in generating predictive signals for pricing and risk management of equity derivatives.

The process through which the coefficients of the volatility parametrization are determined is called calibration and is often performed automatically [1] by an algorithm tailored to optimize pre-determined objective functions. For instance, most algorithms aim at matching the mid implied volatility at all expiries where there is a visible market under non-arbitrage constraints. The fitting algorithms used in the industry often account for large sets of conditions such as liquidity, presence of outlier quotes or even earnings and macro events however all those algorithms have native rigidity in them as they follow preset rules for every market conditions. By design, the "deterministic algorithms" cannot find optimal solutions outside of their definition scope and cannot re-use past accumulated experiences as every output is flashed out.

In this paper, we look at the fitting problem from the angle of Reinforcement Learning (RL). In this new paradigm, we assimilate the fitting-algorithms to agents evolving in a stochastic environment where: (a) states are the collection of observable market quotes and anterior volatility surfaces [2] (b) actions are the bumps to apply in order to shift the prior volatility

*emmanuel.gnabeyeu-mbiada@polytechnique.edu
†omar.karkar@citi.com
‡imad.idboufous@citi.com

[1] It can also be performed on a less regular basis depending on the liquidity of the asset.
[2] States can also account for spot and/or forward movements.

**DISCLAIMER**

The views, opinions, and conclusions expressed in this report are solely those of the author and do not reflect the official policy or position of Citigroup Global Market Limited, thus should not be attributed to Citigroup. Any content provided in this report is only for informational purposes and is subject to change without notice. It is not intended as a recommendation, advice or solicitation to buy or sell any financial product or service. Citigroup disclaims any responsibility for the accuracy, completeness, or quality of the information contained in this material, and bears no liability for any reliance on, or use of, in any form.

**CONFIDENTIALITY STATMENT**

The methodologies, results and conclusions presented in this thesis are novel, as they arise from original research project. While they could potentially be reproduced based on publicly available knowledges, the content of Chapter 2 should be treated as confidential until the official publication of our research paper.

**ACKNOWLEDGEMENTS:**

For now, let me express my deepest gratitude to all those who have worked too hard to see me presenting this work. And let me express my heartfelt appreciation to the team Equities Derivatives Exotics and Flows for the faith and trust they have placed in me.

I would like to extend my sincere gratitude to my managers Omar Karkar and Truong Nguyen, for their guidance and unwavering support throughout my internship. Their assistance played a crucial role in ensuring the thoroughness of my work.

I would like to thank my internship supervisors at Citi, Omar and Imad for introducing me to the topic. Throughout the project you have actively discussed my progress and results and you have often inspired me to come up with new ideas and approaches. Your constant support has been invaluable to me. Thanks for being patient ( in fact, slow is smooth , and smooth is faster).

I would also like to thank the Markets Quantitative Analysis (MQA) Equities Derivatives of Citigroup Global Market limited for promptly providing me with the necessary insights and for showing genuine interest in the project. A heartfelt thank you to the dedicated teams and members, including CIS, SES, Hybrids, as well as the amazing Directors who generously shared their expertise. Your guidance and support have been instrumental !

I express my appreciation to Citigroup and all its employees who have accompanied me throughout this project, offering support and insightful suggestion to my challenges. Their valuable contributions, coupled with constructive feedback, have significantly enriched my journey and propelled my progress.

Finally, I am grateful to the administration of Ecole Polytechnique and Sorbonne University, to my teachers for fostering a culture of methodical and rigorous work, encouraging a multidisciplinary perspective to address significant scientific and technological challenges at the forefront of knowledge.

I extend my sincere thanks to my advisors at the Master's program in Probability and Finance, for their invaluable feedback, consistent academic guidance and supervision. Heartfelt Thanks to my fellow interns at citi: ilyass, Hippolyte, Junior and Amine. Our Cafe breaks were always a refreshing pause.

Most Importantly, I am deeply grateful for the unwavering support of my family throughout my studies.

**ABSTRACT**

The mathematical approach to many financial decision-making problems has traditionally been through modelling (then relying to model assumptions) with stochastic processes and using techniques from stochastic control with a choice of models mainly dictated by the need to balance tractability with applicability. The rapid changes in the finance industry due to the increasing amount of data have revolutionized the techniques on data processing and data analysis and brought new theoretical and computational challenges. New developments from AI and Machine Learning are able to make full use of the large amount of financial data with fewer model assumptions and to improve decisions in complex financial environments. There have been many applications of these Machine Learning algorithms in a variety of decision-making problems in finance, including optimal execution, portfolio optimization, option pricing and hedging, market making, smart order routing, and robo-advising. This research delves into an in-depth exploration of the application of the particular case of Reinforcement Learning algorithms to some core problems of the Equity Derivatives business amongst others, the volatility fitting and the hedging of derivatives.

The volatility fitting is one of the core problems in the equity derivatives business. Through a set of deterministic rules, the degrees of freedom in the implied volatility surface encoding (parametrization, density, diffusion) are defined. Whilst very effective, this approach widespread in the industry is not natively tailored to learn from shifts in market regimes and discover unsuspected optimal behaviors. In this work, we change the classical paradigm and leverage the latest advances in Deep Reinforcement Learning(DRL) to solve the fitting problem. In particular, we show that variants of Deep Deterministic Policy Gradient (DDPG) and Soft Actor Critic (SAC) can achieve at least as-good as standard fitting algorithms. Furthermore, we explain why the reinforcement learning framework is appropriate to handle complex objective functions and is natively adapted for online learning.

**Keywords:** Volatility Fitting, Continuous Action Spaces, Stochastic and Continuous Control, Sequential Decision Making and Deep Reinforcement Learning.

# Contents

# Chapter 1

# Deep Reinforcement Learning

## 1.1 Introduction

Deep Reinforcement Learning (DRL) algorithm based on using Neurals Networks with the idea to train an agent to learn by interacting with an uncertain environment have showcased oustanding results on complex decisions making and control tasks. The recent availaibility of increasing amounts of data in the complex financial industry has brought news ideas on data processing and statistical modelling techniques. This chapter aims at giving some backgrounds on Reinforcement learning with an extension to Deep Reinforcement learning algorithms, cornerstone of our research work.

## 1.2 Elements of Reinforcement Learning

Reinforcement Learning (RL) refers to a goal-directed learning and decision-making process where an agent acting within an evolving system, learns to make optimal decisions through repeated experiences gained by interacting with the environment without relying on supervision or complete model of the environment.

RL uses a formal framework defining the interaction between a learning agent and their environment in terms of states, actions, and rewards.

The history of Modern RL is grounded in two threads: learning by trial and error and the problem of optimal control for an evolving system and its solution using value functions and dynamic programming that has lead to algorithms such as Q-learning and Actor-Critic. Over the last few years, DRL, emerges on combining classical theoretical results with deep learning and the functional approximation paradigm and has proved to be a fruitful approach to many artificial intelligence tasks from diverse domains.

### 1.2.1 Agent-Environment interaction

RL agent learns through interaction with the environment that it cannot change arbitrarily. It takes action( i.e. any decision the agent wants to learn ) at a certain system state and observes the response from the environment, within which it can learn to optimize its behaviour by trials and errors.

According to this model, the learning process is done in few episodes and steps. At each time step t, the agent observes a state $s_t$ and choose an action $a_t$ to perform. When the action is taken, the environment and the agent transition to a new state $s_{t+1}$, based on the current state and the chosen action. Moreover, every time the environment moves to a new state, it also provides a scalar reward $r_{t+1}$ to the agent as feedback, indicating how good or bad has been the action selected by the agent over the environment. This process is repeated until the agent approaches to an optimal behaviour.

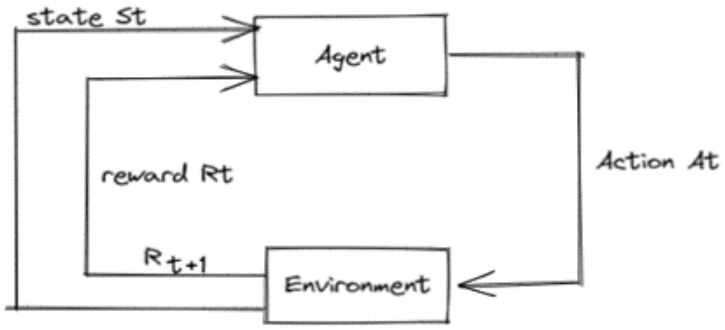



Figure 1.1: The agent–environment interaction in reinforcement learning

### 1.2.2 Markov Decision Process (MDP)

The Markov property says that "The future is independent of the past given the present", which means that starting only from knowledge of the present state, it is possible to make a prediction of all the expected future states and rewards iterating the model.

We consider a stochastic environment $\mathcal{E}$, assumed fully-observed and thus formulate the problem of decision making of an agent acting within an environment over a finite time horizon, as a Markov Decision Process $(\mathcal{S}, \mathcal{A}, r, p)$, where at each time, the agent observes a current state $s_t$ from the continuous state space $\mathcal{S} \in \mathbb{R}^d$, takes an action $a_t$ based on that current state from the continuous action space $\mathcal{A} \in \mathbb{R}^n$ and receives a scalar reward $r_t$ from the stochastic scalar-valued reward function $r = r(a_t, s_t)$.

$p(s', r|s, a)$ is the probability of transitioning to state s', with reward r, from s, a i.e. $p(s', r|s, a) = \mathbb{P}[s_{t+1} = s'|s_t = s, a_t = a]$

The transition dynamics is given by $p(s_{t+1}|s_t, a_t)$ and represents the probability density of the next state. The agent behaviour is defined by a policy $\pi$ (decision-making rule) i.e. actions are drawn from $\pi : \mathcal{S} \to \mathcal{A}$ (resp. $\mathcal{P}(\mathcal{A})$) for a deterministic policy (resp. for a randomized policy which maps the current state $s_t$ to a probability distribution over the action space). we will denote by $\pi(s)$ the action taken in state s under deterministic policy $\pi$ and $\pi(a|s)$ the probability of taking action a in state s under stochastic policy $\pi$. $\pi(a|s) = \mathbb{P}[a_t = a|s_t = s]$

We will use $\rho_\pi(s_t)$ and $\rho_\pi(s_t, a_t)$ to denote the state and state-action marginals of the trajectory distribution induced by a policy $\pi$.

Denoting by $\gamma \in [0, 1]$ a discounting factor, which determines how much importance is to be given to the immediate rewards and future rewards. (Lower values place more emphasis

on immediate rewards, while values close to 1 give more importance to future rewards.), it is standard to define the return from a state as the sum of discounted [1] future reward from following the policy $\pi$ (cumulative reward received from the environment).

$$G_t = R^{\pi}(s_t) = \sum_{i=t}^{T} \gamma^{(i-t)} r(s_i, a_i) = \sum_{i=t}^{T} \gamma^{(i-t)} R_i. \tag{1.1}$$

The agent will learn a policy $\pi$ that describes the way of acting to maximize the expected return in every state.

### 1.2.3 Value functions

Almost all reinforcement learning algorithms involve estimating value functions i.e. functions of states (or of state–action pairs) that estimate how good it is for the agent to be in a given state (or how good it is to perform a given action in a given state). The notion of "how good" here is defined in terms of future rewards that can be expected, or, to be precise, in terms of expected return.

We define the state-action value function or Q-value(value of taking action a in state s under policy $\pi$, denoted $Q^{\pi}(s, a)$) as as the expected return starting from s, taking the action a, and thereafter following policy $\pi$:

$$Q^{\pi}(s, a) = \mathbb{E}_{\pi}[G_t | s_t = s, a_t = a] \tag{1.2}$$

$$= \sum_{i=t}^{T} \mathbb{E}_{(s_i, a_i) \sim \rho_{\pi}}[\gamma^{(i-t)} r(s_i, a_i) | s_t = s, a_t = a]. \tag{1.3}$$

Likewise, the value or state value function[2] (value of state s under policy $\pi$, denoted $V^{\pi}(s)$ (expected return)), is the expected return when starting in s and following $\pi$ thereafter:

$$V^{\pi}(s) = \mathbb{E}_{\pi}[G_t | s_t = s] \tag{1.4}$$

$$= \sum_{i=t}^{T} \mathbb{E}_{(s_i, a_i) \sim \rho_{\pi}}[\gamma^{(i-t)} r(s_i, a_i) | s_t = s] \tag{1.5}$$

$$= max_{a \in \mathcal{A}} Q^{\pi}(s, a). \tag{1.6}$$

The objective of Reinforcement Learning is to learn a policy $\pi$ (i.e. a sequence of actions over time) which maximizes the expected sum of discounted reward from the system (value functions): An agent reinforced with awards based on its interaction with the environment would learn to operate optimally in that environment to maximize the rewards (trials and errors).

[1] it is necessary to use the discount factor to avoid the infinite return discounting the cumulative rewards r, if the task is continuous and does not have any terminal state.

[2] 1.6 and 1.20 provide a prediction of how good each state or each state/action pair is.

The problem is thus formulated mathematically by defining the optimal value function(Resp. optimal Q-function) for each state s as follow:

$$V^*(s) = \sup_{\pi} V^{\pi}(s) = \sup_{\pi} \mathbb{E}_{\pi}\left[\sum_{i=t}^{T} \gamma^{(i-t)} r(s_i, a_i) \middle| s_t = s\right]. \tag{1.7}$$

$V^*(s)$ is the value of state s under the optimal policy i.e. $V^*(s) = V^{\pi^*}(s)$.

### 1.2.4 Bellman Equations and Bellman Optimality Equations

A fundamental property of value functions used throughout reinforcement learning and dynamic programming is that they satisfy particular recursive relationships. For any policy $\pi$ and any state s, the following consistency condition holds between the value of s and the value of its possible successor states:

$$\begin{aligned}
V^{\pi}(s) &= \mathbb{E}_{\pi}[G_t | s_t = s] && (1.8)\\
&= \mathbb{E}_{\pi}\left[\sum_{i=t}^{T} \gamma^{(i-t)} r(s_i, a_i) \middle| s_t = s\right] && (1.9)\\
&= \mathbb{E}_{\pi}\left[\sum_{i=t}^{T} \gamma^{(i-t)} R_i \middle| s_t = s\right] && (1.10)\\
&= \mathbb{E}_{\pi}\left[R_t + \gamma \sum_{i=t+1}^{T} \gamma^{(i-t)} R_i \middle| s_t = s\right] && (1.11)\\
&= \sum_{a} \pi(a|s) \sum_{s'} \sum_{r} p(s', r|s, a)\left[r + \gamma \mathbb{E}_{\pi}\left[\sum_{i=t+1}^{T} \gamma^{(i-t)} R_i \middle| s_{t+1} = s'\right]\right]. && (1.12)
\end{aligned}$$

$$V^{\pi}(s) = \sum_{a} \pi(a|s) \sum_{s', r} p(s', r|s, a)\left[r + \gamma V^{\pi}(s')\right]. \tag{1.13}$$

This is the bellman equation for V.

The standard approach to solve this optimization in 1.15 makes use of the well-known Dynamic Programming Principle (DPP) to derive the following variation of the recursive linear Bellman equation called the **Bellman optimality equation:**

$$V^*(s) = \sup_{a \in \mathcal{A}}\left[\mathbb{E}[r(s, a)] + \gamma \mathbb{E}_{s' \sim p(s'|s, \pi(s))}\left[V^*(s')\right]\right] \tag{1.14}$$

In fact, the well-known Dynamic Programming Principle (DPP), that the optimal policy can be obtained by maximizing the reward from one step and then proceeding optimally from the

new state (1.13), can be used to derive the above Bellman equation.

$$V^*(s) = \sup_{a\in\mathcal{A}} Q^{\pi^*}(s,a) \tag{1.15}$$

$$= \sup_{a\in\mathcal{A}} \mathbb{E}_{\pi^*}\left[\sum_{i=t}^{T} \gamma^{(i-t)} r(s_i,a_i)\middle| s_t = s, a_t = a\right]. \tag{1.16}$$

$$= \sup_{a\in\mathcal{A}} \mathbb{E}_{\pi^*}\left[\sum_{i=t}^{T} \gamma^{(i-t)} R_i\middle| s_t = s, a_t = a\right]. \tag{1.17}$$

$$= \sup_{a\in\mathcal{A}} \mathbb{E}_{\pi^*}\left[R_t + \gamma \sum_{i=t+1}^{T} \gamma^{(i-t)} R_i\middle| s_t = s, a_t = a\right]. \tag{1.18}$$

$$= \sup_{a\in\mathcal{A}} \sum_{s',r} p(s',r|s,a)\Big[r + \gamma V^{\pi}(s')\Big] \tag{1.19}$$

The Bellman optimality equation for Q is:

$$Q^*(s,a) = Q^{\pi^*}(s,a) = \mathbb{E}_{\pi}[R_{t+1} + \gamma \sup_{a'} Q^*(s_{t+1},a')|s_t = s, a_t = a] \tag{1.20}$$

$$= \sum_{s',r} p(s',r|s,a)\Big[r + \gamma \sup_{a'} Q^*(s',a')\Big]. \tag{1.21}$$

It follows from the literature that, in order to solve this problem, we can perform either value iteration [15], policy iteration [9, 18] or a combinasion of the two in actor-critic methods [12]. In fact, there are two main approaches to solving RL problems: methods based on value functions and methods based on policy search. Moreover, there is also a hybrid actor-critic approach that employs both value functions and policy search.

The goal of reinforcement learning is to find an optimal behaviour strategy for the agent to obtain optimal rewards. The policy gradient methods target at modelling and optimizing the policy directly through the **Policy Gradient Theorem.**

### 1.2.5 Policy Gradient Theorem

The policy is usually modelled with a parameterized function respect to $\theta$, $\pi_\theta(a|s)$. The objective function $J = V(s_0)$ become a function of $\theta$(i.e. J($\theta$)). Using gradient ascent, we can move $\theta$ toward the direction suggested by the gradient $\nabla_\theta J(\theta)$ to find the best $\theta$ for $\pi_\theta$ that produces the highest return.

Policy Gradient Theorem provides a nice reformation of the derivative of the objective function and simplify the gradient computation $\nabla_\theta J(\theta)$

**Theorem 1** (Policy Gradient)**.**

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi[Q^\pi(s,a)\nabla_\theta \ln \pi_\theta(a|s)]$$

Where $\mathbb{E}_\pi$ refers to $\mathbb{E}_{s\sim\rho_\pi(s),a\sim\pi_\theta} = \mathbb{E}_{(s,a)\sim\rho_\pi(s,a)}$

Proof. see 3.2

## 1.3 Deep Reinforcement Learning

Combing the above-mentioned ideas with Deep Learning has lead to significant performance improvement and the spreading of the so called Deep Reinforcement Learning (DRL).

In fact, when we are in the setting of continuous, high dimensional action spaces, it is known from the literature [1] that classical tabular value-based and policy-based methods are challenging, we rather consider a parametrized value functions $Q(s, a; \theta^Q)$ and $V(s, \theta^V)$ or policies $(\pi(s, a; \theta^\pi))$ and then use Neural Networks as function approximators.[3]. There are several popular neural network architectures amongst which the fully-connected neural networks which we will be focussing on in the next paragraph.

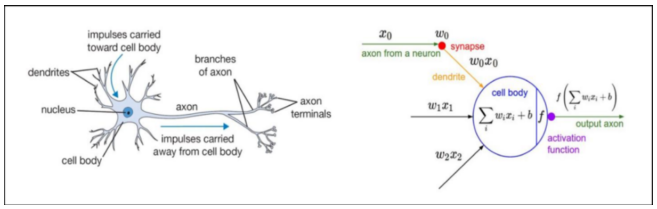



Figure 1.2: Human and mathematical neurons.

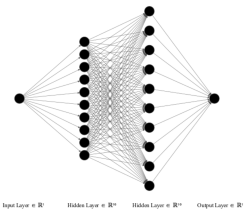


Figure 1.3: Neural Network architecture.

### 1.3.1 Fully Connected Artificial Neural Networks (FCNN)

**Fully Connected Artificial Neural Networks (FCNN):** It is a neural network architecture where any given neuron(Cf Figure 1.2) is connected to all neurons in the previous layer.( cf Figure 1.3).

Let us consider data samples $x_i \in \mathbb{R}^{d_{in}}$ and $y_i \in \mathbb{R}^{d_{out}}$ for $1 \leq i \leq M$ associated to a regression problem, where $(x_i)$ are the inputs and $(y_i)$ are the outputs. That is, we look for a function $F : \mathbb{R}^{d_{in}} \to \mathbb{R}^{d_{out}}$ which fits to the data i.e. $\forall 1 \leq i \leq M, F(x_i) \sim y_i$

Let $K + 1$, $K \in \mathbb{N}$ be the number of layers of the neural network and for $k = 0, ..., K$; let $d_k \in \mathbb{N}$ be the size of the $k^{th}$ layer with $d_0 = d_{in}$ and $d_K = d_{out}$.

Let $\phi : \mathbb{R} \to \mathbb{R}$ be a non-linear function, generally chosen to be a sigmoid-type or a ReLU-type function (called activation function in the neural networks literature),

For $k = 1, ..., K, x \in \mathbb{R}^{d_{k-1}}$ and for $\theta_k \in \mathcal{M}_{d_k, d_{k-1}}((\mathbb{R}))$ (weight matrices) and $b_k \in \mathbb{R}^{d_k}$(bias vectors).

We define the below vector in $\mathbb{R}^{d_k}$ where the scalar function $\phi$ is applied to the vector $\theta_k x + b_k$ coordinate by coordinate:

$$\phi_{\theta_k, b_k}(x) := [\phi([\theta_k x + b_k]_i)]_{1 \leq i \leq d_k} \tag{1.22}$$

Writing $\Phi = (\theta_1, b_1, ..., \theta_K, b_K) = (\theta_k, b_k)_{k=1,...,K}$, the output of the neural network with $\sum_{l=1}^{K} d_{l-1} \times d_l$ parameters, is given by:

[3]Thanks to the universal approximation theorem, Neural networks are known for their ability to approximate a wide range of non-linear functions in high dimension and to fit to many non-linear regression task [34, 31, 32]

$$F_\Phi(x) = \theta_K(\phi_{\theta_{K-1},b_{K-1}} \circ ... \circ \phi_{\theta_1,b_1}(u)) + b_K \tag{1.23}$$

### 1.3.2 Training of Neural Networks

**Training of Neural Networks:** The objective is to extract a model from the empirical data. We look for a function $F$ in a family of functions parametrized by a finite-dimension parameter: $\left\{F_\Phi, \Phi \in \mathbb{R}^{d_1 \times d_{in}} \times \mathbb{R}^{d_1} \times ... \times \mathbb{R}^{d_K \times d_{out}} \times \mathbb{R}^{d_K}\right\}$

For a loss function $L : \mathbb{R}^{d_{out}} \to \mathbb{R}^+$, which measures the error between the prediction $F_\Phi(x_i)$ and the true data $y_i$, the regression problem is the minimization of the average loss over a small batch of data $\mathcal{I}_{n+1}$. The parameters $\Phi$ are updated in the descent direction of L using either Stochastic algorithms as introduced by Robbins and Monro [2] or a more stable version and still computationally efficient, the batch gradient descent and further their extension:

$$\Phi_{n+1} = \Phi_n - \frac{\gamma_{n+1}}{N_{batch}} \sum_{i \in \mathcal{I}_{n+1}} \partial_\Phi F_\Phi(x_i) \dot{\nabla} L(F_\Phi(x_i) - y_i) \tag{1.24}$$

Using Neural networks for reinforcement learning involve some (above-mentioned-like) optimization algorithm which assumes that samples are independently and identically distributed (IID). This assumption does not hold any more when the samples are generated from exploring sequentially in an environment, this is challenging in the context of RL. we used a Replay Buffer to address that issue and improve the learning performance.

### 1.3.3 Replay Buffer.

**Replay Buffer.** The agent learns over time to select his actions based on his past experiences (exploitation) and/or by trying new choices (exploration). The past experiences are stored in a *"Replay Buffer"* mainly used to improve the performance of neural networks by providing nearly non correlated samples at each parameters update. Essentially, it is a finite size cache memory, which temporarily saves the agent observations during the learning process.

Before the training process, it is full of tuples or transitions $(s_t, a_t, r_t, s_{t+1})$ sampled from the environment according to the initial exploration policy. During the training process and at each time step, we update the replay buffer by discarding the transitions with the worst reward or the oldest reward and storing a new tuple $(s_t, a_t, r_t, s_{t+1})$ sampled from the environment according to the current policy. In this work, the weights of our Neural Networks are initialized from uniform distributions (Xavier initialisation).

The first motivation of the replay memory in DQN [25] was to alleviate the problems of correlated data and non-stationary distributions by smoothing the training distribution over many past experiences. In fact, learning processes are usually in mini-batches i.e. the parameters of the neural networks are updated by uniformly choosing a minibatch of samples from the buffer and performing batch gradient descent (a more stable version and still computationally efficient of Stochastic gradient descent). Such algorithm assume each batch sample to be (IID) from a fixed distribution. However, exploration from the environment clearly generates correlated samples, subject to distributional shift. Thus we consider a *"Replay Memory"* with large enough size, to

store past experiences and uniformly sample a mini batch at each update of Neural Networks parameters in order to minimize the correlations between samples allowing the algorithms to learn across a set of uncorrelated transitions.

It is usefull to Off-policy RL methods which allow learning based on data captured by arbitrary policies [4].

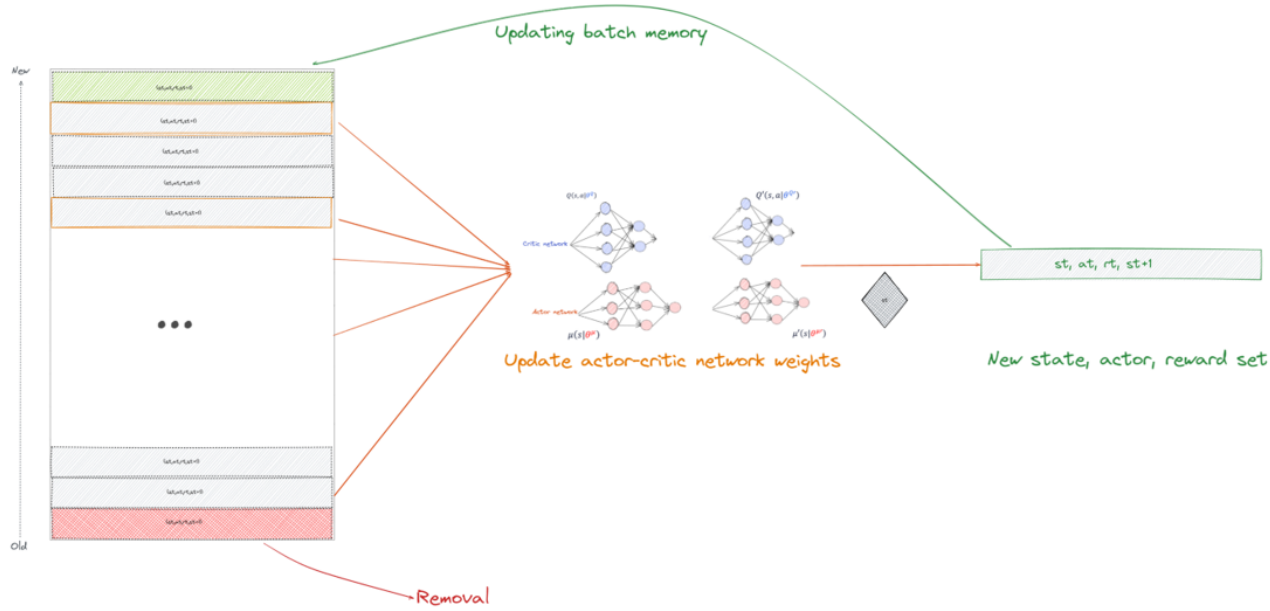



Figure 1.4: A snapshot of the Replay Memory: Finite sized cache used in actor/critic update.

## 1.4 Actor-Critic algorithms: DDPG and SAC

The combinaison of value-based and policy-based methods leads to Actor-Critic algorithms. It can be extend to neural Actor-Critic algorithms by using neural networks for functional approximations. This has lead to significant performance improvement and the spreading of the so called Deep Reinforcement Learning (DRL).

In fact, when we are in the setting of continuous, high dimensional action spaces, it is known from the literature [1] that classical tabular value-based and policy-based methods are challenging, we rather consider parametrized value functions $Q(s, a; \theta^Q)$ and $V(s, \theta^V)$ or policies $(\pi(s, a; \theta^\pi))$ and then use Neural Networks as function approximators [5]. There are several popular neural network architectures amongst which the fully-connected neural networks used in this study and for which the reader is invited to refer to 1.3.1 for more details.

The combination of value-based and policy-based methods leads to Actor-Critic algorithms. This class of RL algorithm alternates between policy evaluation by computing the value function for a policy and policy improvement by using the value function to obtain a better policy.

[4] In contrast to on-policy algorithms which require updating function approximators according to the currently followed policy. I

[5] Thanks to the universal approximation theorem, Neural networks are known for their ability to approximate a wide range of non-linear functions in high dimension and to fit to many non-linear regression task [34, 31, 32]

It can be extended to neural Actor-Critic algorithms by using neural networks for functional approximations. Deep Deterministic Policy Gradient (DDPG) and Soft Actor-Critic algorithms are part of that class of RL algorithms (see Figure 1.5) which have demonstrated state-of-art performance on a range of challenging decision-making and control tasks. Those algorithms use a replay buffer to improve the performance of neural networks. see the Figure 1.4 for more details.

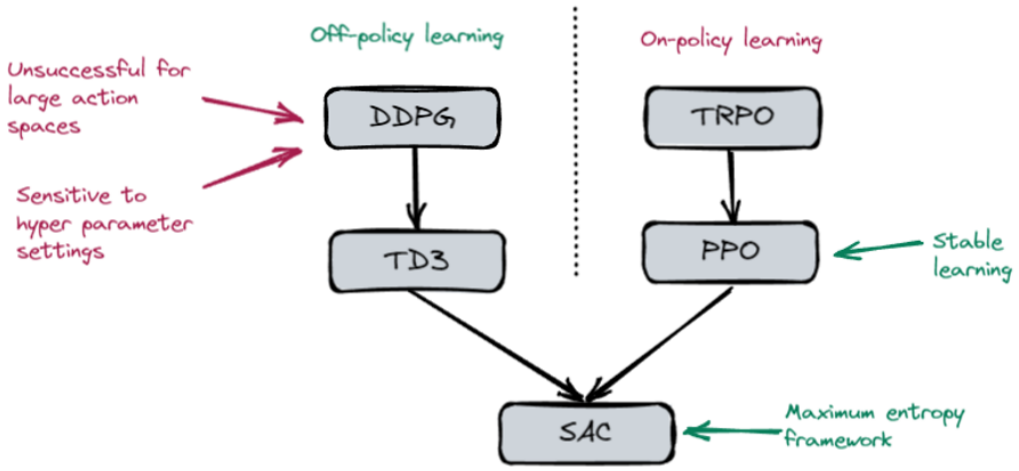



Figure 1.5: RL algorithms.

### 1.4.1 Deep Deterministic Policy Gradient(DDPG).

**DDPG:** It is a model-free off-policy[6] Actor-Critic algorithm, first introduced in [10], which combines the Deep Q-Network (DQN) [25] and Deep Policy Gradient (DPG) algorithms. It extends the DQN to continuous action spaces by incorporating DPG to learn a deterministic strategy $\pi^D$. The critic estimates the $Q$-value function using off-policy data and the recursive Bellman equation:

$$Q(s_t, a_t) = r(s_t, a_t) + \gamma Q\left(s_{t+1}, \pi^D(s_{t+1}, \theta^\pi)\right),$$

The actor is trained to maximize the critic's estimated $Q$-values by back-propagating through both networks.

To encourage exploration, it uses the following action (stochastic policy):

$$a_t \sim \pi^D(s_t, \theta^\pi) + \mathcal{N} \tag{1.25}$$

where $\mathcal{N}$ is either an Ornstein-Uhlenbeck (Ornstein and Uhlenbeck, 1930) noise process $\mathcal{N} = \text{OU}(0, \sigma^2)$ (correlated) or an additive Gaussian noise $\mathcal{N} = \mathcal{N}(0, \sigma^2 I)$ (uncorrelated), chosen according to the environment. .[7] It thus treat the problem of exploration independently from the learning algorithm.

[6] An off-policy agent learns the value of the optimal policy independently of the agent's actions while an on-policy agent learns the value of the policy being carried out by the agent including the exploration steps.

[7] We will used an additive Gaussian noise with a decreasing standard deviation to dampen exploration as the agent is learning the optimal policy.

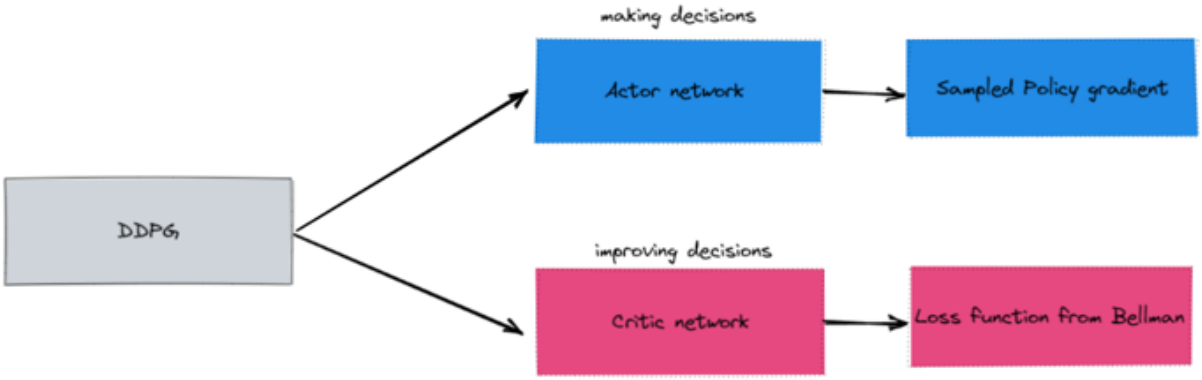



Figure 1.6: A snapshot of the Actor-critic approach based on DPG algorithm

However, this algorithm is known to suffer from high sample complexity and sensibility to hyper-parameters. [8] Subsequently, we also, consider an extension known as SAC.

### 1.4.2 Soft Actor Critic (SAC)

**Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor:** It is an off-policy actor-critic deep RL algorithm based on the maximum entropy reinforcement learning framework. It was introduced in [11] and extend the DDPG setting by allowing the actor to maximize the expected reward while also maximizing the entropy. The off-policy formulation enables reuse of previously collected data for efficiency, and the entropy maximization enable stability and provides a substantial improvement in exploration and robustness. It favors stochastic policies by considering a more general and augmented entropy objective: [9]

$$J_\pi = \sum_{i=t}^{T} \mathbb{E}_{(s_i,a_i)\sim\rho_\pi}[\gamma^{(i-t)}(r(s_i,a_i) + \alpha\mathcal{H}(\pi(\cdot|s_t))]. \tag{1.26}$$

$$\pi^* = \arg\max_{\pi\in\Pi} J_\pi.$$

where $\mathcal{H}$ is the entropy functional, $\Pi$ a family of tractable policy [10] and $\alpha$ is the temperature parameter which determines the relative importance of the entropy term against the reward, and thus controls the stochasticity of the optimal policy.

We want to project the improved policy into the desired set of policies in order to account for the constraint that $\pi \in \Pi$. We will rather use the information projection defined in terms of the Kullback-Leibler divergence and redefined the actor loss function. [11]

[8] We witnessed it by changing differents hyperparameters.

[9] Sample-efficient, Robustness to noise, random seed and hyperparameters, Scale to high-dimensional observation/action space

[10] we prefer policies that are tractable, we will additionally restrict the policy to some set of policies $\Pi$, which can correspond, for example, to a parameterized family of distributions such as Gaussians.

[11] In principle we could choose any projection

Let denote by $Q_\theta$ the critic network, parameterized by $\theta$ and $\pi_\phi$ the actor network, parameterized by $\phi$ which outputs a probability distribution over the action space i.e. the best action to take from state $s_t$ is then sampled from the probability distribution $\pi_\phi(\cdot|s_t)$.

We redefined the policy loss as the KL divergence or the gap between the probability distribution over the action space proposed by the actor network and the probability distribution induced by the exponentiated current Q function normalized by the factor $Z_\theta$: In fact, the high values of the latter indicates the areas of the action space where the cumulative expected sum of rewards is approximated to be high, minimizing the KL divergence means getting an efficient actor network associated to actions yielding highly rewarded trajectories.

With the definition of the KL divergence, we have:

$$\begin{aligned}
J_\pi(\phi) &= \mathbb{E}_{s_t\sim\mathcal{D}}\left[D_{KL}\left(\pi_\phi(\cdot|s_t)\;\middle\|\;\frac{\exp\left(\frac{1}{\alpha}Q_\theta(s_t,\cdot)\right)}{Z_\theta(s_t)}\right)\right] \\
&= \mathbb{E}_{s_t\sim\mathcal{D}}\left[\int_{a_t}\pi_\phi(a_t|s_t)\log\frac{\pi_\phi(a_t|s_t)}{\exp\left(\frac{1}{\alpha}Q_\theta(s_t,a_t)\right)/Z_\theta(s_t)}da_t\right] \\
&= \mathbb{E}_{s_t\sim\mathcal{D}}\left[\int_{a_t}(\pi_\phi(a_t|s_t)\alpha\log\pi_\phi(a_t|s_t)-\pi_\phi(a_t|s_t)Q_\theta(s_t,a_t))da_t\right]+C^{\text{ste}} \\
&= \mathbb{E}_{s_t\sim\mathcal{D}}\left[\mathbb{E}_{a_t\sim\pi_{\phi(\cdot|s_t)}}[\alpha\log\pi_\phi(a_t|s_t)-Q_\theta(s_t,a_t)]\right]+C^{\text{ste}} \qquad (\star)
\end{aligned}$$

The soft Q-function parameters can be trained to minimize the soft Bellman residual:

$$J_Q(\theta)=\mathbb{E}_{(s_t,a_t)\sim\mathcal{D}}\left[\frac{1}{2}\left(Q_\theta(s_t,a_t)-\left(r(s_t,a_t)+\gamma\,\mathbb{E}_{s_{t+1}\sim p}\left[V_{\bar\theta}(s_{t+1})\right]\right)\right)^2\right], \qquad (1.27)$$

where the value function is implicitly parameterized through the soft Q-function parameters and it can be optimized with stochastic gradients.

$$\hat\nabla_\theta J_Q(\theta)=\nabla_\theta Q_\theta(a_t,s_t)\left(Q_\theta(s_t,a_t)-\left(r(s_t,a_t)+\gamma\left(Q_{\bar\theta}(s_{t+1},a_{t+1})-\alpha\log\left(\pi_\phi(a_{t+1}|s_{t+1})\right)\right)\right)\right). \qquad (1.28)$$

The update makes use of a target soft Q-function with parameters $\bar\theta$ that are obtained as an exponentially moving average of the soft Q-function weights, which has been shown to stabilize training. Finally, the policy parameters can be learned by directly minimizing the expected KL-divergence in $(\star)$:

$$J_\pi(\phi)=\mathbb{E}_{s_t\sim\mathcal{D}}\left[\mathbb{E}_{a_t\sim\pi_\phi}\left[\alpha\log\left(\pi_\phi(a_t|s_t)\right)-Q_\theta(s_t,s_t)\right]\right] \qquad (1.29)$$

There are several options for minimizing $J_\pi$. A typical solution for policy gradient methods is to use the likelihood ratio gradient estimator , which does not require backpropagating the gradient through the policy and the target density networks. However, in this case, the target density is the Q-function, which is represented by a neural network an can be differentiated, and it is thus

convenient to apply the reparameterization trick instead for the probability law according to which actions are drawn: [12] i.e.

$$a_t = f_\phi(\epsilon_t; s_t) = \mu_\phi(s_t) + \epsilon_t \sigma_\phi(s_t), \tag{1.30}$$

where $\epsilon_t$ is an input noise vector such as a spherical Gaussian ($\epsilon_t \sim \mathcal{N}(0, I)$ ).

We can now rewrite the objective in Equation 1.29 as

$$J_\pi(\phi) = \mathbb{E}_{s_t \sim \mathcal{D}, \epsilon_t \sim \mathcal{N}} \left[ \alpha \log \pi_\phi(f_\phi(\epsilon_t; s_t)|s_t) - Q_\theta(s_t, f_\phi(\epsilon_t; s_t)) \right], \tag{1.31}$$

where $\pi_\phi$ is defined implicitly in terms of $f_\phi$. We can approximate the gradient of [13] with

$$\hat{\nabla}_\phi J_\pi(\phi) = \nabla_\phi \alpha \log \left( \pi_\phi(a_t|s_t) \right) + \left( \nabla_{a_t} \alpha \log \left( \pi_\phi(a_t|s_t) \right) - \nabla_{a_t} Q(s_t, a_t) \right) \nabla_\phi f_\phi(\epsilon_t; s_t), \text{[14]} \tag{1.32}$$

where $a_t$ is evaluated at $f_\phi(\epsilon_t; s_t)$. This unbiased gradient estimator extends the DDPG style policy gradients to any tractable stochastic policy.

SAC is brittle with respect to the temperature parameter. Unfortunately it is difficult to adjust temperature, because the entropy can vary unpredictably both across tasks and during training as the policy becomes better. An improvement on SAC formulates a constrained optimization problem: **while maximizing the expected return, the policy should satisfy a minimum entropy constraint:**

**Automating Entropy Adjustment for Maximum Entropy RL:** Choosing the optimal temperature is non-trivial since it need to be tuned for each task. Simply forcing the entropy to a fixed value is a poor solution, since the policy should be free to explore more in regions where the optimal action is uncertain, but remain more deterministic in states with a clear distinction between good and bad actions.

We overcome this issue as proposed in [13] by formulating a different maximum entropy RL objective, where the average entropy of the policy is treated as a constraint, while the entropy at different states can vary.

We show that the dual to this constrained optimization leads to the soft actor-critic updates, along with an additional update for the dual variable, which plays the role of the temperature.

Our aim is to find a stochastic policy with maximal expected return that satisfies a minimum expected entropy constraint. Formally, we need to solve the entropy-constrained maximum expected return objective problem.

$$\max_{\pi_{t:T}} \mathbb{E}_{\rho_\pi} \left[ \sum_{i=t}^{T} \gamma^{(i-t)} r(s_i, a_i) \right] \tag{1.33}$$

$$\text{s.t. } \mathbb{E}_{(s_t, a_t) \sim \rho_\pi} \left[ -\log(\pi_t(a_t|s_t)) \right] \geq \mathcal{H}_0 \quad \forall t \tag{1.34}$$

[12] Instead of the standard likelihood ratio method. This may result in a lower variance estimator and still is an open question on which no consensus is found in the literature.

[13] by writing the second order Taylor development of the loss function

[14] Note that this analytical gradient is useless in practice with common deep learning frameworks such as pytorch [Paszke et al., 2017] or tensorflow [Abadi et al., 2016]. Indeed, these libraries are such that writing the policy loss is enough: the gradient is automatically computed when the backward pass is performed.

where $\mathcal{H}_0$ is a desired minimum expected entropy [15].

Since the policy at time $t$ can only affect the future objective value, we can employ an (approximate) dynamic programming approach, solving for the policy backward through time. We rewrite the objective as an iterated maximization:

$$\max_{\pi_0}\left(\mathbb{E}\left[r(s_0,a_0)\right]+\max_{\pi_1}\left(\mathbb{E}\left[\ldots\right]+\max_{\pi_T}\mathbb{E}\left[r(s_T,a_T)\right]\right)\right), \tag{1.35}$$

subject to the constraint on entropy. Starting from the last time step, we change the constrained maximization to the dual problem. Subject to $\mathbb{E}_{(s_T,a_T)\sim\rho_\pi}\left[-\log(\pi_T(s_T|s_T))\right]\geq\mathcal{H}_0$,

$$\max_{\pi_T}\mathbb{E}_{(s_t,a_t)\sim\rho_\pi}\left[r(s_T,a_T)\right]=\min_{\alpha_T\geq 0}\max_{\pi_T}\mathbb{E}\left[r(s_T,a_T)-\alpha_T\log\pi(a_T|s_T)\right]-\alpha_T\mathcal{H}_0, \tag{1.36}$$

where $\alpha_T$ is the dual variable. We have also used strong duality, which holds since the objective is linear and the constraint (entropy) is convex function in $\pi_T$. This dual objective is closely related to the maximum entropy objective with respect to the policy, and the optimal policy is the maximum entropy policy corresponding to temperature $\alpha_T$: $\pi_T^*(a_T|s_T;\alpha_T)$. We can solve for the optimal dual variable $\alpha_T^*$ as:

$$\arg\min_{\alpha_T}\mathbb{E}_{s_t,a_t\sim\pi_t^*}\left[-\alpha_T\log\pi_T^*(a_T|s_T;\alpha_T)-\alpha_T\mathcal{H}_0\right]. \tag{1.37}$$

To simplify notation, we make use of the recursive definition of the soft Q-function

$$Q_t^*(s_t,a_t;\pi_{t+1:T}^*,\alpha_{t+1:T}^*)=\mathbb{E}\left[r(s_t,a_t)\right]+\mathbb{E}_{\rho_\pi}\left[Q_{t+1}^*(s_{t+1},a_{t+1})-\alpha_{t+1}^*\log\pi_{t+1}^*(a_{t+1}|s_{t+1})\right], \tag{1.38}$$

with $Q_T^*(s_T,a_T)=\mathbb{E}\left[r(s_T,a_T)\right]$. Now, subject to the entropy constraints and again using the dual problem, we have

$$\begin{aligned}&\max_{\pi_{T-1}}\left(\mathbb{E}\left[r(s_{T-1},a_{T-1})\right]+\max_{\pi_T}\mathbb{E}\left[r(s_T,a_T)\right]\right) \\ &\quad=\max_{\pi_{T-1}}\left(Q_{T-1}^*(s_{T-1},a_{T-1})-\alpha_T\mathcal{H}_0\right) \\ &\quad=\min_{\alpha_{T-1}\geq 0}\max_{\pi_{T-1}}\left(\mathbb{E}\left[Q_{T-1}^*(s_{T-1},a_{T-1})\right]-\mathbb{E}\left[\alpha_{T-1}\log\pi(a_{T-1}|s_{T-1})\right]-\alpha_{T-1}\mathcal{H}_0\right)+\alpha_T^*\mathcal{H}_0.\end{aligned} \tag{1.39}$$

In this way, we can proceed backwards in time and recursively optimize Equation 1.34. Note that the optimal policy at time $t$ is a function of the dual variable $\alpha_t$. Similarly, we can solve the optimal dual variable $\alpha_t^*$ (the optimal temperature at step t) after solving for $Q_t^*$ and $\pi_t^*$:

$$\alpha_t^*=\arg\min_{\alpha_t}\mathbb{E}_{a_t\sim\pi_t^*}\left[-\alpha_t\log\pi_t^*(a_t|s_t;\alpha_t)-\alpha_t\mathcal{H}_0\right]. \tag{1.40}$$

[15] we choose the entropy target $\mathcal{H}_0$ to be -dim($\mathcal{A}$) as proposed by the article.

## 1.5 Overview and Outlook

In the finance industry, new ideas coming from reinforcement learning have been developed and adapted to financial problems with many successes. For surveys in the literature on RL applications in finance, the reader is invited to refer to [1, 7, 23, 17, 19, 20, 21]. This has attracted a lot of attention in applying RL techniques to improve decision-making in various financial markets and thus has instigated the following natural extension to the volatility fitting problem.

# Chapter 2

# Deep Reinforcement Learning Framework For The Volatility Modeling And Fitting Problem

This Research work delves into the application of DRL for volatility fitting problem: We investigate the performance of model-free RL aka Actor-Critic Methods on continuous action space for a novel approach to fit the volatility surface. To the best of our knowledge, this is the first study of DRL on volatily fitting model. Our Research focus on providing a method for fitting the volatility surface after a market move for an arbitrary initial volatility surface and market state. We demonstrate the effectiveness of this framework for differents markets scenari accross different random seeds.

## 2.1 Introduction

The implied volatility, which is one of the most important risk factors in the equity business, is usually encoded in a functional form with a limited set of quasi-orthogonal parameters. This functional form, often referred to as volatility parametrization, allows users to describe the movements and deformations of the implied volatility surface in an easy way and helps in generating predictive signals for pricing and risk management of equity derivatives.

The process through which the coefficients of the volatility parametrization are determined is called calibration and is often performed automatically [1] by an algorithm tailored to optimize pre-determined objective functions. For instance, most algorithms aim at matching the mid implied volatility at all expiries where there is a visible market under non-arbitrage constraints. The fitting algorithms used in the industry often account for large sets of conditions such as liquidity, presence of outliers or even macro-events however all those algorithms have native rigidity in them as they follow preset rules for every market conditions. By design, the "deterministic algorithms" cannot find optimal solutions outside of their definition scope and cannot re-use past accumulated experiences as every output is flashed out.

[1] It can also be performed on a less regular basis depending on the liquidity of the asset.

In this work, we look at the fitting problem from the angle of reinforcement learning (RL). In this new paradigm, we assimilate the fitting-algorithms to agents evolving in a stochastic environment where: (a) states are the collection of observable market quotes and anterior volatility surfaces (b) actions are the bumps to apply in order to shift the prior volatility parametrization and (c) rewards are the opposite of well designed errors. By adopting a RL framework, one can benefit from the native exploration/exploitation trade-off where optimal actions are taken with some margin for exploration that could unveil new optimal decisions. Furthermore, in continuous states (which is the case for the volatility fitting problem), one can heavily benefit from replay buffers which are massive logs with historical or synthetic experiences. Finally, by learning in a stochastic environment where market quotes are continuously evolving RL agents can learn to track market dynamics and to better stabilize the volatility parametrization coefficients.

## 2.2 Modeling Specification

For a market participant in the Equities derivatives market, one of the fundamental objects to manipulate is the implied volatility surface of an asset (defined by their prices) indexed by the expiration of the options on the underlying asset and/or its strike price.

The construction of implied volatility surfaces which are both sufficiently flexible and compatible with arbitrage constraints (call spread, butterfly and calendar spread) is a subject of major importance for any participant in the financial markets.

The volatility calibration consists in adjusting the coefficients of the volatility parametrisation in order to fit the market observable implied volatility. More generally, a parametric functional form is used to encode the implied volatility for a given maturity at every strike. The volatility fitting problems aims to fit the market implied volatility and provide interpolation and extrapolation whenever the market data is not available. It is used to price structured products with volatility exposure, to calibrate base model (local volatility, stochastic local volatility etc.) and for risk management.

This section starts with the traditional approach to volatility calibration which consists in adjusting the coefficients of the volatility parametrisation in order to fit the market observable implied volatility and next extend the problem to the novel RL framework where the objective is rather to bump the parameters in order to maximize a defined reward function.

In this section, we provide a brief overview of how the volatility fitting is traditionally performed. We then frame the problem into an RL setting, translating the classical definitions into the reinforcement learning framework. We explain why the RL architecture, compared to the deep neural networks, is well suited for efficient volatility fitting. Finally, we conclude this section by detailing the implementation of the Actor-critic algorithms (DDPG and SAC) and explaining how they were adapted to the fitting game.

### 2.2.1 Volatility Fitting: State of the art

Many methods exist in the industry to encode the implied volatility information extracted from the market. For example, some practitioners can use closed or semi-closed parametric forms to define the variance (or the volatility) at every strike (see for example [27]). Other approaches consist in modeling the implied density of the asset directly (see [26]) or even use a diffusion

style model to fit the market at every expiry.
For a survey of methodologies for constructing implied volatility surface (IVS) in practice, the reader can refer to [30].
In this paper, we restrict ourselves to the modeling of the variance using a parametric function. The techniques developed to solve the calibration of the volatility surface can be perfectly scaled to all the other parametrization choices. Parametrization of the variance can take small to very large sets of coefficients [2]. To keep the action space small, we restrict ourselves to 3 parameters and consider extending to a larger set in a future study. Using 3 parameters is often not enough to fit the market properly, as there are not enough degrees of freedom to sufficiently bend the model implied volatility, this is why we will be adding to the RL results a benchmark generated from a classical optimizer.

Traditionally, once the vehicle encoding the implied volatility is selected, the choice of the target is defined. In most cases, practitioners aim at positioning the model implied volatility at the mid [3] factoring into account several effects such as the presence of arbitrage, smoothness of the term-structure, stability between consecutive fits and so on. A careful selection of the reward is thus important when translating the classical model-based approach into a model-free reinforcement learning architecture. The fitting logic itself is driven by a set of deterministic rules where market data is massaged to generate aggregated quotes, the parametrization coefficients adjusted to approach the mid volatilities for some strikes and the output coefficients set cleaned from any bias induced by the numerical method.

A first and natural alternative would be to consider the volatility fitting problem as a predictive exercise where the input $x$ is the collection of all available information at time $t$ and the output $\Delta y$ is the variation of the parametrization's parameters between $t$ and $t + \epsilon$. In such setting, the goal is to approximate the application $\phi$ such that $\Delta y = \phi(x)$. A very famous family of approximators are the feed forward neural networks which can learn large set of functions. While it is a viable and intuitive approach, solving the problem with such design has several downsides: on the one hand, the training set used to calibrate the weights of the neural networks captures solely a finite number of market regimes. This would require either a frequent re-calibration or training from the user to track changes in market regimes or an immense dataset to expand the coverage of the training set. On the other hand, labeling all the inputs $x$ assumes implicitly a choice of the objective function (for example low error to the mid). This means the calibrated neural network cannot be re-used when the user has a different objective function in mind [4] .

The second alternative, which we introduce in this paper, is to swap the *"deterministic algorithms"* with the Deep Reinforcement Learning framework. On the one hand, the actor neural network $\phi_{reward}$ provides a direct link between the inputs $x$ and the output $\Delta y$. On the other hand, the replay buffer is a dynamic training set which can swiftly track the market evolution. Furthermore, when the objective function considered for the fitting is complex (e.g:

[2]Some market providers use parametrizations with up to 20 coefficients.
[3]The mid is computed from the aggregated bid/ask volatilities.
[4]The user could build different neural nets to overcome this limitation but it also means having distinct training sets which could bare time and monetary cost challenges.

desk PnL) and the standard optimizer too heavy to run [5], the RL approach can learn how to interactively adjust the volatility surface through intelligent cues (the rewards) freely available (e.g: in risk systems). As we will showcase in the coming sections, the Deep Reinforcement Learning (DRL) techniques are effective and can produce the same results as standard optimizers. They can be trained offline (for example outside market hours) or online after an initial warm-up phase.

**Traditional Approach** Traditionally, we have some objective functional depending on the choice of the loss function. Since, we usually want to match the mid, we traditionally consider the squared-error loss or the hubber loss.

Setting $\sigma^{\text{spread}}(t_i, \kappa_j) := \sigma^{Ask}_{t_i,\kappa_j} - \sigma^{Bid}_{t_i,\kappa_j}$ the aggregate volatility spread and $\sigma^{\text{Mid}}(t_i, \kappa_j) := \frac{1}{2}(\sigma^{Ask}_{t_i,\kappa_j} + \sigma^{Bid}_{t_i,\kappa_j})$ the aggregate volatility mid, we define the objective to be:

- Either the squared-error loss :

$$\xi(\vec{\theta}_{t_i}) := \sqrt{\frac{\sum_{j=1}^{n} w_j \Xi(t_i, \kappa_j)^2}{\sum_{j=1}^{n} w_j}} \tag{2.1}$$

  where $\Xi(t_i, \kappa_j) := \frac{\sigma^{Mid}(t_i,\kappa_j) - \Psi^{vol}_{\vec{\theta}_{t_i}}(\kappa_j)}{\sigma^{\text{spread}}(t_i,\kappa_j)}$ and $w_j$ the Black-Scholes Vega for example.

- or the hubber loss defined by :

$$\xi(\vec{\theta}_{t_i}) := \sum_{j=1}^{n} L_\delta(\Xi(t_i, \kappa_j)) \tag{2.2}$$

  where $L_\delta(a) = \begin{cases} \frac{1}{2}a^2 & \text{for } |a| \le \delta, \\ \delta \cdot \left(|a| - \frac{1}{2}\delta\right), & \text{otherwise.} \end{cases}$

The hubber loss function is usually used to assign less penalty to the outliers compared to the error loss, and really useful when the data contains large number of outliers.

### 2.2.2 Reinforcement Learning framework for Fitting

**Parametrisation and market observables:** Let us fix a maturity T and observe the market at time $t_i$, i.e. a collection $\left\{\sigma^{Ask}_{Call/Put}(t_i, \kappa_j), \sigma^{Bid}_{Call/Put}(t_i, \kappa_j)\right\}_{j\in[1,\dots,n]}$ of markets quotes for different moneyness $(\kappa_1, ..., \kappa_n)$.

[5] Multiple repricing can be required in the optimization routines.

We consider a setting with a parametrisation $\Psi^{vol}_{\vec{\theta}}$ [6] of the volatility slice with a vector of K parameters denoted by $\vec{\theta}_{t_i} = \left(\theta^1_{t_i}, ..., \theta^K_{t_i}\right)$. The market is moving over time, and at time $t_{i+1}$, this fitting problem consists of finding the optimal vector $\vec{\theta}^*_{t_{i+1}}$ which maximizes an objective function. The parametrisation $\Psi^{vol}_{\theta}$ is classically optimized to be close to the mid volatilities $\left\{\sigma^{Mid}(t_i, \kappa_j)\right\}_{j\in[1,...,n]}$ with $\sigma^{Mid}(t_i, \kappa_j) = \frac{\sigma^{Ask}(t_i,\kappa_j)+\sigma^{Bid}(t_i,\kappa_j)}{2}$ (See Figure 2.1)

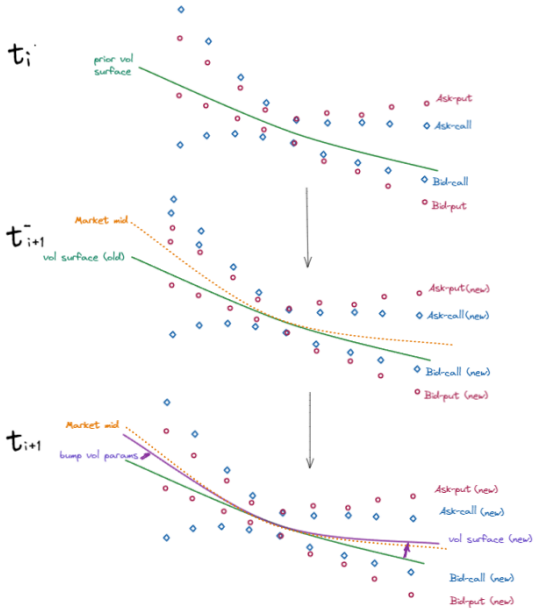



Figure 2.1: A snapshot of the agent acting in the market: The old volatility surface is bumped to a new one following the move of the market, conditional on prior volatility surface.

In the RL framework, we leave the problem open to more than just matching the mid as much as possible. In fact, our goal is to maximize a certain reward defined in term of an objective function: With a market move at time $t_{i+1}$, the challenge for an agent is to bump the parameters of our volatility slice parametrisation $\Psi^{vol}_{\vec{\theta}}$ in order to maximize a certain reward.

Subsequently, for a range of moneyness, and starting from a prior parametrized volatility surface $\Psi^{vol}_{\vec{\theta}_{t_i}}$ ( i.e. the latest fit is the prior input), the problem is to find the shifted vector $\Delta\vec{\theta}_{t_i} = \left(\Delta\theta^1_{t_i}, ..., \Delta\theta^K_{t_i}\right)$ to bump the old parameters $\vec{\theta}_{t_i}$ in order to maximize our selected reward function.

[6] We can think about the Stochastic volatility inspired (SVI) parameterization of the total implied variance for a fixed time to maturity proposed by Gatheral [28, 26, 29].

$$\Psi^{SVI}_{\vec{\theta}}(k)^2 = a + b(\rho(k-m) + \sqrt{(k-m)^2 + \sigma^2})$$

where $k$ is the log-forward moneyness, and $\vec{\theta} = (a, b, \rho, m, \sigma)$.

**State and Action spaces:** Our state is the set of tuple consisting of the observed markets quotes and the prior volatility parameters:

$$s_{t_i} = \left(\sigma^{Bid}(t_i, \kappa_j), \sigma^{Ask}(t_i, \kappa_j), \theta^1_{t_{i-1}}, ..., \theta^K_{t_{i-1}}\right)_{j\in[1,n]} \in \mathbb{R}^N$$

(with $N \geq 2n + K$), where we consider the bid and ask market implied volatility, i.e. the state space is continuous. [7]

From the RL perspective, the fitting cycle looks as follows: At time $t^-_{i+1}$,

1. The agent sees the state $t_{i+1}$ of our environment built on the market bid and ask volatilities (homogeneous variable) and the old volatility surface parameters $\vec{\theta}_{t_i}$ (endogenous variable),

2. It takes some action $a_{t_{i+1}} = \Delta\vec{\theta_{t_{i+1}}} = \theta_{t_{i+1}} - \theta_{t_i}$ on the adjustment of the old parameters $\vec{\theta}_{t_i}$, and receives some reward $r(s_{t_{i+1}}, a_{t_{i+1}} = \Delta\vec{\theta}_{t_{i+1}})$.

3. It boils down a new state for which the resulting volatility slice maximizes the reward.

The error term between our volatility surface and the mid market is denoted $\xi(\vec{\theta}_{t_{i+1}})$ .

Since the vector of bumps $\Delta\vec{\theta}$ ( resp. the actions) can take any values in $\mathbb{R}^K$ ( resp. $\mathbb{R}^K$) space, it boils down a continuous actions spaces.

**Reward functions:** We use two metrics to measure the goodness of the fit: [8]

- Mean squared error (MSE) defined by :

$$\xi(\vec{\theta}_{t_i}) := \sum_{j=1}^{n} (\sigma^{Mid}(t_i, \kappa_j) - \Psi^{vol}_{\vec{\theta}_{t_i}}(k_i))^2 \tag{2.3}$$

  It is used to penalise larger deviations from the true values, making it an effective metric for assessing the quality of fit.

- Black Scholes Vega weighted Mean squared error (BMSE) defined by :

$$\xi(\vec{\theta}_{t_i}) := \sum_{j=1}^{n} \text{vega}_{BS}(\kappa_j)(\sigma^{Mid}(t_i, \kappa_j) - \Psi^{vol}_{\vec{\theta}_{t_i}}(\kappa_j))^2 \tag{2.4}$$

  It is used to penalise markets quotes located at the neighbourhood of the at-the-money region.

[7] We may also rather consider $\left(\sigma^{Mid}(t_i, \kappa_j), \sigma^{\text{spread}}(t_i, \kappa_j), \theta^1_{t_{i-1}}, ..., \theta^K_{t_{i-1}}\right)_j \in \mathbb{R}^N$ (with $N \geq 2n + K$), where $\sigma^{\text{spread}}(t_i, \kappa_j) := \sigma^{Ask}(t_i, \kappa_j) - \sigma^{Bid}(t_i, \kappa_j)$ is the bid-ask volatility spread.

[8] We can also design a reward functional taking into account a set of constrains like the liquidity, macro-events, term-structure, stability.

**Remark** : We can also use the Scaled Mean squared error (SMSE) [9]to give more importance to market quotes with small bid-ask spread.

Within all these cases, we defined the reward as the opposite of that error term:
$r(s_{t_i}, a_{t_i} = \Delta\vec{\theta}_{t_i}) = -\xi(\vec{\theta}_{t_i})$

## 2.3 Algorithms Overview For Volatility Fitting

In this setting, the state-action value can be written as:

$$Q^{\pi}(s_{t_i}, a_{t_i} = \Delta\vec{\theta}_{t_i}) = -\sum_{k=t_i}^{T} \mathbb{E}_{(s_k,a_k)\sim\rho_\pi}[\gamma^{(k-t_i)}\xi(\vec{\theta}_k)]. \tag{2.6}$$

We adapt the framework of DRL technique and provide a natural extension of DDPG and SAC to the volatility fitting problem, and adjust our exploration preference (decaying the standard deviation of the exploration noise and the temperature coefficient with steps as we draw close to the optimal reward). We also choose to store only transitions which improve the replay memory i.e. for which the reward is greater that the worst reward in the replay buffer.

Note that the sampled minibatch from the replay buffer in our algorithms is a state-action-reward-next state tuple of the form

$$(s_{t_1}, a_{t_1}, r_{t_1}, s_{t_2}), \ldots, (s_{t_K}, a_{t_K}, r_{t_K}, s_{t_K+1}).$$

Using the observed markets quotes and the prior volatility parameters to represent the market state present the challegenge of feature scaling regarding their order of magnitude. In this context, unlike supervised learning, we cannot simply rescaled the features in the naive way since the a-priori distribution of states under the optimal policy is unknown. We used batch normalisation(dynamic feature scaling or running means and standard deviation) of states as solution to this problem [24].

In fact the input or physical statistics are different by nature and even statistics of the same type may vary a lot across multiple batchs. Batch normalization is applied to fix it by normalizing every dimension across samples in one minibatch.

The critic and actor networks are then updated according to these samples with a normalization of layer inputs as proposed in [24] to reduce the internal covariate shift. The reader should refer to the appendix 3.3 for the hyper-parameters used for the training.

We use a benchmark optimizer for performance comparison. For the actor and the critic, we use a two layer feedwordward neural network with n and m units in each layer respectively, and ReLu, Tanh activations.

To stabilize the estimates of the target values, we use a soft update rule which slowly transfers the weights via Polyak averaging$\bar{\phi} = \tau\phi + (1-\tau)\bar{\phi}$ for some small $\tau$ . In this way, the target

[9]Setting $\sigma^{\text{spread}}(t_i, \kappa_j) := \sigma^{Ask}_{t_i,\kappa_j} - \sigma^{Bid}_{t_i,\kappa_j}$ the volatility spread, we define the SMSE :

$$\xi(\vec{\theta}_{t_i}) := \sum_{j=1}^{n} \left(\frac{\sigma^{Mid}(t_i, \kappa_j) - \Psi^{vol}_{\vec{\theta}_{t_i}}(\kappa_j)}{\sigma^{\text{spread}}(t_i, \kappa_j)}\right)^2 \tag{2.5}$$

network values are constrained to change slowly, different from the design in DQN that the target network stays frozen for some period of time.

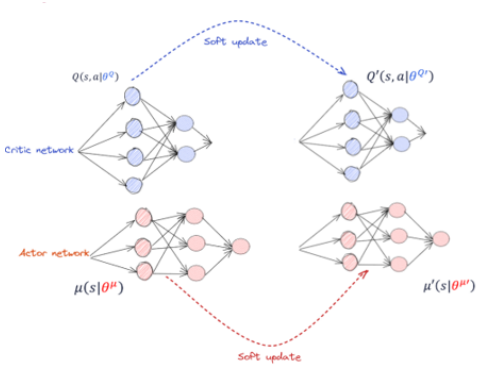



Figure 2.2: None stability for critic value network – Target networks and soft update.

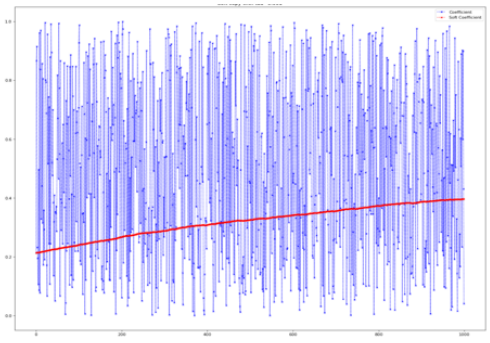
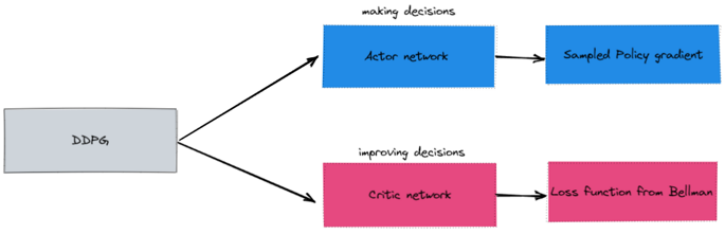


Figure 2.3: stability and soft update.

### 2.3.1 Deep deterministic policy gradient (DDPG):

In our setting (see section 2.4 ), we perform exploration using a Gaussian Noise (GN) instead of Ornstein Uhlenbeck (OU) action noise as GN is time-independent while OU noise presents some auto-correlation.




$$a_t \sim \pi^D(s_t, \theta^\pi) + \mathcal{N}$$
with $\mathcal{N} \in \mathrm{OU}(0, \sigma_n^2), \mathcal{N}(0, \sigma_n^2 I)$

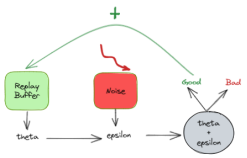



Figure 2.4: Replay Memory and Noise in the DDPG Framework.

We choose to dampen exploration by powerly decaying the standard deviation of the Gaussian Noise ( figure 2.4 ).

**Algorithm 1 DDPG variant for Volatility fitting**

**Input**: Randomly initialize an actor network $\pi^D(s;\theta)$, a critic network $Q(s,a;\phi)$,with parameters $\theta^{(0)}$ and $\phi^{(0)}$
Initialize target networks $\bar{\pi}^D$ and $\bar{Q}$ with parameters $\bar{\phi}^{(0)} \leftarrow \phi^{(0)}$ and $\bar{\theta}^{(0)} \leftarrow \theta^{(0)}$,
Initialize the replay buffer $\mathcal{RB}$
**for** $n = 0, \dots, N-1$ **do**
  Initialize a gaussian process $\mathcal{N}$ for action exploration and fix LearningFlag= True.
  Initialize state $s_0$ with a flat volatility parameters
  **for** $t = 1, \dots, M$ **do**
    Select the bumps $a_t \sim \pi^D(s_t;\theta^{(n)}) + \epsilon_t$ with $\epsilon_t \sim \mathcal{N}(0,\sigma_n)$, $\sigma_n = \max(\sigma_0(1-\frac{n}{N})^4, \sigma_{\min})$
    Execute the bumps $\pi^D(s_t;\theta^{(n)})$ (deterministic) and $a_t$(exploration), then receive reward $r_t^D$, $r_t$ and observe new state $s_{t+1}$
    Smartly Store the transition $(s_t, a_t, r_t, s_{t+1})$ in $\mathcal{RB}$ , (see Fig.2.4)
    **if** LearningFlag **then**
      Sample a random mini-batch of $N_{batch}$ transitions $\{(s_{(i)}, a_{(i)}, r_{(i)}, s_{(i+1)})\}_{i=1}^{N_{batch}}$ from $\mathcal{RB}$
      Compute the target
      $Y_i = r_i + \gamma\bar{Q}(s_{i+1}, \bar{\pi}^D(s_{i+1};\bar{\theta}^{(n)});\bar{\phi}^{(n)})$.
      Update the critic by minimizing the loss: $\phi^{(n+1)} = \phi^{(n)} - \beta\nabla_\phi \mathcal{L}_{\text{DDPG}}(\phi^{(n)})$
      Update the actor by using the sampled policy gradient: $\theta^{(n+1)} = \theta^{(n)} + \alpha\nabla_\theta J(\theta^{(n)})$
      Update the target networks via polyak averaging:

$$\begin{aligned}\bar{\phi}^{(n+1)} &\leftarrow \tau\phi^{(n+1)} + (1-\tau)\bar{\phi}^{(n)} \\ \bar{\theta}^{(n+1)} &\leftarrow \tau\theta^{(n+1)} + (1-\tau)\bar{\theta}^{(n)}\end{aligned}$$

    **end if**
    Update LearningFlag wrt to a preset criterion[10]
  **end for**
**end for**

$\nabla_\theta J(\theta^{(n)}) \approx \frac{1}{N_{batch}} \sum_{i=1}^{N_{batch}} \nabla_a Q(s,a;\phi^{(n)}) \nabla_\theta \pi^D(s;\theta^{(n)})|_{s=s_i, a=a_i}$ [11]
$\mathcal{L}_{\text{DDPG}}(\phi) = \frac{1}{N_{batch}} \sum_{i=1}^{N_{batch}} (Y_i - Q(s_i,a_i;\phi))^2$

### 2.3.2 Soft Actor Critic (SAC):

In addition to the Q-function (we use two soft Q-function to mitigate positive bias in the policy improvement step), and the policy, we learn the optimal temperature coefficient at each step by minimizing the dual objective in Equation 2.8

[10] $r_t^D > \mathcal{R}_0$ (in the static and sequential toy case) or $r_t^m > \mathcal{R}_0$(in the dynamic case) where $\mathcal{R}_0$ is a reward threshold and $r_t^m$ the mean reward of the evaluation episode(see 2.4.3 ).

[11] this analytical gradient is useless in practice with common deep learning frameworks such as pytorch since these libraries are such that writing the policy loss is enough: the gradient is automatically computed when the backward pass is performed.

In particular, we parameterize two soft Q-functions, with parameters $\theta_i$, and train them independently to optimize $J_Q(\theta_i)$. We then use the minimum of the the soft Q-functions for the stochastic gradient in Equation 1.28 and policy gradient in Equation 1.32.

**Algorithm 2 SAC variant for Volatility Fitting**

**Input**: Randomly initialize an actor network $\pi(s;\phi)$, two critics networks $Q(s,a;\theta_1)$ and $Q(s,a;\theta_2)$, with parameters $\phi^{(0)}$, $\theta_1^{(0)}$ and $\theta_2^{(0)}$.
Initialize $\alpha^{(0)}$ and set $\bar{\mathcal{H}} = -\dim(\mathcal{R}^K) = -K$
Initialize target networks $\bar{Q}$ with weights
$\bar{\theta}_1^{(0)} \leftarrow \theta_1^{(0)}$ and $\bar{\theta}_2^{(0)} \leftarrow \theta_2^{(0)}$,
Initialize an empty replay pool
$\mathcal{RB} \leftarrow \emptyset$ and fix LearningFlag= True.
**for** $n = 0, \ldots, N-1$ **do**
  Initialize state $s_0$ with a flat volatility parameters
  **for** $t = 1, \ldots, M$ **do**
    Sample the vector of bumps from the policy
    $a_t \sim \pi_\phi(a_t|s_t) := \mu_\phi(s_t) + \epsilon_t \sigma_\phi(s_t)$
    Execute the bumps $\mu_\phi(s_t)$ (deterministic) and $a_t$(exploration) , then receive rewards $r_t^D$, $r_t$ and observe new state $s_{t+1} \sim p(s_{t+1}|s_t, a_t)$
    Store the transition in the replay pool
    $\mathcal{RB} \leftarrow \mathcal{RB} \cup \{(s_t, a_t, r(s_t, a_t), s_{t+1})\}$
    **if** LearningFlag **then**
      Sample a random mini-batch of $N_{batch}$ transitions $\{(s_{(i)}, a_{(i)}, r_{(i)}, s_{(i+1)})\}_{i=1}^{N_{batch}}$ from $\mathcal{RB}$
      Compute the target $y_i =$
      $r_i + \gamma \left(\bar{Q}_\theta(s_{i+1}, a_{i+1}) - \alpha^{(n)} \log \pi(a_i|s_i;\phi^{(n)})\right)$.
      Update the Q-function parameters: [12]for $j \in \{1,2\}$ $\theta_j^{(n+1)} \leftarrow \theta_j^{(n)} - \lambda_Q \hat{\nabla}_{\theta_j} J_{Q_j}(\theta_j^{(n)})$
      Update policy weights: [13]
      $\phi^{(n+1)} \leftarrow \phi^{(n)} - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi^{(n)})$
      Adjust temperature:
      $\alpha^{(n+1)} \leftarrow \alpha^{(n)} - \lambda \hat{\nabla}_\alpha J(\alpha^{(n)})$
      Update the target networks weights via polyak averaging:
      $\bar{\theta}_j^{(n+1)} \leftarrow \tau \theta_j^{(n+1)} + (1-\tau)\bar{\theta}_j^{(n)}$ for $j \in \{1,2\}$
    **end if**
    Update LearningFlag wrt to a preset criterion[14]
  **end for**
**end for**
**Ensure:** Optimized parameters $\theta_1^*$, $\theta_2^*$ and $\phi^*$

We compute gradients for $\alpha$ with the following objective Equation 2.7:

$$J(\alpha) = \mathbb{E}_{a_t \sim \pi_t} \left[ -\alpha \log \pi_t(a_t|s_t) - \alpha \bar{\mathcal{H}} \right]. \tag{2.7}$$

We can proceed backwards in time and recursively optimize Equation 1.34. The optimal policy at time $t$ is a function of the dual variable $\alpha_t$. Similarly, we can solve the optimal dual variable $\alpha_t^*$ (the optimal temperature at step t) after solving for $Q_t^*$ and $\pi_t^*$:

$$\alpha_t^* = \arg\min_{\alpha_t} \mathbb{E}_{a_t \sim \pi_t^*} \left[ -\alpha_t \log \pi_t^*(a_t|s_t; \alpha_t) - \alpha_t \bar{\mathcal{H}} \right]. \tag{2.8}$$

## 2.4 Insights from toy-markets

In order to assess the performance of the reinforcement learning algorithms for the volatility fitting problem, we consider toy markets of increasing complexity. In particular, as we will detail in the coming lines, we start with a simple scenario where market data is static. We then relax this hypothesis to slowly account for the market dynamics.

### 2.4.1 Static Scenario

In this scenario, the market quotes are static and do not change during the full experiment. In particular, the initial state is a flat volatility surface and the fitting episode ends after one step. The state space is degenerated into a single state but the number of actions is infinite. The goal of the agent is to immediately detects the right parametrization bumps to apply in order to approach the market mid.

### 2.4.2 Sequential Scenario

In this scenario, the market quotes are static and do not change during the full experiment. In particular, the initial state is a flat volatility surface and the fitting episode ends after several steps (e.g: 50 steps [15] ). Both the state and action spaces are infinite. The goal of the agent is to detect the right parametrization bumps to apply in order to approach the market mid. Due to the effect of the discount factor, the agent is encouraged to detect the right parameter shifts from the first step.

---

[14] by minimizing the MSE loss against Bellman backup $J_{Q_j}(\theta_j^{(n)}) = \frac{1}{N_{batch}} \sum_{i=1}^{N_{batch}} (y_i - \bar{Q}(s_i, a_i; \theta_j^{(n)}))^2 for j \in \{1, 2\}$

[14] by minimizing the Entropy-regularized policy loss:
$J_\pi(\phi) = \mathbb{E}_{s_t \sim \mathcal{D}} \left[ \mathbb{E}_{\epsilon \sim \mathcal{N}(0,I)} [\alpha \log \pi_\phi(f_\phi(\epsilon; s_t)|s_t) - Q_\theta(s_t, f_\phi(\epsilon; s_t))] \right]$
$Q_\theta(s_i, a_i) = \min(Q(s_i, a_i; \theta_1^{(n)}), Q(s_i, a_i; \theta_2^{(n)}))$.

[14] same as 10

[15] This order of magnitude is in line with the number of fits that can happen between the open and the close if a fit is scheduled every 10mn.

### 2.4.3 Quasi-Dynamic Scenario

In this scenario, we allow the market to evolve freely during a full episode. In particular, the mids and spreads (for different moneyness) are random quantities with distinct marginals and a common joint distribution[16]. The means, variances and correlations are calibrated using public *"real-market"* data for a full trading day for certain stocks. This scenario is a clear cut versus the static and sequential market as:

- The dimensionality of the state space is higher as the market quotes components is changing at every step.
- The success criteria is more subtle as it doesn't only consist in approaching a single static terminal curve but to have the agent approach and track the market for several steps.

In this setting, we perform three successive operations:

1. Training Phase: we calibrate the hyper-parameters (e.g: learning rates, volatility noise, replay buffer size, etc) to increase the average reward in the evaluation episodes. The evaluation episodes are modes where all the randomness of the DRL algorithms is removed and no updates to the neural networks is performed. This training phase has a side benefit as it helps to determine a stopping threshold (for learning) for our agents.
2. Validation Phase: in this phase, with the optimal configuration of hyper-parameters and the stopping criterion for learning, we train several agents and upon completion of training, we compare the performance of different successful agents under different seeds to select the most promising candidate.
3. Testing Phase: we assess the performance of the best agent in a test episode.

## 2.5 Numerical results

In this section, we evaluate the performance obtained by the reinforcement learning algorithms [17] for the toy problems described in 2.4 .
The algorithms performance is tested against different market configurations that can be encountered in live trading activity.

### 2.5.1 Static Market

We compare the performance of the DDPG and SAC (averaged across multiple seeds) against a classical optimizer (Benchmark) in 2.4.1. We first show a summary table with the final rewards for different market configurations before displaying the implied volatility at convergence.

[16]The marginals are assimilated to normal distributions and the joint-dynamic is treated as a Gaussian copula
[17]Note: The DDPG and SAC are variants of the seminal algorithms adjusted for the volatility fitting problem.

**MSE Reward Summary:**

As can be seen from table 2.1 and 2.2, the final rewards achieved by the variants of the actor-critic algorithms is close to the one coming from the optimizer. The performance is stable across the different market configuration considerered.

Table 2.1: DDPG MSE Rewards

| type | Skew | High Smile | Inv. Smile |
|---|---|---|---|
| Bench | -0.011913 | -0.0000220 | -0.0000436 |
| seeds' avg | -0.012145 | -0.000163 | -0.000315 |

Table 2.2: SAC MSE Rewards

| type | Skew | High Smile | Inv. Smile |
|---|---|---|---|
| Bench | -0.011913 | -0.0000220 | -0.0000436 |
| seeds' avg | -0.011945 | -0.000221 | -0.001800 |

**High Smile Market (MSE):**

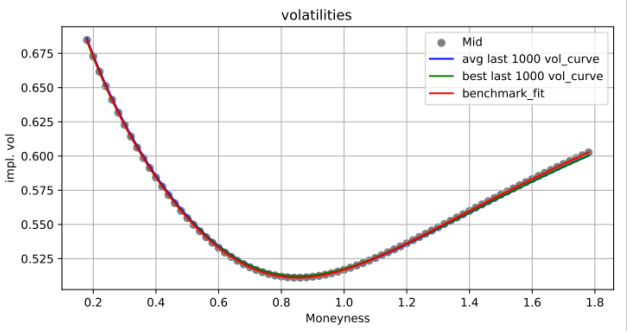



Figure 2.5: Implied volatility snapshot in a "High Smile" configuration with **DDPG**: A Monte-Carlo on 5 differents random seeds is performed with a power decaying noise. We represent in (*green*) the best response of the agent amongst the last 1000 episodes and in (*blue*) the mean of the last 1000 volatility slices.

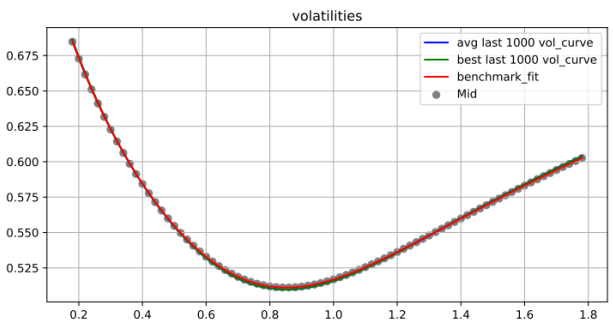



Figure 2.6: Implied volatility snapshot in a "High Smile" configuration with **SAC**: A Monte-Carlo on 5 differents random seeds. We represent in (*green*) the best response of the agent amongst the last 1000 episodes and in (*blue*) the mean of the last 1000 volatility slices.

**Skew Market (MSE):**

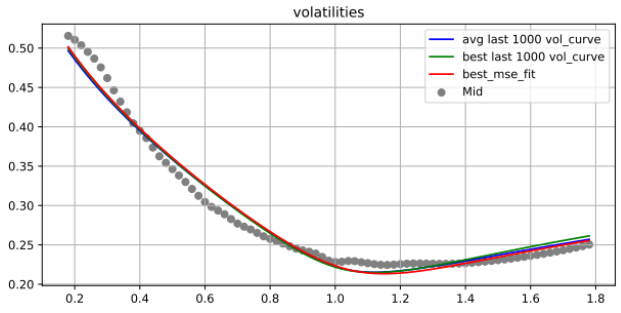



Figure 2.7: Implied volatility snapshot in a "Skewed" configuration with **DDPG**: A Monte-Carlo on 5 differents random seeds is performed with a power decaying noise. We represent in (*green*) the best response of the agent amongst the last 1000 episodes and in (*blue*) the mean of the last 1000 volatility slices.

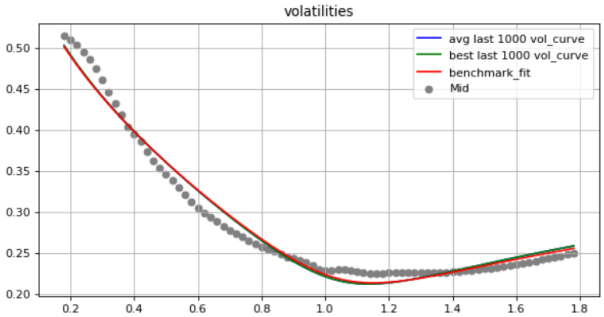



Figure 2.8: Implied volatility snapshot in a "Skewed" configuration with **SAC**: A Monte-Carlo on 5 differents random seeds. We represent in (*green*) the best response of the agent amongst the last 1000 episodes and in (*blue*) the mean of the last 1000 volatility slices.

**Inverse Smile Market (MSE):**

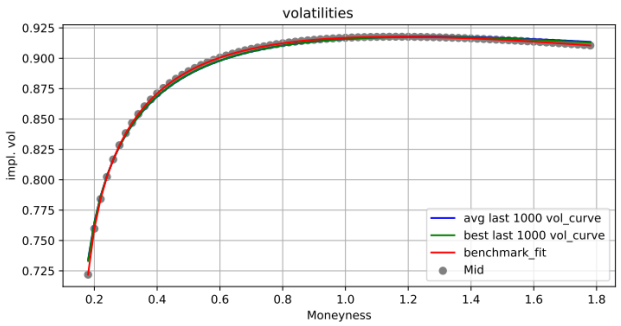



Figure 2.9: Implied volatility snapshot in a "Inverse Smile" configuration with **DDPG**: A Monte-Carlo on 5 differents random seeds is performed with a power decaying noise. We represent in (*green*) the best response of the agent amongst the last 1000 episodes and in (*blue*) the mean of the last 1000 volatility slices.

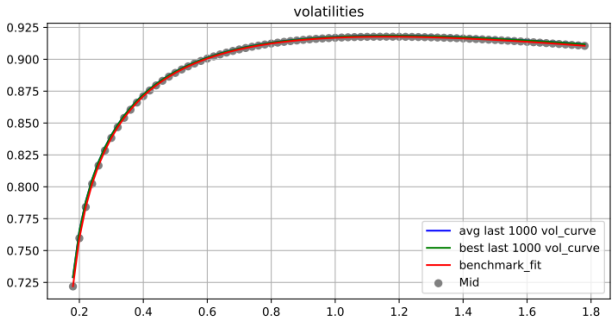



Figure 2.10: Implied volatility snapshot in a "Inverse Smile" configuration with **SAC**: A Monte-Carlo on 5 differents random seeds. We represent in (*green*) the best response of the agent amongst the last 1000 episodes and in (*blue*) the mean of the last 1000 volatility slices.

For more details on the black-scholes vega weighted reward, please refer to the **appendix**

**Black Scholes Vega Weighted MSE reward:**

**High Smile Market:**

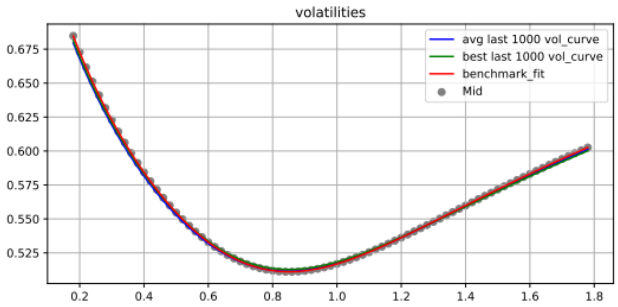



Figure 2.11: Implied volatility snapshot in a "High Smile" configuration with **DDPG**: A Monte-Carlo on 5 differents random seeds is performed with a power decaying noise. We represent in (*green*) the best response of the agent amongst the last 1000 episodes and in (*blue*) the mean of the last 1000 volatility slices..

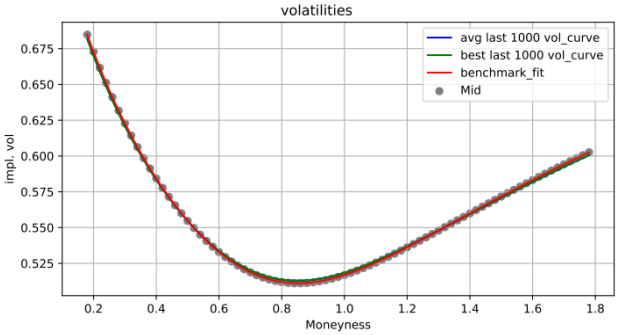



Figure 2.12: Implied volatility snapshot in a "High Smile" configuration with **SAC**: A Monte-Carlo on 5 differents random seeds. We represent in (*green*) the best response of the agent amongst the last 1000 episodes and in (*blue*) the mean of the last 1000 volatility slices.

**Skew Market:**

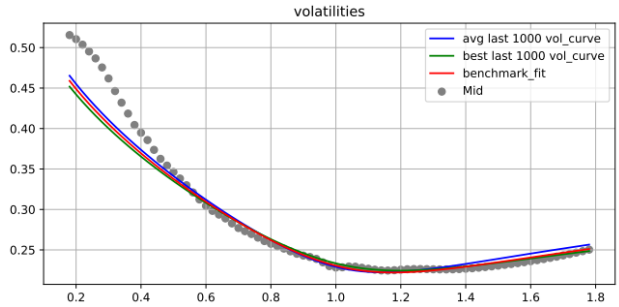



Figure 2.13: Implied volatility snapshot in a "skew Smile" configuration with **DDPG**: A Monte-Carlo on 5 differents random seeds is performed with a power decaying noise. We represent in (*green*) the best response of the agent amongst the last 1000 episodes and in (*blue*) the mean of the last 1000 volatility slices.

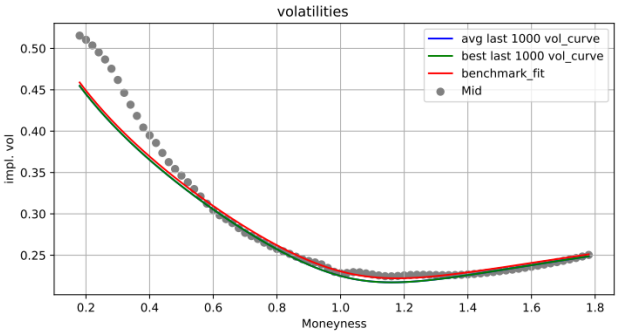



Figure 2.14: Implied volatility snapshot in a "High Smile" configuration with **SAC**: A Monte-Carlo on 5 differents random seeds. We represent in (*green*) the best response of the agent amongst the last 1000 episodes and in (*blue*) the mean of the last 1000 volatility slices.

**Inverse Smile Market:**

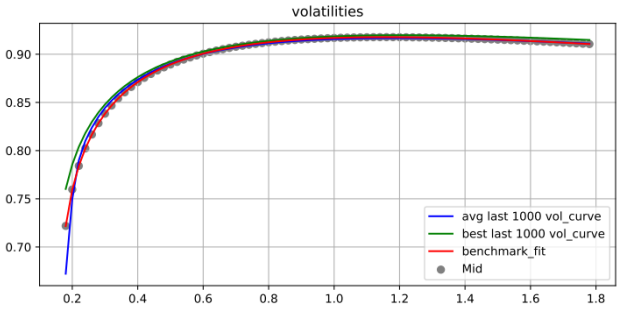



Figure 2.15: Implied volatility snapshot in an "Inverse Smile" configuration with **DDPG**: A Monte-Carlo on 5 differents random seeds is performed with a power decaying noise. We represent in (*green*) the best response of the agent amongst the last 1000 episodes and in (*blue*) the mean of the last 1000 volatility slices.

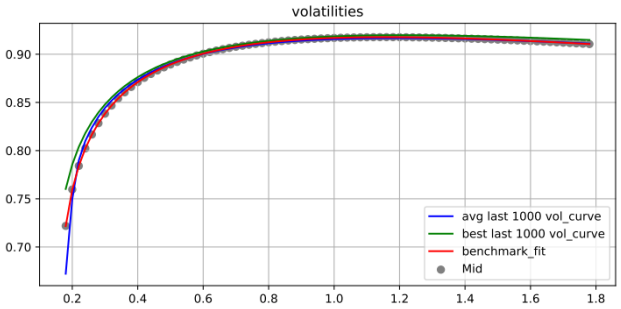



Figure 2.16: Implied volatility snapshot in a "High Smile" configuration with **SAC**: A Monte-Carlo on 5 differents random seeds. We represent in (*green*) the best response of the agent amongst the last 1000 episodes and in (*blue*) the mean of the last 1000 volatility slices.

### 2.5.2 Sequential scenario

We compare the performance of the DDPG and SAC (averaged across multiple seeds) against a classical optimizer (Benchmark) in 2.4.2. In this scenario, more than one step is allowed per episode. We first show a summary table with the final rewards for different market configurations before displaying the implied volatility at convergence.

**MSE Reward Summary:**

As can be seen from table 2.3 and 2.4, the final rewards achieved by the variants of the actor-critic algorithms is close to the one coming from the optimizer. The performance is stable across the different market configuration considerered.

Table 2.3: DDPG MSE Rewards

| type | Skew | High Smile | Inv. Smile |
|---|---|---|---|
| Bench | -0.011913 | -0.0000220 | -0.0000436 |
| seeds' avg | -0.017231 | -0.005591 | -0.001318 |

Table 2.4: SAC MSE Rewards

| type | Skew | High Smile | Inv. Smile |
|---|---|---|---|
| Bench | -0.011913 | -0.0000220 | -0.0000436 |
| seeds' avg | -0.006107 | -0.0006141 | -0.0018047 |

**Skew Market (MSE):**

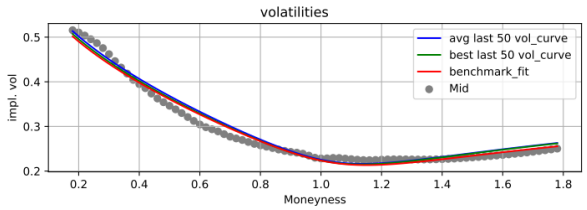



Figure 2.17: Implied volatility snapshot in a "Skewed" configuration with **DDPG**: A Monte-Carlo on 5 differents random seeds is performed with a power decaying noise. We represent in (*green*) the best response of the agent amongst the last 50 episodes and in (*blue*) the mean of the final volatility slices for the last 50 episodes.

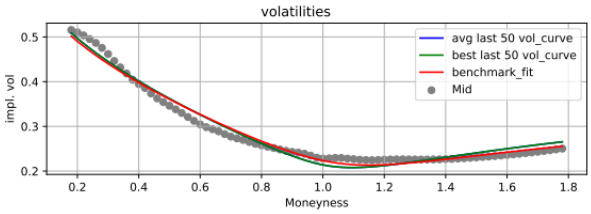



Figure 2.18: Implied volatility snapshot in a "Skewed" configuration with **SAC**: A Monte-Carlo on 5 differents random seeds is performed with automatic entropy adjustement. We represent in (*green*) the best response of the agent amongst the last 50 episodes and in (*blue*) the mean of the final volatility slices for the last 50 episodes.

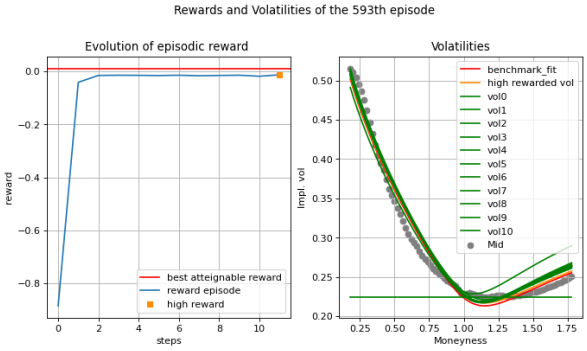



Figure 2.19: Episodic agent's Rewards and Implied volatility evolution in a "Skewed" configuration with **DDPG**.

The figure shows that, the agent is detecting the right parameter shifts from the first step.

**Smile Market (MSE):**

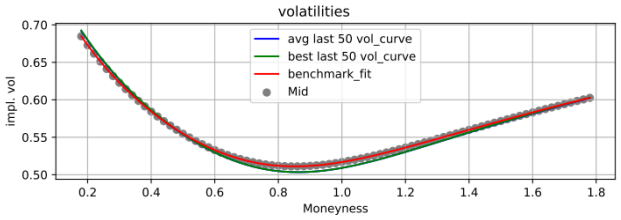



Figure 2.20: Implied volatility snapshot in a "High Smile" configuration with **DDPG**: A Monte-Carlo on 5 differents random seeds is performed with a power decaying noise. We represent in (*green*) the best response of the agent amongst the last 50 episodes and in (*blue*) the mean of the final volatility slices for the last 50 episodes.

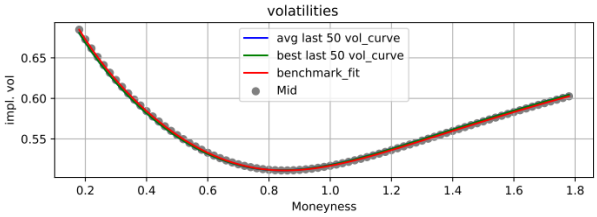



Figure 2.21: Implied volatility snapshot in a "High Smile" configuration with **SAC**: A Monte-Carlo on 5 differents random seeds is performed with automatic entropy adjustement. We represent in (*green*) the best response of the agent amongst the last 50 episodes and in (*blue*) the mean of the final volatility slices for the last 50 episodes.

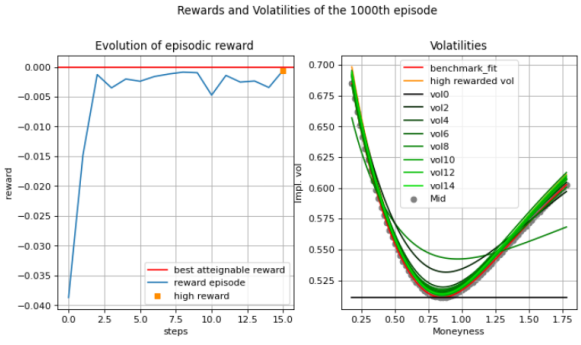



Figure 2.22: Episodic agent's Rewards and Implied volatility evolution in a "Smile" configuration with **DDPG**.

The figure shows that, the agent is detecting the right parameter shifts from the first step.

**Inverse Smile Market (MSE):**

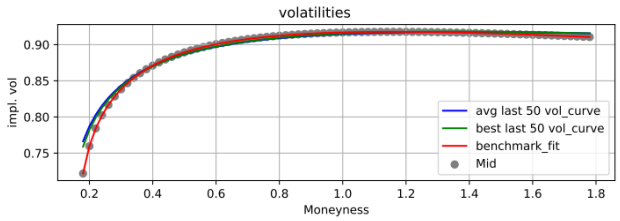



Figure 2.23: Implied volatility snapshot in an "Inverse Smile" configuration with **DDPG**: A Monte-Carlo on 5 differents random seeds is performed with a power decaying noise. We represent in (*green*) the best response of the agent amongst the last 50 episodes and in (*blue*) the mean of the final volatility slices for the last 50 episodes.

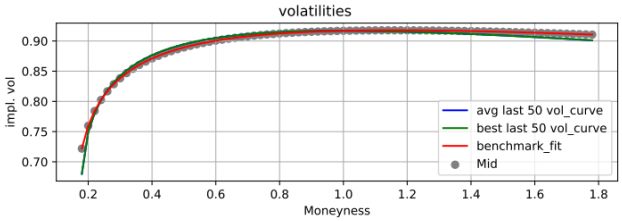
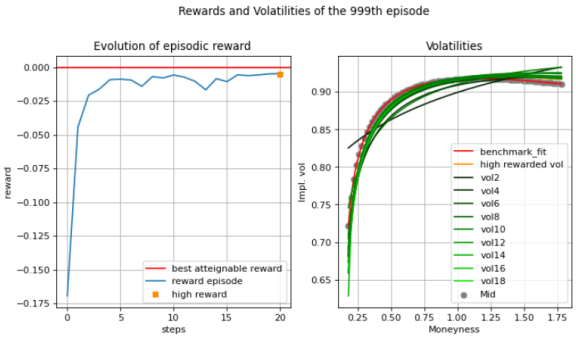



Figure 2.24: Implied volatility snapshot in an "Inverse Smile" configuration with **SAC**: A Monte-Carlo on 5 differents random seeds is performed with automatic entropy adjustement. We represent in (*green*) the best response of the agent amongst the last 50 episodes and in (*blue*) the mean of the final volatility slices for the last 50 episodes.




The figure shows that, the agent is detecting the right parameter shifts from the first step.

Figure 2.25: Episodic agent's Rewards and Implied volatility evolution in an "Inverse Smile" configuration with **DDPG**.

**Static Market, SAC algorithm and MSE reward:**

Below, best response of the agent amongst the last 1000 episodes (*blue*) and the mean of the last 1000 volatilities slices ( *red*).

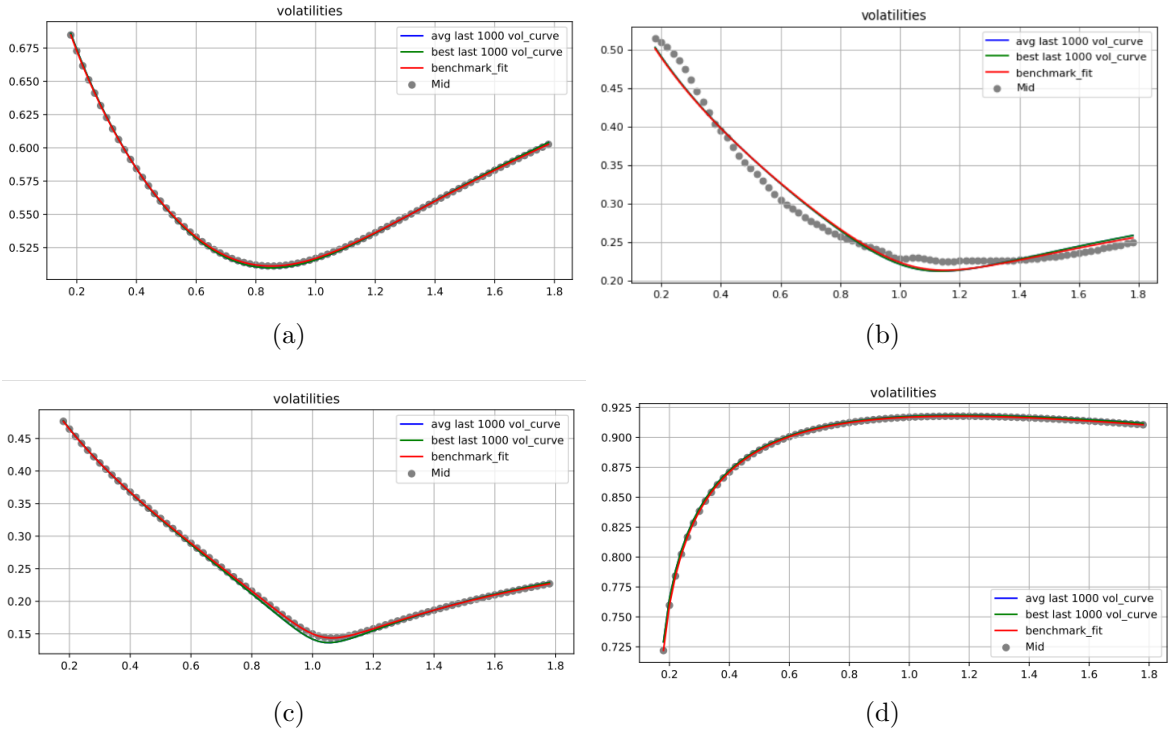



Figure 2.26: (a) High Smile (b) Skew (c) High Skew (d) Inverse Smile

**Black Scholes Vega Weighted MSE reward:**

**Smile Market:**

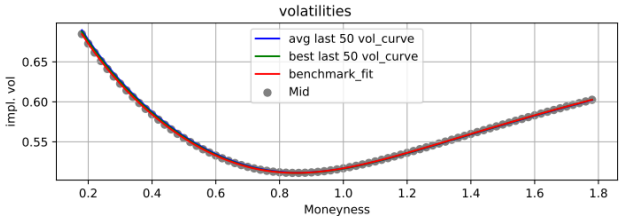



Figure 2.27: Implied volatility snapshot in a "Smile" configuration with **DDPG**: A Monte-Carlo on 5 differents random seeds is performed with a power decaying noise. We represent in (*green*) the best response of the agent amongst the last 50 episodes and in (*blue*) the mean of the final volatility slices for the last 50 episodes.

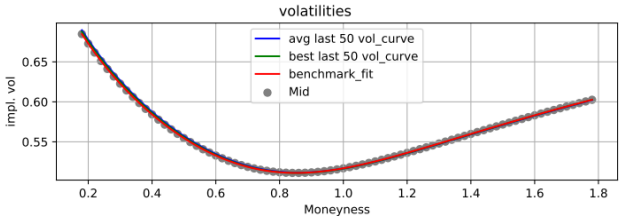



Figure 2.28: Implied volatility snapshot in a "High Smile" configuration with **SAC**: A Monte-Carlo on 5 differents random seeds is performed with automatic entropy adjustement. We represent in (*green*) the best response of the agent amongst the last 50 episodes and in (*blue*) the mean of the final volatility slices for the last 50 episodes.

**Skew Market:**

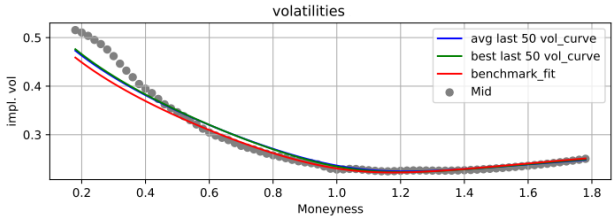



Figure 2.29: Implied volatility snapshot in a "Skew" configuration with **DDPG**: A Monte-Carlo on 5 differents random seeds is performed with a power decaying noise. We represent in (*green*) the best response of the agent amongst the last 50 episodes and in (*blue*) the mean of the final volatility slices for the last 50 episodes.

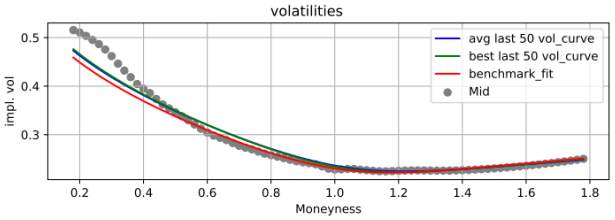



Figure 2.30: Implied volatility snapshot in a "High Smile" configuration with **SAC**: A Monte-Carlo on 5 differents random seeds is performed with automatic entropy adjustement. We represent in (*green*) the best response of the agent amongst the last 50 episodes and in (*blue*) the mean of the final volatility slices for the last 50 episodes.

**Inverse Smile Market (MSE):**

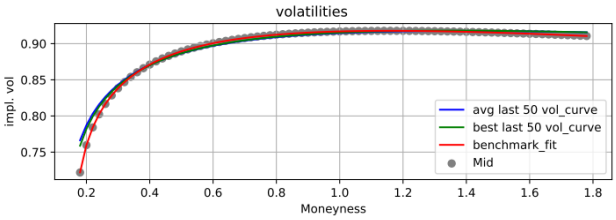



Figure 2.31: Implied volatility snapshot in an "Inverse Smile" configuration with **DDPG**: A Monte-Carlo on 5 differents random seeds is performed with a power decaying noise. We represent in (*green*) the best response of the agent amongst the last 50 episodes and in (*blue*) the mean of the final volatility slices for the last 50 episodes.

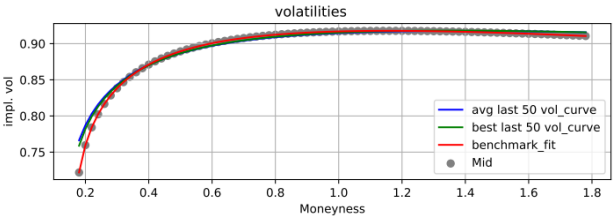



Figure 2.32: Implied volatility snapshot in a "High Smile" configuration with **SAC**: A Monte-Carlo on 5 differents random seeds is performed with automatic entropy adjustement. We represent in (*green*) the best response of the agent amongst the last 50 episodes and in (*blue*) the mean of the final volatility slices for the last 50 episodes.

**Practitioner's corner:** SAC typically converges faster and more consistently across different environments. Moreover the introduction of entropy helps mitigate over fitting during training. While robust, its require handling both a stochastic policy and an entropy regularization term. In our settings where fine continuous control over actions is important, DDPG version might be a better choice since it may be more straightforward to integrate and tune for a quasi dynamic environment (which might be more sensitive to precise adjustments). It also has less

computational overhead and resource usage.

### 2.5.3 Quasi-dynamic scenario

In this section we present the training, validation and testing phases of RL agents using **DDPG** algorithm. The agents are trained on data generated for two stocks (one with a wide spread and the other with a tight spread) across several episodes, each consisting of 50 steps. While the convergence rate of DDPG can be slow due to its sensibility to hyperparameters, Fine-tuning is often crucial for successful training.

**Training Phase:** The agents are trained to maximize the average reward during evaluation episodes within a game. In those evaluation episodes, all sources of randomness in the DRL algorithms are turned off and no updates to the neural networks are made. We construct a hypercube of hyper-parameters (e.g: learning rates, volatility noise, replay buffer size, batch size, etc). For each tuple or hyper-parameter combination, we track the evolution of average rewards (25% quantile excluded) during the evaluation process. This method enables us to identify the optimal set of hyper-parameters, and we define the stopping criterion for training as the highest average reward obtained for the optimal configuration. (see **??**).

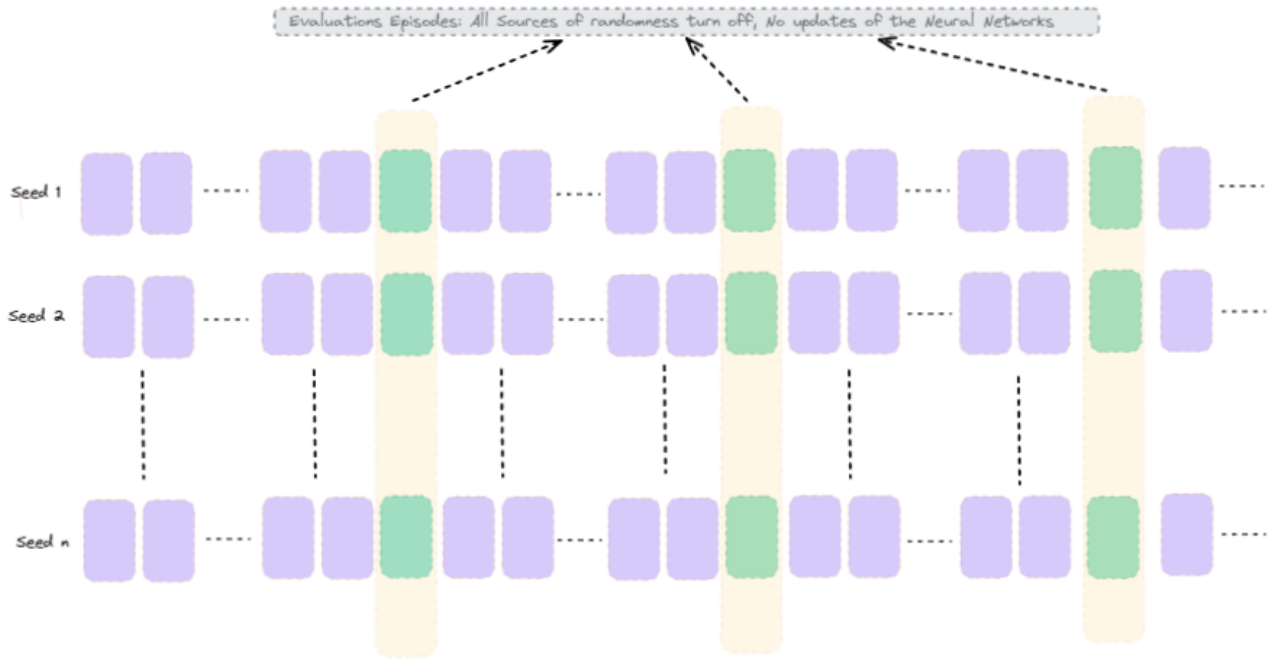



Figure 2.33: Training and Consistency over random seeds.

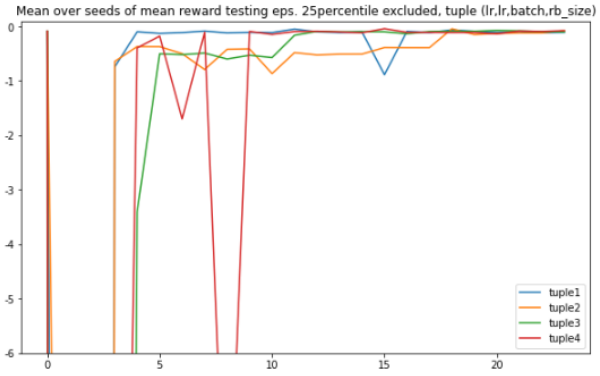



Figure 2.34: Mean Test-Eval episodic rewards for hyper-parameters search.

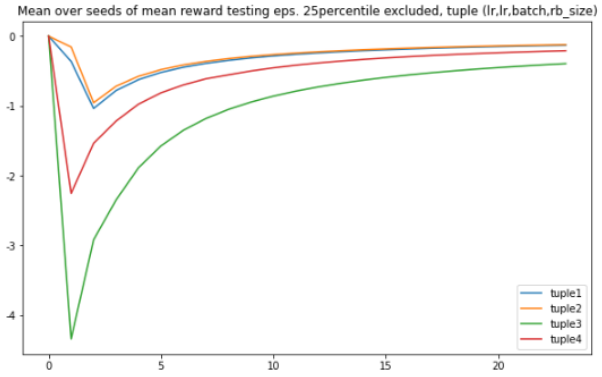



Figure 2.35: Cumulative Mean Test-Eval episodic rewards for hyper-parameters search.

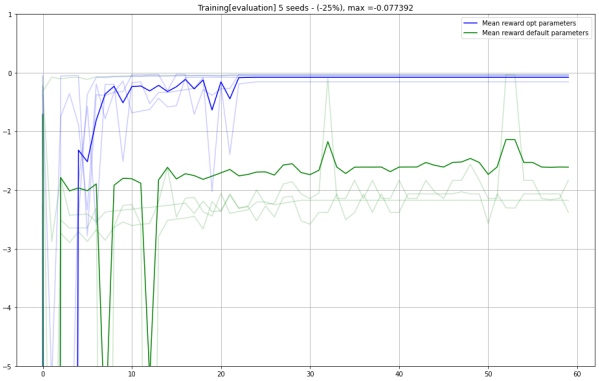



Figure 2.36: Definition of the stopping criterion.

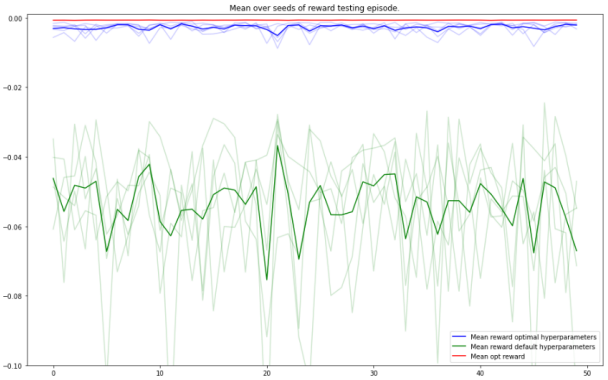



Figure 2.37: Comparaison of mean episodic rewards accross multiple random seeds for agents trained with default versus optimized hyper-parameters.

As illustrated in Figure2.37, we compute the average evaluation rewards across multiple random seeds to identify the optimal set of hyper-parameters and establish the stopping criterion. The curves shown in Figure **??** represent the mean results across seeds, ensuring a robust, stable and consistent process throughout the validation phase.

Once the optimal hyper-parameters are determined, we can compare the mean testing episodic rewards(averaged across a different set of random seeds) for agents trained with default versus optimized hyper-parameters. Figure 2.37 demonstrates the substantial improvement achieved when fine-tuning hyper-parameters compared to the default setup.

**Validation Phase:** upon completion of training, we evaluate the performance of distinct RL agents using *different random seeds* within a validation episode. We compute the mean reward for each agent, as shown in the figures below (2.38 and 2.39). This validation phase allows us to identify agents that successfully reach or surpass the pre-determined reward threshold.

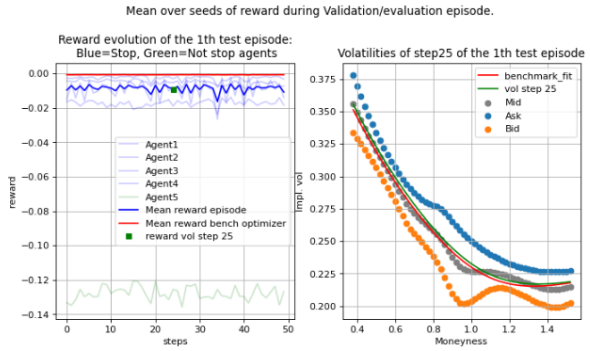



Figure 2.38: Evolution of mean episodic rewards(over several random seeds) and Implied volatility(of one selected step) for a wide spread stock with **DDPG**.

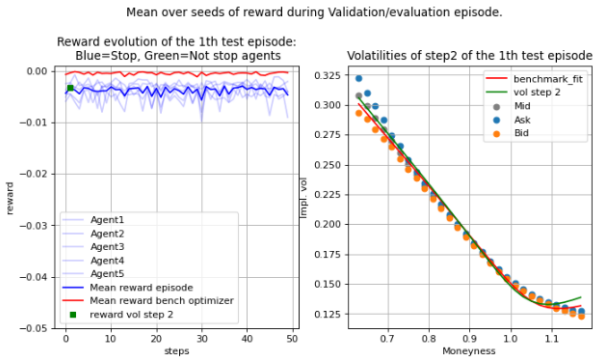



Figure 2.39: Evolution of mean episodic rewards(over several random seeds) and Implied volatility(of one selected step) for a tight spread stock with **DDPG**.

**Testing Phase:** In the final testing phase, we select the agent with the best validation performance and assess its behavior in a test episode of 50 steps for different random seed. Figures 2.40 and 2.41 display the evolution of agent's rewards and implied volatility(of one selected step) during the testing phase for stocks with wide and tight spreads respectively.

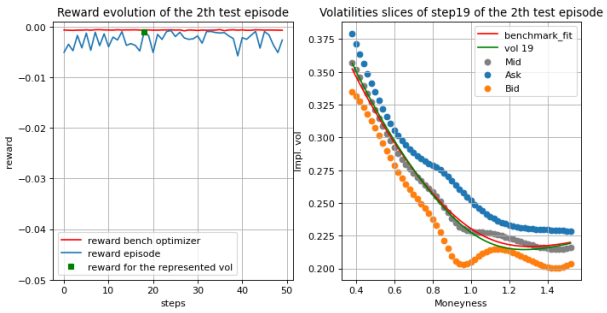



Figure 2.40: Evolution of agent's episodic rewards and Implied volatility(of one selected step) for a wide spread stock during testing with **DDPG**.

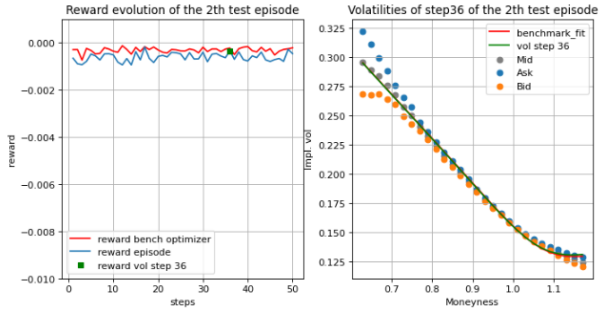



Figure 2.41: Evolution of agent's episodic rewards and Implied volatility(of one selected step) for a tight spread stock during testing with **DDPG**.

As can be seen from figures 2.40 and 2.41, the performance of the best agent under both tigh and wide spread is satisfactory. The agent is able to track the market through time keeping the reward high (close to the optimal level for the number of parametrization coefficients used in the test). It is also important to highlight that there is also a good margin of improvement by deepening the hyperparametes search, extending the training period and optimization the scoring metric in the evaluation episodes.

## 2.6 Conclusion and Future works

In this work, we proposed a model-free deep reinforcement learning architecture to solve the dynamic volatility fitting problem in continuous state and action spaces. We showed that DRL algorithms are natively tailored for the volatility fitting problem as they possess (a) native exploration (b) effective catalog of past experiences and (c) good predictive power in large spaces. Using toy problems with increasing complexity, we showed that deep reinforcement learning algorithms can achieve satisfactory performance.

Traditional methods such as least squares fitting and gradient-based optimisation techniques, often struggle with complex state spaces and non-linearities. In contract RL's ability to learn from the interactions with the continuous environment allows it to progressively improve its performance by exploring and exploiting patterns that emerges from the data. Moreover, our approach highlights the benefits of rewards shaping and careful environment design, which allow the RL agent to learn efficiently in complex market conditions.

This work therefore lays the groundwork for a full AI-based dynamic volatility fitting. Future work can extend the fitting to larger parametrizations and reflects stylistic effects in volatility surface term-structure.

# Chapter 3

# Appendix

## 3.1 Automatic Entropy Adjustment

### 3.1.1 Complete Proof

$$\max_{\pi_0,\dots,\pi_T} \mathbb{E}\Big[\sum_{t=0}^{T} r(s_t,a_t)\Big] \text{s.t. } \forall t,\, \mathcal{H}(\pi_t) \geq \mathcal{H}_0$$

where $\mathcal{H}_0$ is a predefined minimum policy entropy threshold. The expected return $\mathbb{E}\Big[\sum_{t=0}^{T} r(s_t,a_t)\Big]$ can be decomposed into a sum of rewards at all the time steps. Because the policy $\pi_t$ at time t has no effect on the policy at the earlier time step, $\pi_{t-1}$, we can maximize the return at different steps backward in time; this is essentially **DP**.

$$\underbrace{\max_{\pi_0}\Big(\mathbb{E}[r(s_0,a_0)] + \underbrace{\max_{\pi_1}\Big(\mathbb{E}[...] + \underbrace{\max_{\pi_T}\mathbb{E}[r(s_T,a_T)]}_{\text{1st maximization}}\Big)}_{\text{second but last maximization}}\Big)}_{\text{last maximization}}$$

where we consider $\gamma = 1$. So we start the optimization from the last timestep $T$:

$$\text{maximize } \mathbb{E}_{(s_T,a_T)\sim\rho_\pi}[r(s_T,a_T)] \text{ s.t. } \mathcal{H}(\pi_T) - \mathcal{H}_0 \geq 0$$

First, let us define the following functions:

$$h(\pi_T) = \mathcal{H}(\pi_T) - \mathcal{H}_0 = \mathbb{E}_{(s_T,a_T)\sim\rho_\pi}[-\log \pi_T(a_T|s_T)] - \mathcal{H}_0$$
$$f(\pi_T) = \begin{cases} \mathbb{E}_{(s_T,a_T)\sim\rho_\pi}[r(s_T,a_T)], & \text{if } h(\pi_T) \geq 0 \\ -\infty, & \text{otherwise} \end{cases}$$

And the optimization becomes:

$$\text{maximize } f(\pi_T) \text{ s.t. } h(\pi_T) \geq 0$$

### 3.1.2 Lagrangian Relaxation and Taylor expansion

we use the Lagrangian relaxation to transform the divergence constraint into a penalty, yielding an expression that is easier to solve.

To solve the maximization optimization with inequality constraint, we can construct a Lagrangian expression ([33] )with a Lagrange multiplier (also known as dual variable), $\alpha_T$:

$$L(\pi_T, \alpha_T) = f(\pi_T) + \alpha_T h(\pi_T)$$

Considering the case when we try to minimize $L(\pi_T, \alpha_T)$ with respect to $\alpha_T$ given a particular value $\pi_T$,

- If the constraint is satisfied, $h(\pi_T) \geq 0$, at best we can set $\alpha_T = 0$ since we have no control over the value of $f(\pi_T)$. Thus, $L(\pi_T, 0) = f(\pi_T)$.
- If the constraint is invalidated, $h(\pi_T) < 0$, we can achieve $L(\pi_T, \alpha_T) \to -\infty$ by taking $\alpha_T \to \infty$. Thus, $L(\pi_T, \infty) = -\infty = f(\pi_T)$.

In either case, we can recover the following equation,

$$f(\pi_T) = \min_{\alpha_T \geq 0} L(\pi_T, \alpha_T)$$

At the same time, we want to maximize $f(\pi_T)$,

$$\max_{\pi_T} f(\pi_T) = \min_{\alpha_T \geq 0} \max_{\pi_T} L(\pi_T, \alpha_T)$$

Therefore, to maximize $f(\pi_T)$, the dual problem is listed as below. Note that to make sure $\max_{\pi_T} f(\pi_T)$ is properly maximized and would not become $-\infty$, the constraint has to be satisfied.

$$\begin{aligned}
\max_{\pi_T} \mathbb{E}[r(s_T, a_T)] &= \max_{\pi_T} f(\pi_T) \\
&= \min_{\alpha_T \geq 0} \max_{\pi_T} L(\pi_T, \alpha_T) \\
&= \min_{\alpha_T \geq 0} \max_{\pi_T} f(\pi_T) + \alpha_T h(\pi_T) \\
&= \min_{\alpha_T \geq 0} \max_{\pi_T} \mathbb{E}_{(s_T, a_T) \sim \rho_\pi}[r(s_T, a_T)] + \alpha_T(\mathbb{E}_{(s_T, a_T) \sim \rho_\pi}[-\log \pi_T(a_T|s_T)] - \mathcal{H}_0) \\
&= \min_{\alpha_T \geq 0} \max_{\pi_T} \mathbb{E}_{(s_T, a_T) \sim \rho_\pi}[r(s_T, a_T) - \alpha_T \log \pi_T(a_T|s_T)] - \alpha_T \mathcal{H}_0 \\
&= \min_{\alpha_T \geq 0} \max_{\pi_T} \mathbb{E}_{(s_T, a_T) \sim \rho_\pi}[r(s_T, a_T) + \alpha_T \mathcal{H}(\pi_T) - \alpha_T \mathcal{H}_0]
\end{aligned}$$

We could compute the optimal $\pi_T$ and $\alpha_T$ iteratively. First given the current $\alpha_T$, get the best policy $\pi_T^*$ that maximizes $L(\pi_T^*, \alpha_T)$. Then plug in $\pi_T^*$ and compute $\alpha_T^*$ that minimizes $L(\pi_T^*, \alpha_T)$. Assuming we have one neural network for policy and one network for temperature parameter, the iterative update process is more aligned with how we update network parameters during training.

$$\pi_T^* = \arg\max_{\pi_T} \mathbb{E}_{(s_T,a_T)\sim\rho_\pi}[r(s_T,a_T) + \alpha_T\mathcal{H}(\pi_T) - \alpha_T\mathcal{H}_0]$$
$$\alpha_T^* = \arg\min_{\alpha_T\geq 0} \mathbb{E}_{(s_T,a_T)\sim\rho_{\pi^*}}[\alpha_T\mathcal{H}(\pi_T^*) - \alpha_T\mathcal{H}_0]$$

$$\alpha_T^* = \arg\min_{\alpha_T\geq 0} \mathbb{E}_{(s_T,a_T)\sim\rho_{\pi^*}}[\alpha_T\mathcal{H}(\pi_T^*) - \alpha_T\mathcal{H}_0]. \tag{3.1}$$

$$\text{Thus, } \max_{\pi_T} \mathbb{E}[r(s_T,a_T)] = \mathbb{E}_{(s_T,a_T)\sim\rho_{\pi^*}}[r(s_T,a_T) + \alpha_T^*\mathcal{H}(\pi_T^*) - \alpha_T^*\mathcal{H}_0]$$

Now let's go back to the soft Q value function:

$$\begin{aligned}
Q_{T-1}(s_{T-1},a_{T-1}) &= r(s_{T-1},a_{T-1}) + \mathbb{E}[Q(s_T,a_T) - \alpha_T\log\pi(a_T|s_T)] \\
&= r(s_{T-1},a_{T-1}) + \mathbb{E}[r(s_T,a_T)] + \alpha_T\mathcal{H}(\pi_T) \\
Q_{T-1}^*(s_{T-1},a_{T-1}) &= r(s_{T-1},a_{T-1}) + \max_{\pi_T}\mathbb{E}[r(s_T,a_T)] + \alpha_T\mathcal{H}(\pi_T^*) \qquad \text{; plug in the optimal } \pi_T^*
\end{aligned}$$

Therefore the expected return is as follows, when we take one step further back to the time step $T-1$:

$$\begin{aligned}
\text{Objt.} &= \max_{\pi_{T-1}}\Big(\mathbb{E}[r(s_{T-1},a_{T-1})] + \max_{\pi_T}\mathbb{E}[r(s_T,a_T]\Big) \\
&= \max_{\pi_{T-1}}\Big(Q_{T-1}^*(s_{T-1},a_{T-1}) - \alpha_T^*\mathcal{H}(\pi_T^*)\Big) \quad \text{s.t. } \mathcal{H}(\pi_{T-1}) - \mathcal{H}_0 \geq 0 \\
&= \min_{\alpha_{T-1}\geq 0}\max_{\pi_{T-1}}\Big(Q_{T-1}^*(s_{T-1},a_{T-1}) - \alpha_T^*\mathcal{H}(\pi_T^*) + \alpha_{T-1}\big(\mathcal{H}(\pi_{T-1}) - \mathcal{H}_0\big)\Big) \quad \text{dual problem w/ Lagrangian.} \\
&= \min_{\alpha_{T-1}\geq 0}\max_{\pi_{T-1}}\Big(Q_{T-1}^*(s_{T-1},a_{T-1}) + \alpha_{T-1}\mathcal{H}(\pi_{T-1}) - \alpha_{T-1}\mathcal{H}_0\Big) - \alpha_T^*\mathcal{H}(\pi_T^*)
\end{aligned}$$

Similar to the previous step,

$$\pi_{T-1}^* = \arg\max_{\pi_{T-1}} \mathbb{E}_{(s_{T-1},a_{T-1})\sim\rho_\pi}[Q_{T-1}^*(s_{T-1},a_{T-1}) + \alpha_{T-1}\mathcal{H}(\pi_{T-1}) - \alpha_{T-1}\mathcal{H}_0]$$

$$\pi_{T-1}^* = \arg\min_{\alpha_{T-1}\geq 0} \mathbb{E}_{(s_{T-1},a_{T-1})\sim\rho_{\pi^*}}[\alpha_{T-1}\mathcal{H}(\pi_{T-1}^*) - \alpha_{T-1}\mathcal{H}_0]. \tag{3.2}$$

The equation for updating $\alpha_{T-1}$ in 3.1 has the same format as the equation for updating $\alpha_{T-1}$ in 3.2 above. By repeating this process, we can learn the optimal temperature parameter in every step by minimizing the same objective function:

$$J(\alpha) = \mathbb{E}_{a_t\sim\pi_t}[-\alpha\log\pi_t(a_t \mid s_t) - \alpha\mathcal{H}_0]$$

The final algorithm is same as SAC except for learning $\alpha$ explicitly with respect to the objective $J(\alpha)$ (see Fig. 7):

## 3.2 Policy Gradient Theorem

Proof of Theorem 1
Let $d^\pi(s) = \lim_{t\to\infty} P(s_t = s|s_0, \pi_\theta)$ is the probability that $s_t = s$ when starting from $s_0$ and following policy $\pi_\theta$ for t steps.($d^\pi(s)$ is the stationary distribution of Markov chain for $\pi_\theta$ (on-policy state distribution under $\pi$). i.e. when the probability of you ending up with one state becomes unchanged when you travel along the Markov chain's states for $\pi_\theta$ ) . The objective function is defined as:

$$J(\theta) = \sum_{s\in\mathcal{S}} d^\pi(s)V^\pi(s) = \sum_{s\in\mathcal{S}} d^\pi(s) \sum_{a\in\mathcal{A}} \pi_\theta(a|s)Q^\pi(s,a)$$

Policy Gradient Theorem provides a nice reformation of the derivative of the objective function and simplify the gradient computation $\nabla_\theta J(\theta)$

$$\begin{aligned}\nabla_\theta J(\theta) &= \nabla_\theta \sum_{s\in\mathcal{S}} d^\pi(s) \sum_{a\in\mathcal{A}} Q^\pi(s,a)\pi_\theta(a|s) \\ &\propto \sum_{s\in\mathcal{S}} d^\pi(s) \sum_{a\in\mathcal{A}} Q^\pi(s,a)\nabla_\theta\pi_\theta(a|s)\end{aligned}$$

We first start with the derivative of the state value function:

$$\begin{aligned}\nabla_\theta V^\pi(s) =& \nabla_\theta\Big(\sum_{a\in\mathcal{A}} \pi_\theta(a|s)Q^\pi(s,a)\Big) \\ =& \sum_{a\in\mathcal{A}} \Big(\nabla_\theta\pi_\theta(a|s)Q^\pi(s,a) + \pi_\theta(a|s)\nabla_\theta Q^\pi(s,a)\Big) && \text{by Derivative product rule.} \\ =& \sum_{a\in\mathcal{A}} \Big(\nabla_\theta\pi_\theta(a|s)Q^\pi(s,a) + \pi_\theta(a|s)\nabla_\theta \sum_{s',r} P(s',r|s,a)(r + V^\pi(s'))\Big) && \text{Extend } Q^\pi \text{ with future state value.} \\ =& \sum_{a\in\mathcal{A}} \Big(\nabla_\theta\pi_\theta(a|s)Q^\pi(s,a) + \pi_\theta(a|s) \sum_{s',r} P(s',r|s,a)\nabla_\theta V^\pi(s')\Big) && P(s',r|s,a) \text{ or } r \text{ is not a function of } \theta \\ =& \sum_{a\in\mathcal{A}} \Big(\nabla_\theta\pi_\theta(a|s)Q^\pi(s,a) + \pi_\theta(a|s) \sum_{s'} P(s'|s,a)\nabla_\theta V^\pi(s')\Big) && \text{Since } P(s'|s,a)=\textstyle\sum_r P(s',r|s,a)\end{aligned}$$

We then have:

$$\nabla_\theta V^\pi(s) = \sum_{a\in\mathcal{A}} \Big(\nabla_\theta\pi_\theta(a|s)Q^\pi(s,a) + \pi_\theta(a|s) \sum_{s'} P(s'|s,a)\nabla_\theta V^\pi(s')\Big)$$

This equation is recursive. Le's denote $f(s) = \sum_{a\in\mathcal{A}} \nabla_\theta\pi_\theta(a|s)Q^\pi(s,a)$ and unroll the recursive representation and extend it infinitely in the the following visitation sequence:

$$s \xrightarrow{a\sim\pi_\theta(.|s)} s' \xrightarrow{a\sim\pi_\theta(.|s')} s'' \xrightarrow{a\sim\pi_\theta(.|s'')} \dots$$

$$\begin{aligned}
\nabla_\theta V^\pi(s) =& f(s) + \sum_a \pi_\theta(a|s) \sum_{s'} P(s'|s,a) \nabla_\theta V^\pi(s') \\
=& f(s) + \sum_{s'} \sum_a \pi_\theta(a|s) P(s'|s,a) \nabla_\theta V^\pi(s') \\
=& f(s) + \sum_{s'} \rho_\pi(s \to x)[f(s') + \sum_{s''} \rho_\pi(s \to s'') \nabla_\theta V^\pi(s'')] \\
=& \ldots \text{ Repeatedly unrolling the part of } \nabla_\theta V^\pi(.) \\
=& \sum_{x \in \mathcal{S}} \sum_{k=0}^{\infty} \rho_\pi(s \to x)^k f(x)
\end{aligned}$$

Where $\rho_\pi(s \to s') = \sum_a \pi_\theta(a|s) P(s'; |s, a)$. Let $\Lambda(s) = \sum_{k=0}^{\infty} \rho_\pi(s \to x)^k$

$$\begin{aligned}
\nabla_\theta J(\theta) = \nabla_\theta V^\pi(s_0) &= \sum_s \Lambda(s) \phi(s) && ; \text{Starting from a random state } s_0 \\
&= \Big( \sum_s \Lambda(s) \Big) \sum_s \frac{\Lambda(s)}{\sum_s \Lambda(s)} \phi(s) \\
&= \sum_s d^\pi(s) \sum_a \nabla_\theta \pi_\theta(a|s) Q^\pi(s,a) && d^\pi(s) = \tfrac{\Lambda(s)}{\sum_s \Lambda(s)} \text{ is stationary distribution.} \\
&= \sum_{s \in \mathcal{S}} d^\pi(s) \sum_{a \in \mathcal{A}} \pi_\theta(a|s) Q^\pi(s,a) \frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} \\
&= \mathbb{E}_\pi[Q^\pi(s,a) \nabla_\theta \ln \pi_\theta(a|s)] && ; \text{Because } (\ln x)' = 1/x
\end{aligned}$$

Where $\mathbb{E}_\pi$ refers to $\mathbb{E}_{s \sim d_\pi, a \sim \pi_\theta}$ when both state and action distributions follow the policy $\pi_\theta$ (on policy).

**Remark** : We Still can prove this theorem in continuous state and action spaces, by using the integration in lieu of summation.

## 3.3 Experiment Details and Hyperparameters

Below, we provide the different hyper-parameters during our experiments(Dynamic setting in brackets when different).

### 3.3.1 DDPG variant Algorithm for Volatility fitting

Table 3.1 lists the DDPG parameters used in our experiments.

Table 3.1: DDPG Hyperparameters

| Parameter | Value |
|---|---|
| optimizer | Adam |
| learning rate Actor and Critic | 0.0025 ($2.5 \cdot 10^{-5}, 2.5 \cdot 10^{-4}$) |
| discount ($\gamma$) | 0.99 |
| replay buffer size | $10^3$ ($2.10^3$) |
| number of hidden layers (all networks) | 2 |
| number of hidden units per layer | 256 |
| number of samples per minibatch (batch size) | 64 (252) |
| Power decaying noise with std bounded in | $[0.01, 0.15]$ |
| nonlinearity | ReLU, Tanh |
| target smoothing coefficient ($\tau$) | 0.001 |
| gradient steps | 1 |
| Neural Network initialisation | Xavier initialisation |

### 3.3.2 SAC variant Algorithm for Volatility fitting

Table 3.2 lists the SAC parameters used in our experiments.

Table 3.2: SAC Hyperparameters

| Parameter | Value |
|---|---|
| optimizer | Adam |
| learning rate Actor and Critic | $2.5 \cdot 10^{-5}, 2.5 \cdot 10^{-4}$ |
| discount ($\gamma$) | 0.99 |
| replay buffer size | $10^3$ |
| number of hidden layers (all networks) | 2 |
| number of hidden units per layer | 256 |
| number of samples per minibatch (batch size) | 64 |
| entropy target | $-\dim(\mathcal{A}) = -K$ |
| automatic entropy tuning | True |
| nonlinearity | ReLU, Tanh |
| target smoothing coefficient ($\tau$) | 0.001 |
| Neural Network initialisation | Xavier initialisation |

## 3.4 Hardware and Computational Ressources

All experiments in this paper were conducted on a machine with the following specifications: An Intel(R) Xeon(R) W-1270 CPU running at 3.40GHz with 64GB of RAM and 64-bit operating system in Windows 11. All algorithms were implemented in Python 3.8.8 using PyTorch 2.3.0+cpu.

## 3.5 Supplementary Results

The below figures display the evolution of some important quantities during the training of the RL agent in several market type and MSE reward: A Monte-Carlo on 5 differents random seeds is performed with a power decaying noise for **DDPG** and automatic entropy adjustment for **SAC**. We represent in (*green*) the best response of the agent amongst the last 1000 steps and in (*blue*) the mean of the final volatility slices for the last 1000 steps.

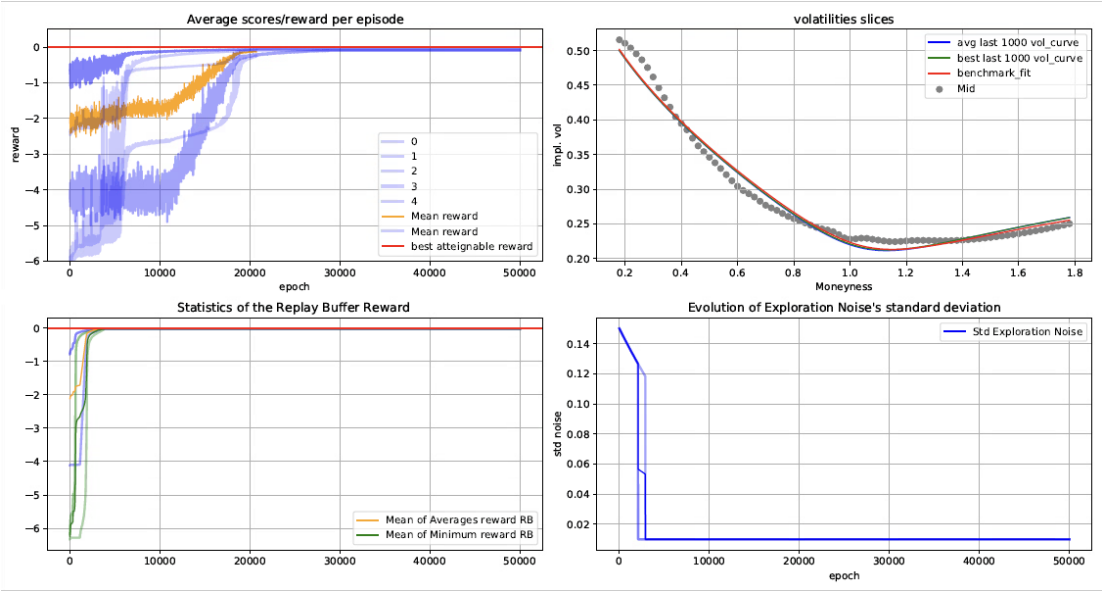



Figure 3.1: A snapshot of the training in a "Skew" configuration with **DDPG**.

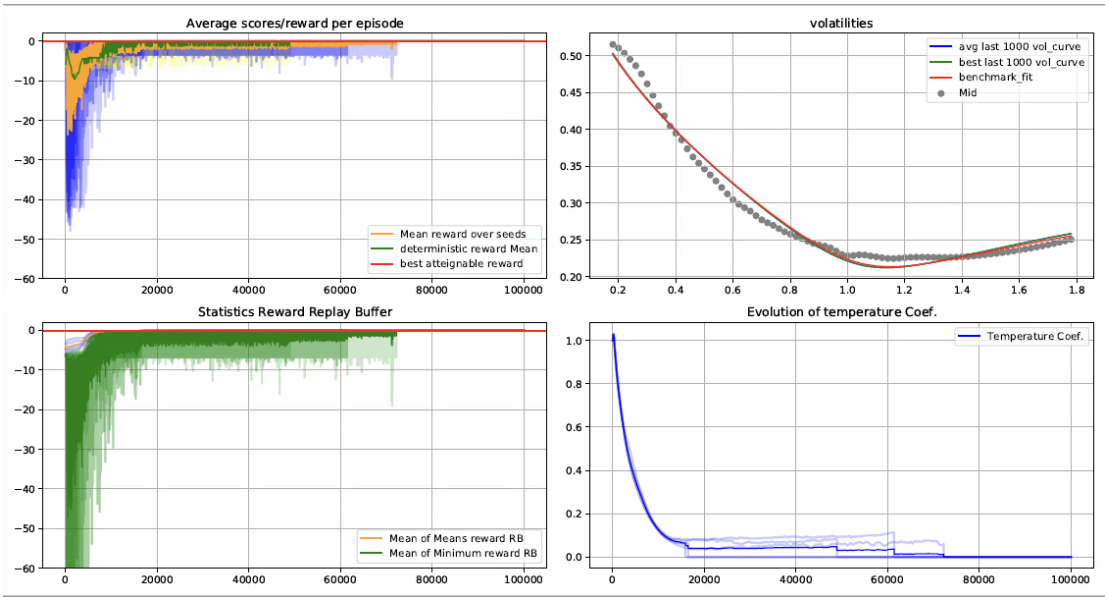



Figure 3.2: A snapshot of the training in a "Skew" configuration with **SAC**.

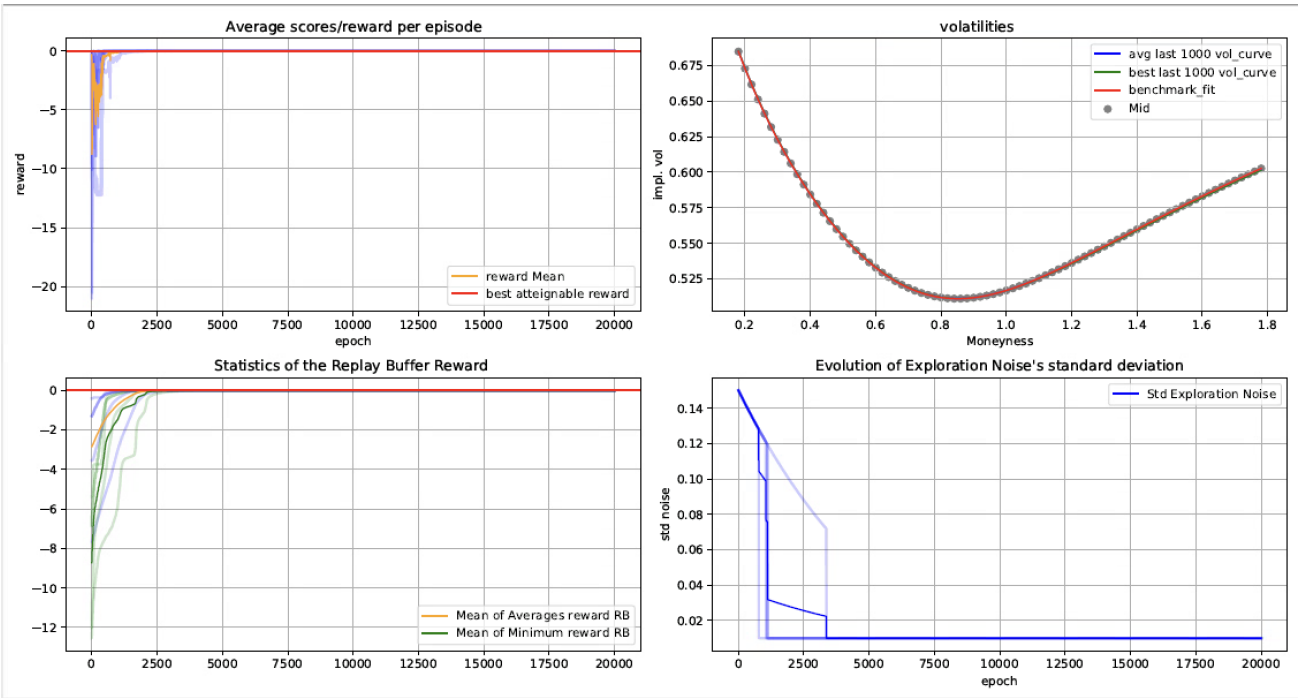



Figure 3.3: A snapshot of the training in a "High Smile" configuration with **DDPG**.

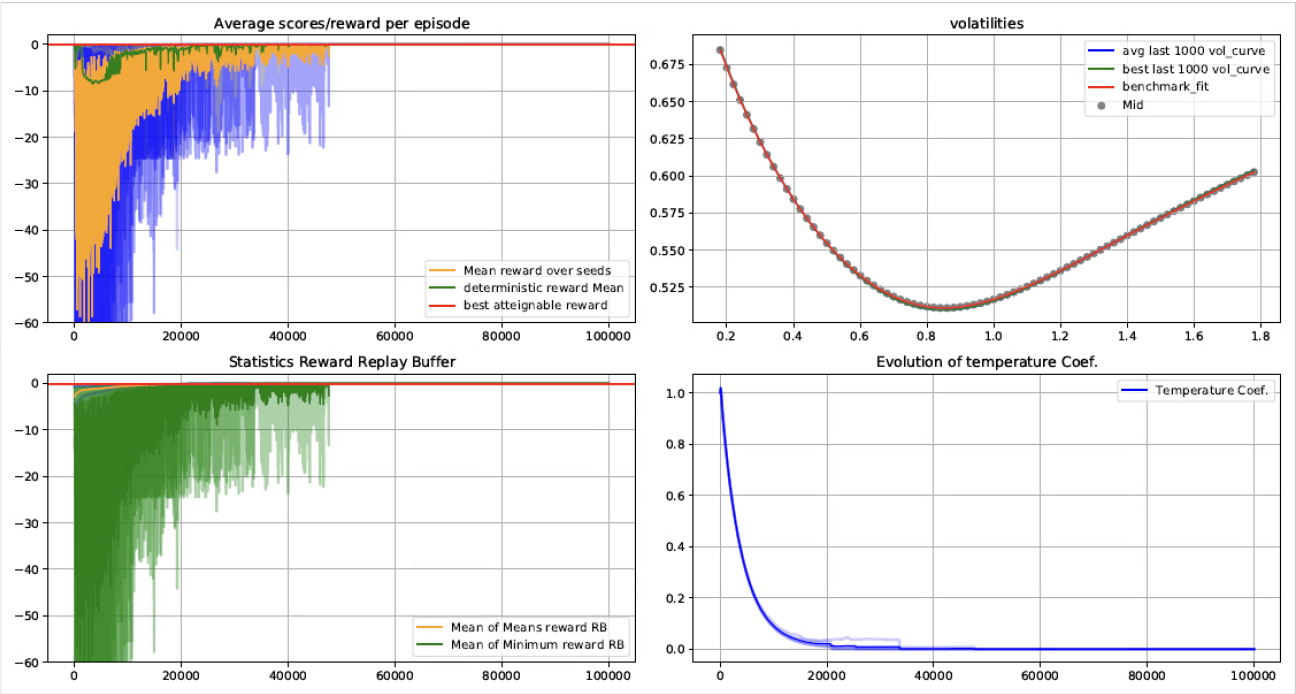



Figure 3.4: A snapshot of the training in a "High Smile" configuration with **SAC**.

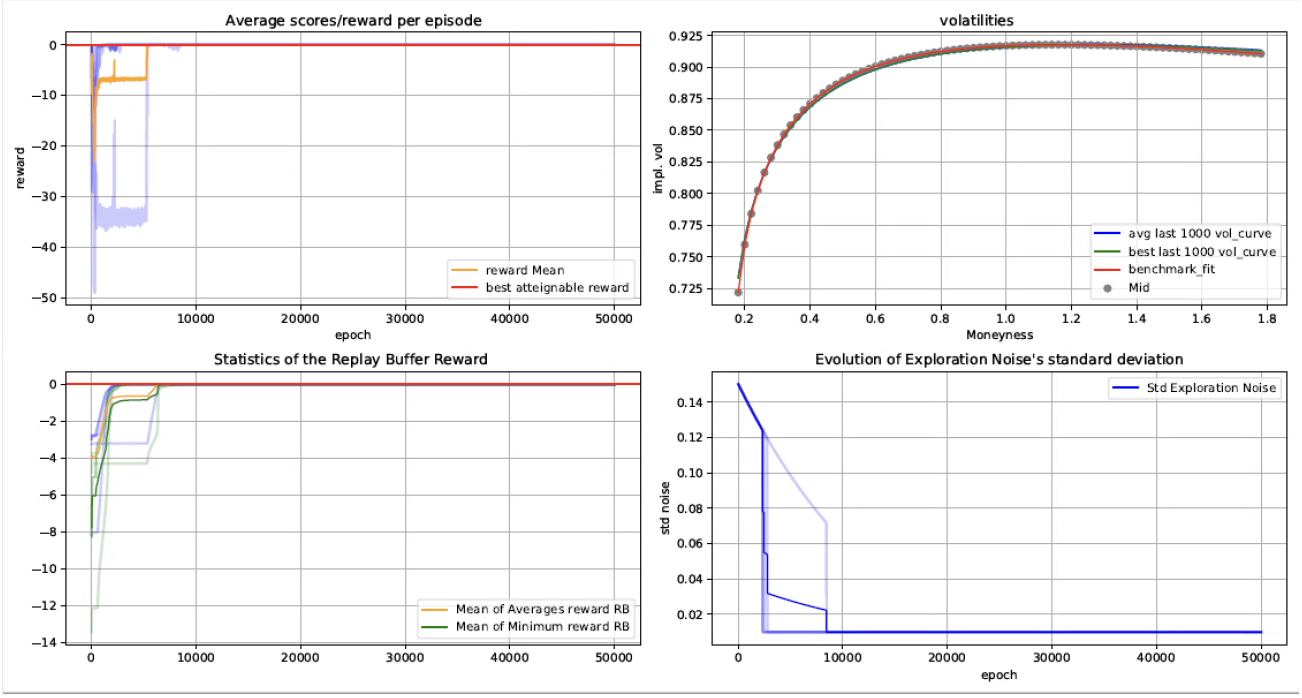



Figure 3.5: A snapshot of the training in a "Inverse Smile" configuration with **DDPG**.

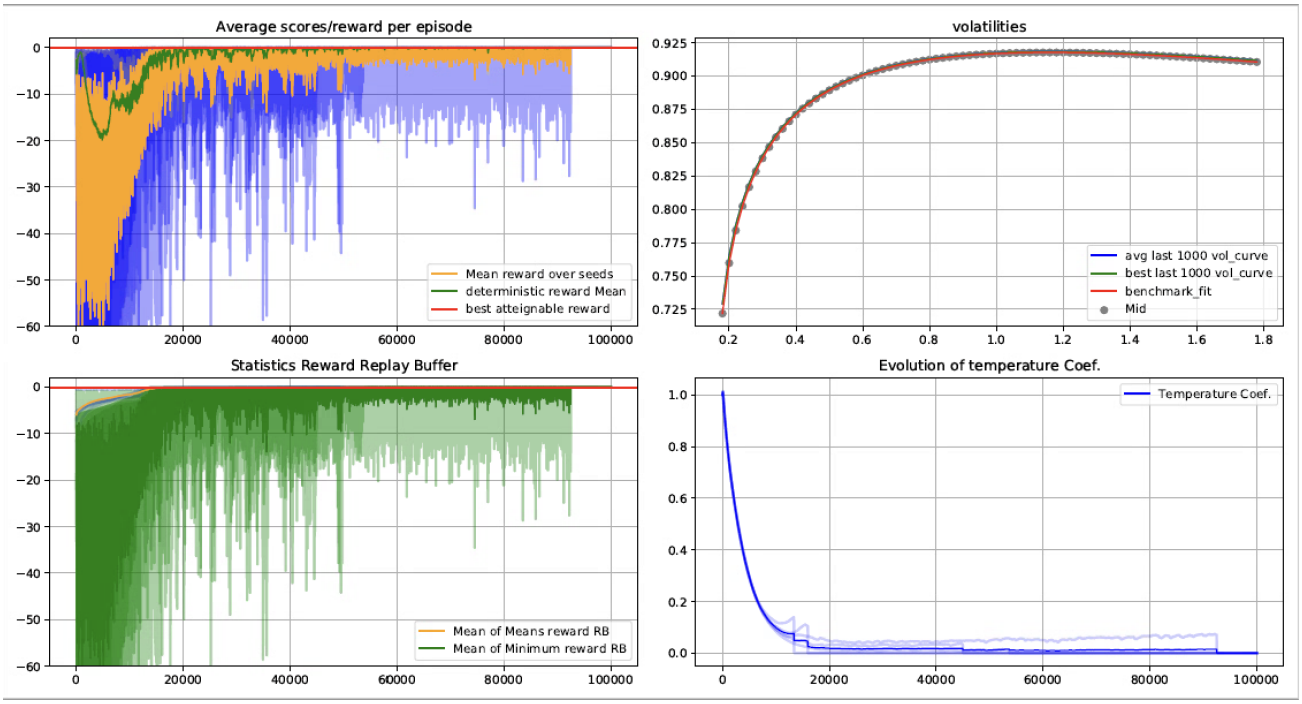



Figure 3.6: A snapshot of the training in a "Inverse Smile" configuration with **SAC**.

# Bibliography


[1] B. Hambly, R. Xu, and H. Yang, *Recent Advances in Reinforcement Learning in Finance*, arXiv:2112.04553, 2021.

[2] H. Robbins, and S. Monro, *A stochastic approximation method.*, Ann. Math. Statistics, 22:400–407, 1951.

[3] Bertsekas, D. P. *Dynamic programming and optimal control*, volume 1. Athena scientific Belmont, MA, 2005.

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

[5] A. Charpentier, R. Elie, and C. Remlinger, *Reinforcement learning in economics and finance*, Computational Economics, (2021), pp. 1–38.

[6] P. N. Kolm, and G. Ritter, *Modern perspectives on reinforcement learning in finance*, Modern Perspectives on Reinforcement Learning in Finance (September 6, 2019). The Journal of Machine Learning in Finance, 1 (2020).

[7] P. Murray, B. Wood, H. Buehler, M. Wiese, and M. S. Pakkanen, *Deep Hedging: Continuous Reinforcement Learning for Hedging of General Portfolios across Multiple Risk Aversions*, arXiv:2207.07467v1, 2022.

[8] M. F. Dixon, I. Halperin, and P. Bilokon, *Machine Learning in Finance*, Springer, 2020.

[9] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, *Policy gradient methods for reinforcement learning with function approximation*, in Advances in Neural Information Processing Systems, 2000, pp. 1057–1063.

[10] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, *Continuous control with deep reinforcement learning*, in 4th International Conference on Learning Representations (ICLR), 2016.

[11] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, *Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor*, in International Conference on Machine Learning, PMLR, 2018, pp. 1861–1870.

[12] V. R. Konda and J. N. Tsitsiklis, *Actor-critic algorithms*, in Advances in Neural Information Processing Systems, 2000, pp. 1008–1014.

[13] T. Haarnoja, A. Zhou,K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, and S. Levine, *Soft Actor-Critic Algorithms and Applications*, in International Conference on Machine Learning, PMLR, 2019.

[14] J. D. Abernethy and S. Kale, *Adaptive market making via online learning*, in NIPS, Citeseer, 2013, pp. 2058–2066.

[15] C. J. Watkins and P. Dayan, *Q-learning*, Machine learning, 8 (1992), pp. 279–292.

[16] T. Spooner, J. Fearnley, R. Savani, and A. Koukorinis, *Market making via reinforcement learning*, in International Foundation for Autonomous Agents and Multiagent Systems, AAMAS '18, 2018, pp. 434–442.

[17] M. Zhao and V. Linetsky, *High frequency automated market making algorithms with adverse selection risk control via reinforcement learning*, in Proceedings of the Second ACM International Conference on AI in Finance, 2021, pp. 1–9.

[18] R. J. Williams, *Simple statistical gradient-following algorithms for connectionist reinforcement learning*, Machine Learning, 8 (1992), pp. 229–256.

[19] S. Ganesh, N. Vadori, M. Xu, H. Zheng, P. Reddy, and M. Veloso, *Reinforcement learning for market making in a multi-agent dealer market*, arXiv preprint arXiv:1911.05892, (2019).

[20] T. Spooner and R. Savani, *Robust Market Making via Adversarial Reinforcement Learning*, in Proc. of the 29th International Joint Conference on Artificial Intelligence, IJCAI-20, 7 2020, pp. 4590–4596.

[21] Z. Ye, W. Deng, S. Zhou, Y. Xu, and J. Guan, *Optimal trade execution based on deep deterministic policy gradient*, in Database Systems for Advanced Applications, Springer International Publishing, 2020, pp. 638–654.

[22] S. Lin and P. A. Beling, *An end-to-end optimal trade execution framework based on proximal policy optimization*, in IJCAI, 2020, pp. 4548–4554.

[23] J. Cao, J. Chen, J. Hull, and Z. Poulos, *Deep hedging of derivatives using reinforcement learning*, The Journal of Financial Data Science, 3 (2021), pp. 10–27.

[24] S. Ioffe and C. Szegedy, *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, arXiv:1502.03167, (2015).

[25] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al., *Human-level control through deep reinforcement learning*, Nature, 518 (2015), pp. 529–533.

[26] J. Gatheral, A. Jacquier., *Arbitrage-Free SVI Volatility Surfaces*, Quantitative Finance, Vol. 14, No. 1, 59-71, 2014.

[27] V. Zetocha., *Sculpting implied volatility surfaces of illiquid assets*, (April 11, 2022).

[28] J. Gatheral., *A parsimonious arbitrage-free implied volatility parameterization with application to the valuation of volatility derivatives.*, (Presentation at Global Derivatives and Risk Management, Madrid, 2004).

[29] G. Guo, A. Jacquier, C. Martini, and L. Neufcourt G. Cybenko., *Generalised arbitrage-free SVI volatility surfaces.*, Computational Finance, (May 27, 2016).

[30] C. Homescu., *Implied volatility surface: construction methodologies and characteristics.*, (arXiv, July 9, 2011).

[31] G. Cybenko., *Approximation by superpositions of a sigmoidal function.*, Math. Control Signals Systems, 2(4):303–314, 1989.

[32] Y. LeCun, Y. Bengio and G. Hinton., *Deep learning.*, Nature, 2015.

[33] D. Knowles., *Lagrangian Duality for Dummies.*, Stanford University, 2010.

[34] K. Hornik, M. Stinchcombe and H. White., *Multilayer feedforward networks are universal approximators.*, Neural Networks, Vol. 2, 359-366.