A SAMPLE EFFICIENT OFF-POLICY ACTOR-CRITIC APPROACH FOR DISCRETE ACTION ENVIRONMENTS

Nicholas Clavette
A SAMPLE EFFICIENT OFF-POLICY ACTOR-CRITIC APPROACH FOR
DISCRETE ACTION ENVIRONMENTS

BY

NICHOLAS CLAVETTE

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
ELECTRICAL ENGINEERING

UNIVERSITY OF RHODE ISLAND
2023
MASTER OF SCIENCE THESIS
OF
NICHOLAS CLAVETTE

APPROVED:

Thesis Committee:

Major Professor    Haibo He
                    Yan Sun
                    Marco Alvarez
                    Brenton DeBoef

DEAN OF THE GRADUATE SCHOOL

UNIVERSITY OF RHODE ISLAND
2023
ABSTRACT

In the current reinforcement learning (RL) taxonomy, there exist few algorithms for discrete action environments that are capable of learning stochastic policies in an off-policy manner. Learning stochastic policies brings benefits such as stable training and smoother exploration strategies. Training an algorithm in an off-policy manner allows for greater sample efficiency, as experiences collected while interacting with a learning environment can be used more than once. Stable performance and good sample efficiency are highly important when collecting experiences from a learning environment is expensive. This thesis proposes a new algorithm for discrete action RL called Discrete General Policy Optimization (Discrete GPO) that has both of the above characteristics. The algorithm is designed following recent theoretical developments in trust region policy optimization techniques. The performance of Discrete GPO is tested in different simulated learning environments, and a comparison to other state of the art methods is provided.
ACKNOWLEDGMENTS

I would like to express my sincerest gratitude to my primary advisor and mentor Dr. Haibo He for providing me with this wonderful opportunity to learn about and contribute to the rich and exciting field of machine learning and artificial intelligence. I would also like to thank my colleague, Dr. Hepeng Li, for working with me so patiently and sharing his deep knowledge of the field. Thanks to the both of you, I have been exposed to and conquered many challenges that would have previously seemed insurmountable. The skills and mindset learned throughout earning this degree have undoubtedly changed me for the better, and I will cherish them for the rest of my life. I would also like to thank my family for their support throughout my studies, and my friends who made this journey so memorable. During my M.S. study, my research has been partially supported by the National Science Foundation (NSF) under ECCS 1731672, and Office of Naval Research (ONR) under N00014-18-1-2396 and N00014-20-C-1096. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the funding agencies. I gratefully acknowledge the support from NSF and ONR for my research.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>ii</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>iii</td>
</tr>
<tr>
<td>TABLE OF CONTENTS</td>
<td>iv</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>vi</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>vii</td>
</tr>
<tr>
<td>CHAPTER</td>
<td></td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Research Motivation and Problem Statement</td>
<td>1</td>
</tr>
<tr>
<td>1.2 A Stochastic Off-Policy Algorithm for Discrete Action RL</td>
<td>4</td>
</tr>
<tr>
<td>1.3 Improved Environment Exploration with Maximum Entropy RL</td>
<td>5</td>
</tr>
<tr>
<td>1.4 An Outline of the Thesis</td>
<td>6</td>
</tr>
<tr>
<td>2 Reinforcement Learning Background</td>
<td>8</td>
</tr>
<tr>
<td>2.1 Problem Formulation</td>
<td>8</td>
</tr>
<tr>
<td>2.2 Reinforcement Learning Algorithms</td>
<td>11</td>
</tr>
<tr>
<td>2.2.1 Off-Policy Learning with Value-Based Methods</td>
<td>11</td>
</tr>
<tr>
<td>2.2.2 On-Policy Learning with Policy Gradient Methods</td>
<td>14</td>
</tr>
<tr>
<td>2.3 Deep Reinforcement Learning</td>
<td>16</td>
</tr>
<tr>
<td>3 Trust Region Optimization in Reinforcement Learning</td>
<td>18</td>
</tr>
<tr>
<td>3.1 Trust Region Methods</td>
<td>18</td>
</tr>
<tr>
<td>3.2 Trust Region Policy Optimization</td>
<td>19</td>
</tr>
</tbody>
</table>
3.3 General Policy Optimization ............................... 21

4 Discrete GPO: A Stochastic Off-Policy Actor-Critic Algorithm 25
  4.1 Parameterized Policy Optimization via Target Policy Minimization 25
  4.2 Learning the Critic Network ............................... 28

5 Experiments and Simulations ................................. 32
  5.1 A Case Study in CartPole and LunarLander Environments ... 34
    5.1.1 Environment Descriptions ............................ 34
    5.1.2 Experiment Configuration ............................. 35
    5.1.3 Results and Findings ................................. 36
  5.2 Playing Games from Images ............................... 40
    5.2.1 Environment Pre-Processing .......................... 40
    5.2.2 Shared Convolutional Neural Network Encoder ........ 42
    5.2.3 Breakout Performance Comparison .................... 44

6 Conclusion and Future Work ................................. 48

LIST OF REFERENCES ........................................... 50

BIBLIOGRAPHY ..................................................... 54
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The agent-environment interaction [1]</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>Gradient ascent on parameterized policy performance</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>Graphical example of true return and surrogate return</td>
<td>21</td>
</tr>
<tr>
<td>4</td>
<td>Graphical representation of the update rule given in Eq. 29</td>
<td>24</td>
</tr>
<tr>
<td>5</td>
<td>An example of a fully connected policy network</td>
<td>26</td>
</tr>
<tr>
<td>6</td>
<td>A graphical representation of the forward passes (solid arrows) and gradient flow (dashed arrows) through the actor and critic networks for a single mini-batch update.</td>
<td>30</td>
</tr>
<tr>
<td>7</td>
<td>Renderings of CartPole (top-left), LunarLander (top-right), and Breakout (bottom-center) Environments.</td>
<td>32</td>
</tr>
<tr>
<td>8</td>
<td>Average episodic return of Discrete GPO, PPO, DQN on CartPole-v1 evaluation environment.</td>
<td>37</td>
</tr>
<tr>
<td>9</td>
<td>Average episodic return of Discrete GPO, PPO, DQN on LunarLander-v2 evaluation environment.</td>
<td>38</td>
</tr>
<tr>
<td>10</td>
<td>Shared CNN feature encoder architecture.</td>
<td>43</td>
</tr>
<tr>
<td>11</td>
<td>Average episodic return of Discrete GPO, PPO, DQN on BreakoutNoFrameskip-v4 evaluation environment.</td>
<td>44</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Discrete GPO, DQN, and PPO hyperparameter table for experiments in CartPole-v1 and LunarLander-v2 Environments.</td>
</tr>
<tr>
<td>2</td>
<td>Discrete GPO, DQN, and PPO hyperparameter table for experiments in BreakoutNoFrameskip-v4 environment.</td>
</tr>
<tr>
<td>3</td>
<td>List of hyperparameters and their description.</td>
</tr>
</tbody>
</table>
CHAPTER 1

Introduction

1.1 Research Motivation and Problem Statement

Reinforcement learning [1] is an important type of learning that occurs in biological life all around us, including ourselves. It is the learning paradigm that increases the likelihood of behaviors which have led to highly rewarding outcomes, and decreases the likelihood of behaviors that have resulted in negative outcomes. For example, as a child we learn to avoid touching hot surfaces after experiencing that it causes us pain, and learn to behave politely so that we receive more positive affirmation from others around us. Through trial and error, we learn how our environment responds to our actions, and seek to influence what happens by modifying our behavior. This biological learning phenomena has inspired many approaches to computational intelligence, ultimately leading to the rich research field of computational reinforcement learning, which is the primary topic of this thesis.

In the context of machine learning (ML), reinforcement learning differs from supervised learning and unsupervised learning methods. Training a model using supervised learning requires large data-sets which must be labeled by a subject matter expert (SME) [2]. Such data-sets can be difficult and time consuming to create. The training data-sets will consist of data-label pairs \((x, y)\) where labels \(y\) act as a “supervisor,” instructing the model to output label \(y\) when predicting on input data \(x\). Supervised learning methods are typically used in applications such as image classification, object recognition, and regression, and can be incredibly powerful especially when training large deep neural networks with gradient descent techniques [3, 4]. Unsupervised learning does not require SME labeled data. Instead, unsupervised learning techniques seek to learn abstract representations of
a training data-set which can be used for upstream tasks such as recommendation systems or anomaly detection [5]. On the other hand, reinforcement learning methods do not require any pre-compiled training data-sets. RL learns behavior strategies from scratch (assuming no prior knowledge) through trial and error within a learning environment.

In the ultimate search for artificial general intelligence (AGI), learning from scratch via trial and error as done in RL has advantages over learning from large labeled data-sets. Firstly, RL’s approach to learning is much more general. Algorithms are designed to learn optimal policies within any environment that can be formulated as a proper Markov Decision Process (MDP). The same algorithm can be applied to learn intelligent behavior in a plethora of environments. Secondly, learning is not limited by the capacity of the subject matter expert themselves. For example, the performance of a game playing program trained via supervised learning on a human created rule set is likely to be surpassed by a program trained with reinforcement learning, as RL has the ability to find more optimal strategies that were never considered or simply not imposed in the SME rule set.

An example that demonstrates the advantage of RL’s ability to learn from scratch and in a diverse set of problems is AlphaZero [6], an algorithm developed by Google DeepMind in 2017. AlphaZero was designed to be a more general and flexible version of the earlier AlphaGo algorithm, which was specifically developed to play the game of Go. AlphaZero uses a combination of deep neural networks and an RL algorithm enhanced with Monte Carlo Tree Search (MCTS) to make decisions in a game. It can learn to play chess, Go, and shogi with superhuman performance completely from scratch with no expert knowledge and without any alterations to the algorithm in the different game domains. In contrast, Stockfish [7], the top performing chess program at the time used a combination of handcrafted
SME features with carefully tuned weights and domain specific alpha-beta tree search techniques [8]. It is RL’s approach to learning from scratch in general settings that makes many believe reinforcement learning will be a great contributor to future general AI systems and artificial general intelligence as witnessed by the recent breakthrough of ChatGPT, which uses reinforcement learning from human feedback (RLHF) [9, 10] to help align model output with human preferences.

Most of RL’s real world success today comes from the combination of fundamental RL theory and powerful non-linear function approximators such as deep neural networks. In deep RL, deep neural networks are used to approximate important functions in RL theory [11]. Despite deep RL’s successes, there are still challenges that plague its use today.

**Challenge 1:** Deep Reinforcement learning can be quite sample inefficient. The sample efficiency of an algorithm refers to the amount of data or experience required to learn an effective policy. In some cases, it can take millions of interactions with an environment to learn a high-performing policy, which can be problematic in real-world situations where collecting experience may be difficult or costly.

**Challenge 2:** Deep reinforcement learning algorithms (and RL algorithms in general) often face a trade-off between exploration and exploitation. Exploration refers to the process of actively seeking out new information and trying out different actions in the environment in order to potentially discover more rewarding paths. Exploitation refers to the act of utilizing the current best-known action or strategy in order to maximize reward. The exploration-exploitation dilemma arises when an RL algorithm must decide whether to continue exploring in search of potentially higher rewards, or to stick with the current best-known action that maximizes reward. Finding the right
balance between exploration and exploitation is a major challenge and is an active area of research in RL.

1.2 A Stochastic Off-Policy Algorithm for Discrete Action RL

In regards to challenge 1, the ability for an algorithm to reuse past experiences to perform policy updates is useful for increasing sample efficiency. Reusing past experiences to perform a policy update is called off-policy learning and algorithms that update policies in an off-policy manner are more sample efficient than their on-policy counterparts. On-policy learning algorithms must perform policy updates using experiences collected while acting according to the most recent policy. In this case, all experiences collected to perform the latest policy update must be tossed out and replaced to compute the next policy update. When sample efficiency is important in solving a problem with RL, it is clear that an off-policy algorithm is the better choice.

Another characteristic of an RL algorithm that can impact sample efficiency and training stability is whether the algorithm trains a deterministic or a stochastic policy. Algorithms that learn deterministic policies tend to be more unstable, as small changes in policy parameters can lead to very different policy behavior during training [12]. Furthermore, in regards to the exploration and exploitation dilemma, deterministic policies often require external exploration noise to be added to aid exploration early in training [13]. In contrast, algorithms which train stochastic policies learn a probability distribution over the action space from which actions are sampled from. In this configuration, small updates to policy parameters will likely only cause small changes in the probability distribution, resulting in small changes in the actions sampled and smoother training.

However, in discrete action RL domains, there exist few algorithms that bring the greater sample efficiency of off-policy learning, and even fewer that train a
stochastic policy in an off-policy manner. For example, strong off-policy reinforcement learning algorithms include Deep Deterministic Policy Gradient (DDPG) [13], Soft Actor-Critic (SAC) [14], Twin Delayed DDPG (TD3) [15], and Q-learning (DQN) [11]. Of these algorithms, DDPG and TD3 only work in continuous action environments, while SAC works in both continuous and discrete action environments but greatly underperforms in the latter [16, 17]. Thus, the current top performing off-policy algorithm for discrete action environments is Q-learning and its variants which train deterministic policies, bringing their own disadvantages.

The primary goal of this thesis is to provide a new sample efficient algorithm to the discrete action RL family that trains a stochastic actor in an off-policy manner. The algorithm is implemented following the theory derived in recent work titled “An Analytical Update Rule for General Policy Optimization” (Li, Clavette, He) [18] which builds upon trust region policy optimization methods [19] to provide an analytical policy update rule that has a monotonic improvement guarantee on policy performance.

1.3 Improved Environment Exploration with Maximum Entropy RL

In regards to challenge 2, there are various approaches to encouraging exploration during training. Value based methods like Q-learning indirectly learn a deterministic policy and often force exploration by utilizing what is known as an “epsilon-greedy” exploration strategy. A hyperparameter $\epsilon \in [0, 1]$ is defined at the start of training, and the RL agent behaves randomly with probability $\epsilon$. Generally, epsilon is chosen to be a high probability at the start of training, such that an agent will explore and collect a diverse set of experiences from which a well performing greedy policy can be found. As training continues, epsilon is decayed to some small number close to zero, so the agent transitions to behavior following a exploitative greedy policy.
Algorithms which directly learn a stochastic policy provide a more natural approach to this problem. First, the policy can be initialized such that there is uniform probability of selecting any action in any state. As the policy improves, the policy may become more concentrated on taking specific highly rewarding actions - but in a smoother fashion. Second, learning a stochastic policy allows for the addition of a maximum entropy objective [14, 20]. The entropy in the policy is a measure from information theory which describes the ‘randomness’ of a given distribution. In maximum entropy RL, an agent’s policy is optimized to receive the highest return while also retaining the highest entropy. Stochastic policies that are trained while trying to maintain a high level of entropy can still learn to prefer actions with highest value, while maintaining some degree of exploration. The proposed algorithm in this thesis incorporates a maximum entropy term into the primary objective for policy optimization to assist with effective exploration of an environment during training.

1.4 An Outline of the Thesis

This thesis consists of six chapters. The first chapter provides some basic knowledge on different types of reinforcement learning algorithms as well as their advantages and disadvantages. It provides motivation as to why a stochastic off-policy algorithm is needed in the discrete action case, and explains the exploration vs. exploitation dilemma that reinforcement learning algorithms face. The second chapter provides a more rigorous background on fundamental reinforcement learning theory and an introduction to value-based learning methods and policy gradient methods and their differences. Chapter two also introduces deep reinforcement learning and some of its challenges. Chapter three introduces trust region optimization, and demonstrates how it has be used in RL theory to derive techniques for optimizing policies with a monotonic improvement guarantee on pol-
icy performance. Chapter three also introduces the recent work by (Li, Clavette, He) [18] which the proposed algorithm is inspired by. Chapter four describes the development process of the proposed algorithm itself. It provides instruction on how the theory from [18] can be adapted to learn a stochastic policy that is represented by a deep neural network for use in large scale complex real world RL problems. Chapter five provides the results of using the proposed algorithm in different RL environments. The algorithm is compared to other state of the art RL methods. Chapter six summarizes the thesis as a whole, and provides a few ideas for future pathways to improve this work.
CHAPTER 2

Reinforcement Learning Background

2.1 Problem Formulation

Reinforcement learning problems are formulated as a Markov Decision Process (MDP) [1]. A MDP is characterized by a state space \( S \), action space \( A \), state transition probability density \( p : S \times A \times S \to \mathbb{R} \), a reward function \( r : S \times A \to \mathbb{R} \), initial state probability density \( \rho_0 : S \to \mathbb{R} \), and discount factor \( \gamma \in [0, 1) \).

A reinforcement learning ‘agent’ interacts with an environment through a sequence of discrete time steps, \( t = 0, 1, 2, 3, \ldots \). At each time step, the agent selects an action \( a_t \) from the current policy (\( a_t = \pi_k(s_t) \) if deterministic, \( a_t \sim \pi_k(\cdot|s_t) \) if stochastic) conditioned on some representation of the environment’s state \( s_t \). The environment’s state transitions from \( s_t \) as a result of the agent’s choice to \( s_{t+1} \sim p(\cdot|s_t, a_t) \) and the agent receives a numerical reward, \( r_t = r(s_t, a_t) \). See Fig. 1 for a visual representation of the agent-environment interaction in an MDP.

![Figure 1. The agent-environment interaction [1].](image)

This agent-environment interaction induces a sequence, called a trajectory \( \tau \).
that is of the following form:

$$\tau = s_0, a_0, r_0, s_1, a_1, r_1, s_2, a_2, r_2, s_3, \cdots, s_{T+1}$$  \hspace{1cm} (1)$$

The total return of a trajectory is written as the sum of discounted rewards:

$$R(\tau) = \sum_{t=0}^{T} \gamma^t r_t$$  \hspace{1cm} (2)$$

where $\gamma \in [0, 1)$ is a ‘discount’ factor to ensure the return is always bounded in the event that $T$ is very close or equal to $\infty$, and as a measure to control how much the agent should care about distant rewards.

Useful functions used in almost all reinforcement learning algorithms include the state-value function $V_\pi(s)$, the action-value function $Q_\pi(s, a)$, and the advantage function $A_\pi(s, a)$. The **state-value** function computes the expected return an agent may incur if it starts in some state $s$, and acts according to a policy $\pi$ forever after or until a terminal state is reached:

$$V_\pi(s) = \mathbb{E}_{\tau \sim \pi} [R(\tau)|s_0 = s]$$  \hspace{1cm} (3)$$

where $\tau \sim \pi$ indicates that the probability of the trajectory is dependent on the policy: $s_0 \sim \rho_0, a_t \sim \pi(\cdot|s_t), s_{t+1} \sim p(\cdot|s_t, a_t)$. Intuitively, the value function provides a measure of how good it is to be in a given state, if all possible future trajectories occur as an agent makes decisions following a policy $\pi$.

The **action-value** function computes the expected return an agent will incur if it starts in some state $s$, takes some action $a$, and then acts according to a policy $\pi$ forever after or until a terminal state is reached:

$$Q_\pi(s, a) = \mathbb{E}_{\tau \sim \pi} [R(\tau)|s_0 = s, a_0 = a]$$  \hspace{1cm} (4)$$

The key difference between $Q_\pi(s, a)$ and $V_\pi(s)$ is the dependence on a specific action which can be chosen arbitrarily and does not have to come from a policy.
In many reinforcement learning algorithms [11, 14, 21, 13, 22], the action-value function is used as a measure to know which action has the highest value - which can be used to improve the policy.

The **advantage** function is the difference between the action-value function and the state-value function at some state-action pair \((s, a)\):

\[
A_\pi(s, a) = Q_\pi(s, a) - V_\pi(s) \tag{5}
\]

It is used to provide a measure for how much better it is to choose an action \(a \in A\) in a given state \(s \in S\) compared to acting ‘normal’ (selecting the action from the current policy \(\pi\)). An action with a positive advantage will lead to a greater return than an action with zero or negative advantage. Similarly to the action-value function \(Q_\pi\), the advantage function is frequently used in policy optimization methods [23] as a measure to determine whether the probability of an action should be increased or decreased when updating the policy.

In each MDP, there exists at least one optimal policy, \(\pi^*\), who’s state-value function and action-value function is also optimal:

\[
V^*(s) = \max_\pi V_\pi(s) \tag{6}
\]

for all \(s \in S\), and

\[
Q^*(s, a) = \max_\pi Q_\pi(s, a) \tag{7}
\]

for all \(s \in S, a \in A\).

The primary goal of reinforcement learning algorithms is to learn a policy that achieves (hopefully global) optimal value in all states to maximize the expected return while interacting and generating trajectories within an environment.
2.2 Reinforcement Learning Algorithms
2.2.1 Off-Policy Learning with Value-Based Methods

Value-based reinforcement learning algorithms seek to learn the optimal action-value function Eq. 7 from which an optimal policy is implicitly extracted. They exploit an important recursive property of the state and action value functions introduced above. Note that the on policy value functions can be written recursively as:

\[ V_\pi(s) = \mathbb{E}_{a \sim \pi, s' \sim p}[r(s,a) + \gamma V_\pi(s')] \] (8)

\[ Q_\pi(s,a) = \mathbb{E}_{s' \sim p}[r(s,a) + \gamma \mathbb{E}_{a' \sim \pi}[Q_\pi(s',a)]] \]

while the optimal value functions can be written recursively as:

\[ V^*(s) = \max_a \mathbb{E}_{s' \sim p}[r(s,a) + \gamma V^*(s')] \]

\[ Q^*(s,a) = \mathbb{E}_{s' \sim p}\left[r(s,a) + \gamma \max_{a'} Q^*(s',a')\right] \] (9)

The above equations are known as Bellman equations, and their intuition is as follows: The value of being in a state is the immediate reward expected for being in that state, plus the value of wherever you end up next.

Bellman equations play a very important role in evaluating (learning) \( V_\pi \) and \( Q_\pi \) for some policy \( \pi \). They can be used as an iterative update rule for to provide more accurate estimates of value functions in dynamic programming (DP) and temporal difference (TD) learning techniques. Consider a sequence of value function estimates \( V_0, V_1, V_2, \ldots \), where \( V_0 \) is initialized arbitrarily everywhere except for terminal states where it is equal to 0. In the dynamic programming framework, where the environment dynamics are assumed to be completely known, each successive estimate can be obtained using the Bellman equation for \( V_\pi \) by computing:

\[ V_{k+1}(s) = \sum_a \pi(a|s) \sum_{s'} p(s'|s,a) [r(s,a) + \gamma V_k(s')] \] (10)
for all $s \in \mathcal{S}$. The sequence $\{V_k\}$ can be shown to converge to $V_\pi$ as $k \to \infty$.

The motive behind evaluating the value function of a policy is so that it can be used to find a new policy. Consider a deterministic policy $\pi$, for which the value function has been learned. In some state $s$, we wish to know if it would be better to perform an arbitrary action not taken from the current policy, $a \neq \pi(s)$. One way to accomplish this is to perform $a \neq \pi(s)$ in $s$ and then follow the policy $\pi$ forever after. The value of this state-action pair is:

$$Q_\pi(s, a) = \sum_{s'} p(s'|s, a) [r(s, a) + \gamma V_\pi(s')]$$ (11)

If it is found that $Q_\pi(s, a) > V_\pi(s)$, the value of selecting action $a$ is greater than selecting an action following the current policy, then the policy would be better off to select action $a$ each time the state $s$ is encountered, and should be modified to do so. This is called policy improvement, and interested readers can learn more in Chapter 4.2 of the wonderful textbook [1]. In fact, if it is better to behave by selecting the action $a$ in a single state $s$ that has the greatest action-value, then it is also better to behave by selecting the action with the highest value in every state: $\pi'(s) = \arg\max_a Q_\pi(s, a)$. This is known as the ‘greedy policy,’ and it is used in value-based reinforcement learning algorithms (e.g. Q-learning, DQN) [22, 11] to perform the policy improvement step.

By utilizing the above policy evaluation and policy improvement steps in a loop, one can iteratively evaluate and improve a policy until an optimal policy and optimal value function are learned:

$$\pi_0 \xrightarrow{E} V_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} V_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \cdots \xrightarrow{I} \pi_\ast \xrightarrow{E} V^*$$ (12)

In large real world RL problems, it is unlikely that the environment dynamics are completely known. Temporal difference learning remedies this by assuming the environment dynamics are unknown. Instead of having to compute Eq. 10 over the
entire state and action space as in DP methods, temporal difference learning can collect experiences from the environment and immediately make a useful update to an estimate $V_k$:

$$V_{k+1}(s_t) \leftarrow V_k(s_t) + \alpha [r(s_t, a_t) + \gamma V_k(s_{t+1}) - V_k(s_t)]$$

where $a_t \sim \pi_k(\cdot|s_t)$

or similarly, make a useful update an estimate $Q_k$:

$$Q_{k+1}(s_t, a_t) \leftarrow Q_k(s_t, a_t) + \alpha [r(s_t, a_t) + \gamma Q_k(s_{t+1}, a_{t+1}) - Q_k(s_t, a_t)]$$

where $a_{t+1} \sim \pi_k(\cdot|s_{t+1})$

and where $\alpha$ is a step-size parameter that controls how much the estimate should change to reduce the temporal difference error (the second terms in Eq. 13, 14).

The Q-Learning [22] algorithm represents a great example to introduce how a practical TD learning algorithm can be designed to use Eq. 14 to iteratively improve estimates of the Q-function while collecting experiences in an environment. Q-Learning is an off-policy algorithm which directly learns the action-value estimates of the greedy policy while behaving according to the ‘epsilon-greedy policy’. The epsilon-greedy policy is a solution to the exploration-exploitation dilemma which is often used in value-based methods. With probability $\epsilon$, the policy is to behave randomly, and with probability $1 - \epsilon$, the policy is to follow the greedy policy from the latest Q-value estimates. Epsilon is typically initialized to be 1, and then is decayed to some small number as learning continues, so the agent transitions from exploring to exploiting. Q-Learning updates the action-value estimate of any experience generated within an environment by:

$$Q_\pi(s, a) \leftarrow Q_\pi(s, a) + \alpha \delta$$

where $\delta = r(s, a) + \gamma \cdot \max_{a'} Q_\pi(s', a') - Q_\pi(s, a)$

In the equation above, the Q-value of any state-action pair is moved towards the Q-values of the greedy-policy, regardless of how they were generated. An
algorithm that can update the policy or value-functions from any experience, no matter how it was generated is classified as an off-policy algorithm.

In this section we introduced techniques to evaluate the value and action-value functions for a policy $\pi$. Temporal difference learning and the Bellman equation are also of use in actor critic methods where the ‘critic’ must learn evaluations of the actor’s policy such that the actor can improve its behavior.

### 2.2.2 On-Policy Learning with Policy Gradient Methods

While value-based methods learn the optimal action-value function from which an optimal policy can be extracted, policy gradient methods directly represent a policy with some form of parameterized function approximator. The approximator may be a linear approximator, or a non-linear approximator such as a deep neural network. Policy gradient algorithms define a policy parameterized by $\theta$ as $\pi_\theta$. The goal of policy gradient methods is to compute the gradient of $\theta$ with respect to the expected return the policy achieves while interacting with the environment. Then, once this gradient is known, one can perform gradient ascent to improve the policy:

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\pi_\theta)|_{\theta=\theta_k}$$

where $J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta} [R(\tau)]$ is the performance (expected return) of the policy with parameters $\theta$. Figure 2 shows an example of the effect of performing gradient ascent on the policy performance with parameters $\theta$.

The Policy Gradient Theorem [24] provides a way to numerically compute $\nabla_\theta J(\pi_\theta)$. The gradient is given as:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[ \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)Q_\pi_\theta(s_t, a_t) \right]$$

Intuitively, the above gradient will push up the probability of actions with high advantage (those that achieve higher return than expected), and push down the
Figure 2. Gradient ascent on parameterized policy performance.

probability of actions with lower advantage.

Eq. 17 computes the gradient as an expectation over all possible trajectories that could be sampled while following a policy. When there are a large number of trajectories, it is infeasible to exactly compute the policy gradient. Practical algorithms which utilize policy gradients such as REINFORCE form an estimate of the true policy gradient by collecting a set of trajectories $D = \{\tau_i\}_{i=1,...,N}$ from the environment.

$$
\nabla_\theta J(\pi_\theta) \approx \frac{1}{N} \sum_{\tau \in D} \sum_{t=0}^T \nabla_\theta \log \pi_\theta(a_t|s_t)Q_{\pi_\theta}(s_t, a_t)
$$

Here, we can see why policy gradient algorithms are on-policy. An approximate policy gradient must be recomputed after each gradient ascent step by collecting new sample trajectories from the environment. This is very sample inefficient, and is one of the main drawbacks of the method. However, there are benefits to policy gradient methods. First, they apply better to environments with large or continuous action spaces, as a stochastic policy can be learned which can easily be sampled from. Value-based methods learn deterministic policies, where the action
with the maximizing value must be computed, which could be problematic with large or continuous action spaces. Second, exploration is more natural, as the policy can be initialized to have high entropy.

2.3 Deep Reinforcement Learning

Deep reinforcement learning (DRL) is a subfield of reinforcement learning (RL) that combines the power of deep learning techniques with reinforcement learning algorithms to solve complex and high-dimensional problems. In deep reinforcement learning, deep neural networks are used as function approximators to represent either the policy, the value function, or both.

Deep neural networks [25] are function approximators with a set of parameters \( \theta \) that map input features \((x_1, x_2, ..., x_N) = x\) to estimated output values \((\hat{y}_1, \hat{y}_2, ..., \hat{y}_N) = \hat{y}\) through multiple layers of non-linear transformations of \(x\): \(f_{\theta}(x) \approx f(x)\). The parameters \( \theta \) of \(f_{\theta}\) can be improved to provide better approximations of the true function \(f\) by performing gradient descent (or ascent) on the parameter vector with respect to some objective function:

\[
\theta_{k+1} \leftarrow \theta_k - \alpha \nabla_{\theta} \mathcal{L}(f(x), f_{\theta}(x))|_{\theta=\theta_k}
\]  

(19)

where \( \alpha \) is the learning rate that controls the magnitude of the parameter change along the direction of the gradient, and \( \nabla_{\theta} \mathcal{L} \) is the vector of partial derivatives \( \frac{\partial}{\partial \theta} \mathcal{L} \) (the gradient) which is computed for each parameter in the network via the backpropagation algorithm.

Deep Q-Network (DQN) [26] is a value-based DRL algorithm that combines Q-learning with deep neural networks to approximate the action-value function. DQN was the first DRL algorithm to achieve human-level performance in several Atari games using only raw pixel input, demonstrating the potential of deep learning in reinforcement learning. In DRL, policy gradient can also easily be used with deep neural networks to directly optimize the policy, as they already
have the assumption of a parameterized policy. Trust Region Policy Optimization (TRPO) \[19\] is an example of a policy-gradient-based DRL method that showed impressive results in various simulated robotic control tasks with continuous action spaces.

Despite all of DRL’s success it still faces several challenges. DRL algorithms, including DQN and policy gradient methods, can suffer from instability during training. This is due to a combination of factors, such as the non-stationary nature of the learning problem, the presence of function approximators, and the highly correlated experience samples. These factors can lead to high variance in the updates, oscillations in performance, or even divergence. As we will see in a later section, methods such as TRPO attempt to alleviate this by enforcing small and controlled policy updates at each step. Furthermore, DRL relies on deep neural networks to approximate the value function or policy. However, this approximation can introduce errors, leading to biased estimates and suboptimal policies. In DQN, the use of target networks and experience replay helps alleviate this issue to some extent, but it remains a challenge for both DQN and policy gradient methods.
CHAPTER 3

Trust Region Optimization in Reinforcement Learning

3.1 Trust Region Methods

Trust region optimization [27] offers a reliable approach for tackling non-linear programming problems. This optimization strategy differs from traditional methods such as gradient descent, in that it focuses on a localized region surrounding the current best solution, rather than the entire search space. By defining a neighborhood, or “trust region,” around the current solution, the algorithm restricts the search space and optimizes a surrogate function that approximates the true objective function within this region.

The choice of surrogate function is critical to the success of trust region optimization, as it should closely mimic the behavior of the true objective function in the region of interest. A well-chosen surrogate enables the algorithm to identify promising directions for optimization and to adjust the size of the trust region accordingly. When the surrogate function closely approximates the true objective, it is possible to expand the trust region, potentially leading to faster convergence towards the optimal solution.

In cases where the original objective function exhibits complex behavior, direct optimization can be challenging. Trust region methods offer an effective alternative by focusing on smaller, localized regions and iteratively refining the solution. As the algorithm progresses, the trust region can be adjusted based on the agreement between the surrogate and the true objective function, ensuring that the method converges to a solution that is optimal or near-optimal.
3.2 Trust Region Policy Optimization

Policy gradient methods directly optimize a policy, which can lead to a stability increase over approximate value-based methods. However, policy gradients are optimized via gradient ascent, and must collect new samples from an environment each time a policy update is to be computed. To maintain stability, the learning rate $\alpha$ of the gradient ascent must be kept small. This leads to very poor sample efficiency and slow learning rates for vanilla policy gradient methods like those introduced in section 2.2.2. Increasing the size of $\alpha$ can cause the policy to change too quickly which may drastically change the on-policy sample distribution, causing performance collapse. Trust region policy optimization looks to trust region methods as a solution to this problem. Trust region policy optimization seeks to identify a good surrogate model to the policy performance which is easier to optimize, and then looks to compute the largest step size that is allowed to improve performance, while ensuring that the policy is not changed too much by enforcing a trust region constraint.

The theory in Trust Region Policy Optimization (TRPO) by Schulman et al. [19] is an important technique for solving policy optimization in a MDP. To begin, the following identity expresses the expected return (performance) of some other policy $\pi'$ in terms of the average advantage of the current policy $\pi_k$ following $\pi'$ (see [28] for proof):

$$J(\pi') = J(\pi_k) + \sum_s \rho_{\pi'}(s) \sum_a \pi'(a|s)A_{\pi_k}(s, a)$$  \hspace{1cm} (20)

where $\rho_{\pi'}$ is the unnormalized discounted state visitation frequencies

$$\rho_{\pi'}(s) = P(s_0 = s) + \gamma P(s_1 = s) + \gamma^2 P(s_2 = s) + \ldots,$$  \hspace{1cm} (21)

and where $s_0 \sim \rho_0$, the start state distribution of the MDP.

Eq. 20 implies that if $\pi'$ achieves nonnegative expected advantage at every
state \( s, \sum_a \pi'(a|s)A_{\pi_k}(s,a) \geq 0 \), then the policy \( \pi' \) is guaranteed to achieve a greater or equal performance than \( \pi_k \), i.e, \( J(\pi') \geq J(\pi_k) \).

It would be great if one could simply treat the right hand side of Eq. 20 as an objective function and find the policy \( \pi' \) to maximize it. However, it is difficult to optimize Eq. 20 for a new \( \pi' \) directly due to the functions complex dependency on the state visitation frequencies of the new policy, \( \rho_{\pi'} \). To solve this problem, Kakade et al. [28] introduce the following surrogate objective, which serves as local approximation to Eq. 20:

\[
L_{\pi_k}(\pi') = J(\pi_k) + \sum_s \rho_{\pi_k}(s) \sum_a \pi'(a|s)A_{\pi_k}(s,a)
\]  

The surrogate model \( L_{\pi_k}(\pi') \) differs from the true objective \( J(\pi') \) in that it ignores changes in state visitation frequency, placing dependence on the known \( \rho_{\pi_k} \), rather than the to be optimized \( \rho_{\pi'} \). This suggests that the surrogate model will provide a good approximation to the true objective of \( \pi' \) when the distance between \( \pi' \) and \( \pi_k \) is sufficiently small. Fig. 3 shows a graphical example of the surrogate model as \( \pi' \) strays from \( \pi_k \). This surrogate model provides a function that can be optimized with trust region techniques. We know that it can be trusted around some local area around the current policy, but we do not know how large this region should be.

The primary goal of TRPO is to solve the following optimization problem:

\[
\pi_{k+1} = \arg\max_{\pi' \in \Pi} L_{\pi_k}(\pi')
\]

s.t. \( D_{\text{KL}}(\pi'\|\pi_k) \leq \delta \)  

where the constraint \( D_{\text{KL}}(\pi'\|\pi_k) \leq \delta \) is the trust region constraint. \( D_{\text{KL}} \) is the Kullback-Leibler divergence, a measure of distance between two probability distributions.

Schulman et al. show that the above constrained optimization can be transformed to an unconstrained optimization problem with an adaptive penalization
They prove that the approximation error between the true objective Eq. 20 and the surrogate model Eq. 22 is bounded by:

\[ |J(\pi') - L_{\pi_k}(\pi')| \leq C \max_s D_{KL}[\pi'||\pi_k](s) \]

where

\[ C = \frac{4\gamma \epsilon}{(1 - \gamma)^2}, \quad \epsilon = \max_{s,a} |A_{\pi_k}(s, a)| \]  \hspace{1cm} (24)

By rearranging Eq. 24 to get the lower bound on the approximation error, one can derive a policy update rule with monotonically increasing policies:

\[ \pi_{k+1} = \arg \max_{\pi' \in \Pi} L_{\pi_k}(\pi') - C \max_s D_{KL}[\pi'||\pi_k] \]  \hspace{1cm} (25)

### 3.3 General Policy Optimization

Recent work titled “An Analytical Update Rule for General Policy Optimization” [18] extends previous work in trust region optimization methods by proving a new theoretical bound which improves Schulman’s result. It is important to note that the theoretical bound Eq. 24 derived in TRPO relies on the maximum KL-Divergence over the entire state space, \( \max_s D_{KL}[\pi'||\pi](s) \). In practical problems where the state space is very large, it could be infeasible to compute this term.
Furthermore, the maximum KL-Divergence could be infinity even if the policies are similar in most states.

[18] proved a new bound on the approximation error of the surrogate objective. The difference between the true policy performance $J(\pi')$ and surrogate model $L_\pi(\pi')$ can be bounded and is given as:

$$|J(\pi') - L_\pi(\pi')| \leq \frac{1}{1 - \gamma} C_\pi \mathbb{E}_{s \sim d^\pi} [D_{KL}[\pi'|\pi](s)]$$

where $C_\pi = \frac{\gamma^2 \epsilon}{(1 - \gamma)^3}$, $\epsilon = \max_{s,a} |A_\pi(s, a)|$.

The above bound improves the TRPO bound Eq. 24 by relating the error to the expected KL-Divergence over the state space, $\mathbb{E}_{s \sim d^\pi} [D_{KL}[\pi'|\pi](s)]$, which is tighter than the maximum KL-Divergence. By expanding the absolute value and rearranging Eq. 26, Li provides the following lower bound on the true policy performance:

$$J(\pi') \geq L_\pi(\pi') - \frac{1}{1 - \gamma} C_\pi \mathbb{E}_{s \sim d^\pi} [D_{KL}[\pi'|\pi](s)]$$

(27)

By maximizing the right hand side of Eq. 27, we are guaranteed to find a new policy $\pi'$ which performs better than or equal to the performance of the old policy $\pi$.

While it is possible to parameterize a policy $\pi$ and maximize Eq. 27 directly, [18] continued to solve the above optimization problem using a technique called calculus of variations [29]. The calculus of variation allows the maximization of Eq. 27 to be solved in the functional space, and provides a pathway to prove a new closed form policy update rule which has a monotonic improvement guarantee:

$$\pi_{new} = \pi_{old} \cdot \frac{e^{\alpha \pi_{old}}}{\mathbb{E}_{a \sim \pi_{old}} [e^{\alpha \pi_{old}}]}$$

(28)

where $\alpha = A_{\pi_{old}} / C_{\pi_{old}}$

By multiplying the numerator and denominator by $e^{V_{\pi_{old}} / C_{\pi_{old}}}$, the update rule in Eq. 28 can be slightly modified to rely on the action-value function $Q_{\pi_{old}}(s, a)$.
rather than the advantage function $A_{\pi_{\text{old}}}(s, a)$, which (as discussed in section 4.2) will be learned directly by the critic:

$$\pi_{\text{new}} = \pi_{\text{old}} \cdot \frac{\exp\{A_{\pi_{\text{old}}} / C_{\pi_{\text{old}}]\}}{\mathbb{E}_a \pi_{\text{old}}[\exp\{A_{\pi_{\text{old}}} / C_{\pi_{\text{old}}}\}]} \cdot \frac{\exp\{V_{\pi_{\text{old}}} / C_{\pi_{\text{old}}}\}}{\exp\{V_{\pi_{\text{old}}} / C_{\pi_{\text{old}}}\}}$$

\begin{equation}
\pi_{\text{new}}(a^i|s) = \pi_{\text{old}}(a^i|s) \cdot \frac{\exp\{Q_{\pi_{\text{old}}}(s, a^i) - V_{\pi_{\text{old}}}(s) - V_{\pi_{\text{old}}}(s) / C_{\pi_{\text{old}}} \}}{\sum_j \pi_{\text{old}}(a^j|s) \cdot \exp\{Q_{\pi_{\text{old}}}(s, a^j) - V_{\pi_{\text{old}}}(s) - V_{\pi_{\text{old}}}(s) / C_{\pi_{\text{old}}} \}}
\end{equation}

where $C_{\pi_{\text{old}}} = \frac{\gamma^2 \epsilon}{(1 - \gamma)^3}$, $\epsilon = \max_{s,a}|A_{\pi_{\text{old}}}(s, a)|$.

The modified update rule given in Eq. 29 can be viewed as a weighted softmax distribution of the old policy. To better understand the rule, one can consider how the probability of selecting a particular action $a^i$ in some state $s$ changes when the critic learns that its Q-value is higher than that of other actions. If $a^i$ has a higher Q-value than other actions, it will carry more weight in update rule, leading to an increase in its probability to be selected the next time $s$ is encountered. The constant $C_{\pi_{\text{old}}}$ controls the amplitude of the change in the policy to ensure that the performance is greater or equal to that of the old policy, $J(\pi_{\text{new}}) \geq J(\pi_{\text{old}})$. Since the update rule follows a softmax distribution, $\pi_{\text{new}}$ continues to be a probability mass function (PMF) over the set of actions $a \in A$. Figure 4 shows a graphical representation of how the update rule modifies a discrete policy in relation to each actions Q-value, following the above intuition.
Figure 4. Graphical representation of the update rule given in Eq. 29
CHAPTER 4

Discrete GPO: A Stochastic Off-Policy Actor-Critic Algorithm

Section 3.3 introduces the update rule from GPO [18] in the theoretical space. It was assumed that the policy and action-value function could be exactly computed for all states and actions. Also, the previous work provided no experiments which used the proposed theory to learn a policy in practice. The main contribution of this thesis is the development of an algorithm which utilizes approximations to the above theoretical foundations, with some tricks to mitigate instability due to approximation error. In real-world RL problems, the state space can be extremely large, and it may not be feasible to represent and learn the exact policy and action-value functions over all states and actions. To make the update rule useful in practice, approximations to the policy and value functions must be made over the entire space, and updates must be computed with finite samples from a learning environment. Deep neural networks are a good choice for approximating and learning generalizations of large, non-linear functions. In this section, both the actor’s policy and and critic’s action-value functions are parameterized with deep neural networks, and both the policy and critic values are learned via finite samples from a learning environment. Because the update rule does not rely on the computation of any on-policy expected values, it can be used to design an off-policy algorithm.

4.1 Parameterized Policy Optimization via Target Policy Minimization

We represent the stochastic policy learned by the actor with a deep neural network with parameters \( \theta \). Since this is an off-policy algorithm, experiences that are generated following any policy can be used to perform policy updates. To that end, as the agent interacts with its environment, each experience \( e_t = \{s_t, a_t, s_{t+1}, r_t, d_t\} \)
is stored into an experience replay buffer $\mathcal{D} = e_1, ..., e_N$ such that they can be used for a policy update in the future. Using past experiences to perform policy updates can greatly increase the sample efficiency of the algorithm, as collected experiences can be utilized more than once. Figure 5 shows an example of a parameterized policy network with discrete actions. Each output $\pi(a^i|s)$ is the probability of taking the action $a^i$ given the policy network is conditioned on the state features $\{x_1, x_2, ..., x_K\} \in s$.

![Image](image.png)

**Figure 5.** An example of a fully connected policy network.

To perform a policy update, a mini-batch of experiences is uniformly randomly sampled from the replay buffer, $B \sim \mathcal{D}$. The samples are drawn uniformly to increase the diversity of experiences seen by the actor and critic networks which reduces overfitting. First, the probability of selecting each action in the action space is computed for each state in the mini-batch by performing a forward pass on the policy network, $\pi_{\theta_k}(a^i|s), i = 1, \ldots, N, \forall s \in B$, where $N$ is the number of discrete actions in the action space. Then, new policy ‘targets,’ $\pi_{\text{new}}(a^i|s)$, are computed for each action in each state over the entire mini-batch using the
following slightly modified update rule:

\[
\pi_{\text{new}}(a^i | s) = \frac{\pi_{\theta_k}(a^i | s) \cdot \exp\{\bar{Q}(s, a^i)/C_{\pi_{\theta_k}}\}}{\sum_j \pi_{\theta_k}(a^j | s) \cdot \exp\{Q(s, a^j)/C_{\pi_{\theta_k}}\}}, \text{for } i = 1, 2, \ldots, N
\]

(30)

where \( C_{\pi_{\theta_k}} = \frac{\gamma^2 \epsilon}{(1 - \gamma)^2}, \epsilon = \max_{s,a} |A_{\pi_{\theta_k}}(s, a)| \).

and where \( \bar{Q}(s, a^i) \) are the action-values provided by the critic networks introduced later in section 4.2. The above update rule differs from the original update rule given in Eq. 29 in that the constant \( C \) is scaled by \( 1/(1-\gamma)^2 \), rather than \( 1/(1-\gamma)^3 \).

This is a heuristic modification that is made to decrease the size of \( C \), as smaller values for \( C \) will result in a greater change in the policy, which improves the learning rate of the algorithm.

The parameters \( \theta_k \) can then be learned by minimizing a measure of error between current estimates of the policy \( \pi_{\theta_k}(a|s) \) and the computed target policies \( \pi_{\text{new}}(a|s) \) via stochastic gradient descent:

\[
\theta_{k+1} \leftarrow \theta_k - \alpha \cdot \nabla_{\theta} \mathcal{L}_B(\pi_{\theta}, \pi_{\text{new}})|_{\theta=\theta_k}
\]

(31)

where \( \alpha \) is the learning rate and \( \mathcal{L}_B \) is some loss function of error over the batch \( B \), e.g. mean squared error (MSE) or mean KL-Divergence.

Finally, a maximum entropy term is added to the objective function to balance exploration even as the policy network becomes more deterministic, similar to the Soft Actor-Critic (SAC) algorithm [14] and Soft Q-Learning [20]. If using the mean squared error loss, the final objective for the actor network is written as:

\[
\text{minimize}_{\theta} \frac{1}{|B|} \sum_{s \in B} \left\{ \sum_{i=1}^N (\pi_{\text{new}}(a^i | s) - \pi_{\theta}(a^i | s))^2 - \lambda H(\pi_{\theta}(\cdot|s)) \right\}
\]

(32)

where \( \lambda \) is a tunable hyperparameter to control the weight placed on the contribution of \( H \), the Shannon entropy of the policy \( H(\pi_{\theta}(\cdot|s)) = -\mathbb{E}_{a \sim \pi_{\theta}(s)} \log \pi_{\theta}(\cdot|s) \).
4.2 Learning the Critic Network

The previous section describes how to formulate a parameterized policy $\pi_\theta$ and how to utilize the update rule in Eq. 30 to compute target policies from which the policy network parameters $\theta$ can be learned. In this section, we will describe how to learn the critic networks. In actor-critic methods, it is the critic's role to learn the action-value function $Q_\pi(s, a)$ for a given policy $\pi$, providing the actor with ‘knowledge’ of the action-values so that it can adapt its policy.

Previous work by Fujimoto [15] shows that parameterized approximators which learn improved Q-estimates via the bellman equation tend to over-estimate the true Q-values which can contribute to poor policy updates. To address this issue, they suggest learning two Q-approximators and taking the minimum of each approximators estimated Q-values when performing policy updates or Q updates. Following this suggestion, two deep neural networks with separate parameters $\phi_1$, $\phi_2$, are used to produce the critics Q-values for the actors policy updates. From now on, $Q(s, a) = \min(Q_{\phi_1}(s, a), Q_{\phi_2}(s, a))$.

Along with the primary online Q-networks $Q_{\phi_1}$ and $Q_{\phi_2}$, the critic also maintains two other Q-networks, called target networks, who's role is to reduce variance in the target Q-values computed with the one-step TD backup, which increases the Q-networks learning stability. Mnih [11] first introduced target networks with hard-parameter updates for the Deep Q-network (DQN) algorithm to provide a fixed Q-function to increase the stability of Q-networks learning when computing Q-update target values with the one-step Bellman backup. However, Lillicrap [13] introduces the Polyak averaging technique in Eq. 33 to provide a softer update to the target networks, resulting in smoother network training. Our critic utilizes the Polyak averaging technique, where parameters $\phi_{\text{targ},1}$ and $\phi_{\text{targ},2}$ are initialized to be equal to $\phi_1$ and $\phi_2$ respectively, but are updated to lag behind $\phi_1$ and $\phi_2$ as
gradient updates are applied:

\[
\phi_{targ,1} \leftarrow \tau \cdot \phi_{targ,1} + (1 - \tau) \cdot \phi_1 \\
\phi_{targ,2} \leftarrow \tau \cdot \phi_{targ,2} + (1 - \tau) \cdot \phi_2
\]

where \( \tau \) is a tunable hyper-parameter to control the lagging ‘distance’ of the target networks.

Both Q-networks are learned concurrently with the policy network. At each policy update with experience mini-batch \( B \in \mathcal{D} \), the Q-networks are trained to minimize the error between current estimated Q-values and target estimated Q-values computed with the one-step TD backup.

First, the Q-values for each discrete action are computed by performing a forward pass through networks \( Q_{\phi_1}(s, a^i), i = 1, \ldots, N \) and \( Q_{\phi_2}(s, a^i), i = 1, \ldots, N \) for each state \( s \in B \), from which the actions with the minimum value are selected to provide \( \bar{Q}(s, a^i), i = 1, \ldots, N \). The Q-value for each action \( \bar{Q}(s, a^i) \) is used by the actor in the computation of target policies with update rule Eq. 30.

Second, the Q-value estimates of each next state \( s' \in B \) are computed with a forward pass through each target network, \( Q_{\phi_{targ,1}}(s', a') \) and \( Q_{\phi_{targ,2}}(s', a') \) where \( a' \sim \pi_k(\cdot | s') \), from which their minimum is selected to get \( \bar{Q}_{targ}(s', a') \). Using the one-step TD equation (see section 2.2.1), target Q-values are computed for each state-action pair \( (s, a) \in B \) in the mini-batch, and parameters \( \phi_1 \) and \( \phi_2 \) are updated via a stochastic gradient descent step to minimize the mean squared error between the predicted \( Q_{\phi_1}(s, a), Q_{\phi_2}(s, a) \) values and the one-step Bellman backup target values:

\[
\min_{\phi_i} \frac{1}{|B|} \sum_{s,a,r,s',d \in B} (y(r, s', d) - Q_{\phi_i}(s, a))^2, \text{ for } i = 1, 2
\]

\( y(r, s', d) = r + \gamma (1 - d) \bar{Q}_{targ}(s', a'), \ a' \sim \pi_\theta(\cdot | s') \)
The pseudo code for the proposed algorithm, called “Discrete GPO”, is provided in [algorithm 1]. Pseudo code lines 1-2 describe the actor and critic network initialization, and the experience replay buffer initialization. Lines 4-7 represent the main agent-environment interaction, where samples for training are collected from the latest iteration of the policy. Lines 9-17 explain the target policy computation, policy loss, critic target computation, and critic losses. Figure 6 also provides a graphical representation of data flow through the actor’s policy network and critic’s Q-network during the learning procedure.

![Figure 6](image)

Figure 6. A graphical representation of the forward passes (solid arrows) and gradient flow (dashed arrows) through the actor and critic networks for a single mini-batch update.
Algorithm 1 DiscreteGPO

1: initialize policy parameters \( \theta \), Q-function parameters \( \phi_1, \phi_2 \), experience replay buffer \( D \)
2: set target Q-function parameters equal to main parameters \( \phi_{\text{targ},1} \leftarrow \phi_1, \phi_{\text{targ},2} \leftarrow \phi_2 \)
3: repeat
   4: observe state \( s \) and select action \( a \sim \pi_\theta(\cdot|s) \)
   5: execute action \( a \) in environment and observe reward \( r \), next state \( s' \), and terminal flag \( d \)
   6: store experience \( \{s, a, r, s', d\} \) into replay buffer \( D \)
   7: if \( s' \) is terminal, reset environment state
   8: if time to update then
      9: for \( n \) in range(number of updates) do
          10: sample mini-batch \( B \sim D \)
          11: compute minimum of two Q-estimates
              \[ \bar{Q}(s, a) = \min_{i=1,2} (Q_{\phi_i}(s, a)), \forall s \in B, \forall a \in A \]
          12: compute the maximum advantage in mini-batch \( B \)
              \[ \max A = \max \left| \bar{Q}(s, a) - \mathbb{E}_{a \sim \pi_\theta(\cdot|s)} [\bar{Q}(s, a)] \right| \]
          13: compute approximate trust region step size \( C = \gamma \cdot \max A / (1 - \gamma)^2 \)
          14: compute target policies \( \pi_{\text{new}} \) :
              \[ \pi_{\text{new}}(a|s) = \frac{\pi_\theta(a^i|s) \cdot e^{\bar{Q}(s,a^i)/C}}{\sum_j \pi_\theta(a^j|s) \cdot e^{\bar{Q}(s,a^j)/C}}, i = 1, \ldots, N \]
          15: compute TD target for Q-functions:
              \[ y(r, s', d) = r + \gamma(1 - d) \cdot \min_{i=1,2} (Q_{\phi_{\text{targ},i}}(s', a')) \]
          16: update policy and critics with one step of gradient descent:
              \[ \nabla_{\theta} \frac{1}{|B|} \sum_{s \in B} [(\pi_\theta(a|s) - \pi_{\text{new}}(a|s))^2 - \lambda H(\pi_\theta(\cdot|s))] \]
              \[ \nabla_{\phi_i} \frac{1}{|B|} \sum_{(s, a, r, s', d) \in B} (Q_{\phi_i}(s, a) - y(r, s', d))^2 \text{ for } i = 1, 2 \]
          17: apply \( \phi_{\text{targ},i} \leftarrow \tau \cdot \phi_{\text{targ},i} + (1 - \tau) \cdot \phi_i \) for \( i = 1, 2 \)
      18: end for
  19: end if
20: until convergence
CHAPTER 5
Experiments and Simulations

The preceding section described the design of the Discrete GPO learning algorithm, which was inspired from the theory provided in [18]. In this section, the performance of the algorithm is tested and compared via experimentation. All of the following experiments were performed in the Python programming language, using PyTorch [30] as the deep learning framework. The popular package “Gym” [31] was used to provide an API to different simulated learning environments. Gym was developed by OpenAI to provide a collection of experimentation environments to RL researchers in effort to reduce variance in reported results.

Figure 7. Renderings of CartPole (top-left), LunarLander (top-right), and Breakout (bottom-center) Environments.

Fig. 7 shows screenshots of the CartPole, LunarLander, and Breakout envi-
ronments to provide the reader with visual intuition about the agent’s task in each experiment.

The algorithms that Discrete GPO was compared to are Deep Q-Network (DQN) [26] and Proximal Policy Optimization (PPO) [23]. DQN and PPO are two state of the art reinforcement learning algorithms for value-based off-policy learning and policy-gradient based on-policy learning respectively.

The implementation of DQN used in the experiments performed below has some differences compared to the implementation documented in the original paper [26]. Specifically, the target Q-network parameters are updated using the Polyak averaging technique [13], which involves updating them slightly towards the online Q-network parameters at each gradient step, rather than using a fixed interval hard target Q-network parameter update. Additionally, to address over-estimation error accumulation in the Q-networks, this implementation of DQN maintains two Q-networks, and the minimum of the two networks’ outputs is used as the estimated values. This approach was originally recommended by Fujimoto et al. [15]. Both of the above techniques are also used when implementing the critic networks in Discrete GPO algorithm 1 and were introduced in section 4.2.

PPO is a current state of the art on-policy reinforcement learning algorithm. PPO aims to approximate the primary goal of the TRPO algorithm - keeping new policy updates close to old - while being simpler to implement and more efficient for larger networks. PPO restricts the change in policy parameters by clipping the policy ratio in the objective function, removing the incentive for the policy to change too far. The Stable-Baselines3 (SB3) [32] implementation of PPO was used in the following experiments. SB3 applies several modifications to the original PPO algorithm in their implementation which can be found in the ICLR blog post by Huang et al., “The 37 Implementation Details of Proximal Policy
Optimization” [33].

5.1 A Case Study in CartPole and LunarLander Environments

In this section, the performance of Discrete GPO, DQN, and PPO are compared on the CartPole-v1 and LunarLander-v2 environments. Below are brief descriptions of each environment, their state representations, action space, and reward functions.

5.1.1 Environment Descriptions

- **CartPole** is a famous classic control task from RL literature [34]. Thanks to its relative simplicity, it serves as a good early benchmark to evaluate the stability of RL algorithms. Each state is represented by four descriptive features: the cart’s position along the horizontal axis, the cart’s velocity along the horizontal axis, the angle of the pole relative to the horizontal axis, and the angular velocity of the pole. At each transition, the agent must decide between two actions: apply a force to the left of the cart, or to the right. The reward function provides the agent with +1 reward every transition where the pole angle and cart position are within the acceptable bounds. If the cart or the pole move outside the acceptable bounds, the episode is terminated. Version v1 of the CartPole environment was used for these experiments, and the environment is considered solved when the agent can keep the pole balanced for 500 time steps (e.g., the agent scores 500 episodic return).

- **LunarLander** represents a more challenging task. Each state is represented by eight features: the coordinates of the lander in x and y, its linear velocities in x and y, its angle and angular velocity, and two boolean values to represent if a leg is touching the surface. The agent has four actions to choose: fire the main engine on the bottom of the craft, fire the left side
engine, the right side engine, or no-op (do nothing). The reward function is more complicated. The agent receives -100 reward if the lander crashes (its body makes contact with the surface), +140 reward if the lander makes it in between the goal flags and an additional +100 if it comes to rest, and +10 reward for each leg that is in contact with the ground. To give incentive for the agent to land the craft swiftly, firing the main engine provides -0.3 reward per frame, while firing the side engines provides -0.03 per frame. Version v2 of the LunarLander environment was used for these experiments, and the environment is considered solved if the agent receive more than 200 return from its landing execution during an episode.

5.1.2 Experiment Configuration

Six learning trials with different pseudo random number generator (RNG) seeds were executed for each algorithm in the two environments. Evaluating the performance of an RL algorithm under multiple different RNG seeds is important because it reveals how robust the method is to slightly different learning scenarios and initial conditions [12].

The hyperparameters and network architectures were left unchanged for each experiment. Table 1 lists the hyperparameters used in both the CartPole and LunarLander experiments, while table 3 provides descriptions for each hyperparameter. For Discrete GPO, the actor network and critic networks were implemented with an MLP architecture with two hidden layers of sizes [256, 256]. The Rectified Linear Unit (ReLU) activation was applied to the output of each hidden layer. A LogSoftmax activation was applied to the output of the actor network to ensure the exponential of the output is a PMF that sums to 1. The output of the critic networks used no activation function. For DQN, the Q-networks used the same network architecture as the Discrete GPO critic networks. For PPO, the default
MLP networks configured by Stable-Baselines3 were used. It has two hidden layers, but with sizes \([64, 64]\) and Hyperbolic Tangent activation.

Every 1000 time steps, training was paused and the latest policy of each algorithm was evaluated in a separate evaluation environment by recording the average return accumulated over 5 episodes of play. The evaluation environment is an exact copy of the training environment, except that it’s RNG seed is unknown. The separate evaluation environment has an independent RNG seed chosen at random to demonstrate the generalization capabilities of the learnt policy, which is represented by a deep neural network. During each evaluation procedure, Discrete GPO and PPO were evaluated whilst following a deterministic policy extrapolated from the current stochastic policy, \(a = \arg \max_a \pi_{\theta_k}(a|s)\).

5.1.3 Results and Findings

Discrete GPO, DQN, and PPO were each trained for a total of 50,000 time steps in the CartPole environment. Figure 8 plots the average evaluation episode return for CartPole accumulated over 5 episodes of play in the evaluation environment. The solid line shows the mean return at each time step over the six different training seeds. The shaded regions show the 95% confidence interval around the mean performance of each method. From the results, it is clear that the off-policy algorithms, DQN (blue) and Discrete GPO (orange) have a fast initial learning rate. However, DQN struggles to reach the 500 reward goal. Fast and unstable learning is a common property of DQN because learning and behavior occurs following a parameterized greedy-policy.

At the time of the fist evaluation procedure (1000 time steps), Discrete GPO has already learned a functional policy across all six random seeds. Then, after an initial performance decrease that was common across all six random seeds as shown by the orange error band, GPO steadily climbs to reach the 500 return goal, and
maintains it for all six seeds as training continues. In the CartPole environment, PPO (green) is able to consistently learn faster than GPO and DQN.

Along with CartPole, Discrete GPO, DQN, and PPO were each trained for a total of 500,000 time steps in the LunarLander-v2 environment. Figure 9 plots the average evaluation episode return achieved by each algorithm in the LunarLander-v2 accumulated over 5 episodes of play in the evaluation environment. Clearly, Discrete GPO (orange) learns a well performing policy, quickly surpassing the goal 200 episodic return at around 75,000 time steps for each seed. While DQN is able to learn a satisfactory policy, there is high variance in episode return as shown by the jagged blue error bands. DQN also shows a much slower learning rate when compared to Discrete GPO and PPO. Initially, PPO (green) achieves a highly negative reward, which clips below the range of the graph. However, by 50,000 time steps, PPO has recovered and reaches the goal 200 return at around 125,000 time steps. PPO continues to learn steadily, and eventually surpasses the
performance of Discrete GPO.

From the CartPole and LunarLander experiments, it is clear that Discrete GPO is able to quickly learn a highly rewarding policy with very nice stability. It competes with the performance of PPO and outperforms DQN in both tasks. In LunarLander-v2, its sample efficiency benefits are very clear, as it learns a rewarding policy significantly faster than PPO.

Figure 9. Average episodic return of Discrete GPO, PPO, DQN on LunarLander-v2 evaluation environment.
Table 1. Discrete GPO, DQN, and PPO hyperparameter table for experiments in CartPole-v1 and LunarLander-v2 Environments.

<table>
<thead>
<tr>
<th>Hyperparameter</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Discrete GPO</td>
</tr>
<tr>
<td>discount factor $\gamma$</td>
<td>0.99</td>
</tr>
<tr>
<td>target polyak $\tau$</td>
<td>0.995</td>
</tr>
<tr>
<td>entropy coef. $\lambda$</td>
<td>1 e$-6$</td>
</tr>
<tr>
<td>final epsilon</td>
<td>N/A</td>
</tr>
<tr>
<td>final epsilon timestep</td>
<td>N/A</td>
</tr>
<tr>
<td>update freq.</td>
<td>50</td>
</tr>
<tr>
<td>update epochs</td>
<td>50</td>
</tr>
<tr>
<td>learning rate $\alpha$</td>
<td>0.001</td>
</tr>
<tr>
<td>mini-batch size</td>
<td>256</td>
</tr>
<tr>
<td>replay buffer size</td>
<td>50,000</td>
</tr>
<tr>
<td>initial buffer size</td>
<td>1,000</td>
</tr>
<tr>
<td>clip range</td>
<td>N/A</td>
</tr>
<tr>
<td>value-function coefficient</td>
<td>N/A</td>
</tr>
<tr>
<td>gae lambda</td>
<td>N/A</td>
</tr>
<tr>
<td>num envs</td>
<td>1</td>
</tr>
<tr>
<td>random seeds</td>
<td>{1-6}</td>
</tr>
</tbody>
</table>
5.2 Playing Games from Images

To evaluate the performance of the Discrete GPO algorithm in environments with complex and highly abstract observations, learning trials were ran in the Atari 2600 game Breakout 7. In Breakout, the goal is to bounce a ball with a controllable paddle to break blocks above. A block will break when the ball strikes it and points are awarded. The challenge is to keep the ball in play by not allowing it to pass the paddle, and by returning the ball back to the blocks to continue collecting more points. The environment has four discrete actions. NOOP (do nothing), move the paddle left, move the paddle right, or fire the ball (needed to start each episode). Each observation (state) from the environment are the pixels which make up the current frame of the game. Breakout is provided by the Arcade Learning Environment (ALE) package [35], and many different Atari specific environment pre-processing was applied as recommended by the RL community.

5.2.1 Environment Pre-Processing

In the following experiment, all algorithms are learning policies to play Breakout from vision. A convolutional neural network (CNN) [36] is used to learn encodings from the high-dimensional pixel space into a lower dimensional feature vector from which the actor and critic networks compute their values on. This represents a significantly more challenging problem, as vital information for decision making such as the balls velocity, ball trajectory, and paddle location are not explicitly provided and must be learned by the CNN encoder.

Before feeding these observations into a CNN encoder, several observation ‘wrappers’ are applied to modify the pixel observations provided from the environment. The original “BreakoutNoFrameSkip-v4” environment used in the following experiment provides pixel observations of shape (3, 210, 160) for the RGB color channels, height, and width of the image respectively. The following environment
pre-processing techniques are taken from the Nature DQN article [26]. First, the three RGB color channels are reduced to one gray-scale channel. Then, the observation is resized to (1, 84, 84), for the gray-scale channel, height, and width respectively. Reducing the resolution of the image allows for less compute and memory needs when running experiments, while retaining the original semantics of each frame. The environment is also configured to skip four frames between observations, while repeating the agents last action. This is because the default framerate of the game is high, so the new information that arises in the between frames may be minuscule. By skipping four frames between observations, more noticeable change is produced in the environment. To provide the agent with the opportunity to learn differential information, such as the velocity of the ball and its trajectory, each observation stacks the frames from the last four observations, changing the observation shape to (4, 84, 84), where the first dimension is the gray-scale values for each pixel at each observation. This provides the opportunity for the CNN encoder to notice changes such as the ball’s location and extract information such as ball movement direction and velocity.

Along with the observational pre-processing from the Nature DQN paper, other pre-processing modifications were applied to the environment following the suggestions from “9 Atari-specific Implementation Details” in the ICLR blog post [33]. These include the “NoopResetEnv”, which adds stochasticity to the environment by automatically repeating the ‘no operation’ (NOOP) action for a random number of time steps at the beginning of each episode. The “EpisodicLifeEnv” wrapper, which returns a terminal flag each time the agent loses a life, and not only when the agent has lost all of its lives. The “FireResetEnv” wrapper which automatically performs the ‘fire’ action to make the episode begin. This wrapper helps avoid cases where an agent repeatedly plays the NOOP action to
prevent the episode starting. Lastly, the “ClipRewardEnv”, which clips all of the rewards received from the environment to -1, 0, 1 depending on the sign of the reward.

5.2.2 Shared Convolutional Neural Network Encoder

For Atari games, Discrete GPO uses a CNN feature encoder that is shared between the actor and critic networks. This helps to reduce computation as only one large CNN feature extractor is needed, rather than two. Also, it is intuitive to imagine that the features needed to be learned to compute a good policy are likely similar to the features needed to compute good action-value estimates.

The CNN encoder transforms the (4, 84, 84) input observations into a feature vector of length 512. The actor and critic network heads take this feature vector as an encoded observation input, and output the policy and action-values respectively. The first layer in the CNN encoder convolves the input observation with 32 4 × 4 filters with stride 4 and applies the ReLU activation. The second layer convolves 64 4 × 4 filters with stride 2 and applies the ReLU activation. The third layer convolves 64 3 × 3 filters with stride 1, applies a ReLU activation, and flattens the layer to arrive at a 1 dimensional feature vector with 3136 neurons. Finally, a fully connected layer maps the 3136 neurons to 512 neurons, and a ReLU activation is applied. Figure 10 shows the architecture of the shared CNN encoder.

The actor’s policy network and critic’s Q-Networks map the CNN encoding to a policy and action-values respectively through a single fully connected layer with 256 hidden neurons and ReLU activation. A LogSoftmax function is applied to the output of the policy network to ensure the exponential of the policy sums to 1.

The DQN Q-networks use the same CNN encoder architecture as Discrete GPO but do not have the actor head. For PPO, the default Stable-Baselines3
CNN policy network was used.

Figure 10. Shared CNN feature encoder architecture.
5.2.3 Breakout Performance Comparison

Discrete GPO, DQN, and PPO were trained for 10 million time steps in the Breakout environment. Similarly to the CartPole and LunarLander experiments, training was paused every 2000 time steps, and the latest policy of each algorithm was evaluated over 5 episodes of play in a separate evaluation environment. Each algorithm was tested with random seeds 1, 2, 3.

To help aid exploration in Discrete GPO, the entropy coefficient $\lambda$ was linearly annealed from a starting value of $10^{-6}$ to $2.5 \times 10^{-8}$ over the first 3 million steps of training. Table 2 provides the hyper parameters used for each algorithm.

![Figure 11. Average episodic return of Discrete GPO, PPO, DQN on BreakoutNoFrameskip-v4 evaluation environment.](image)

Figure 11 plots the average evaluation return over the course of 10 million training steps. The performance data shown in the plot was smoothed using an averaging sliding window of length 100. It is clear from the graph that DQN (blue) has a substantially faster learning rate than both Discrete GPO and PPO. Again, this is due in part to DQN directly learning a greedy policy, which aggressively
and deterministically seeks to maximize the Q-Values at every step. However, in the DQN learning trial with random seed 3, the performance dropped to near 0 at around 2.3 million steps. On this seed, DQN struggled to recover, which is shown clearly by the wide error bands.

PPO (green) has a slower initial learning rate, but steadily increases to a highly rewarding policy - scoring about 325 episodic return on average. Clearly, Discrete GPO (orange) underperforms in this task. While the learning rate is higher than PPOs early in training, it struggles to improve to find a better policy. Still, the confidence band suggests that Discrete GPO is more robust to different random seeds than DQN, and similar in stability to PPO.

One possible reason for Discrete GPO’s poor performance is a side effect of off-policy learning. The actor and critic networks are being trained on experiences from many different policies from the replay buffer. This is called the ‘behavioral distribution’. When trained on data following the ‘behavioral distribution’, the actor and critic networks are required to learn good approximations over a much more diverse set of experiences than the experiences collected only following the latest policy (as done in on-policy methods like PPO). Poor or inconsistent approximations of the action-value function can cause poor and inconsistent policy updates. Ways that this side effect could be improved are discussed later in future works.
Table 2. Discrete GPO, DQN, and PPO hyperparameter table for experiments in BreakoutNoFrameskip-v4 environment.

<table>
<thead>
<tr>
<th>Hyperparameter</th>
<th>Discrete GPO</th>
<th>DQN</th>
<th>PPO</th>
</tr>
</thead>
<tbody>
<tr>
<td>discount factor $\gamma$</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
</tr>
<tr>
<td>target polyak $\tau$</td>
<td>0.995</td>
<td>0.995</td>
<td>N/A</td>
</tr>
<tr>
<td>entropy coef. $\lambda$</td>
<td>1e−6</td>
<td>N/A</td>
<td>0.01</td>
</tr>
<tr>
<td>entropy coef. end</td>
<td>2.5e−8</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>entropy coef. end step</td>
<td>3e6</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>final epsilon $\epsilon$</td>
<td>N/A</td>
<td>0.05</td>
<td>N/A</td>
</tr>
<tr>
<td>final epsilon timestep</td>
<td>N/A</td>
<td>1e6</td>
<td>N/A</td>
</tr>
<tr>
<td>update freq.</td>
<td>50</td>
<td>4</td>
<td>1024</td>
</tr>
<tr>
<td>update epochs</td>
<td>25</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>learning rate $\alpha$</td>
<td>2.5e−4</td>
<td>2.5e−4</td>
<td>2.5e−4</td>
</tr>
<tr>
<td>learning rate end step</td>
<td>N/A</td>
<td>N/A</td>
<td>10e6</td>
</tr>
<tr>
<td>mini-batch size</td>
<td>256</td>
<td>32</td>
<td>256</td>
</tr>
<tr>
<td>replay buffer size</td>
<td>1e6</td>
<td>1e6</td>
<td>N/A</td>
</tr>
<tr>
<td>initial experiences</td>
<td>50,000</td>
<td>50,000</td>
<td>N/A</td>
</tr>
<tr>
<td>clip range</td>
<td>N/A</td>
<td>N/A</td>
<td>0.1</td>
</tr>
<tr>
<td>value-function coef.</td>
<td>0.5</td>
<td>N/A</td>
<td>0.5</td>
</tr>
<tr>
<td>gae lambda</td>
<td>N/A</td>
<td>N/A</td>
<td>0.95</td>
</tr>
<tr>
<td>num envs</td>
<td>8</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>random seeds</td>
<td>{1, 2, 3}</td>
<td>{1, 2, 3}</td>
<td>{1, 2, 3}</td>
</tr>
</tbody>
</table>


Table 3. List of hyperparameters and their description.

<table>
<thead>
<tr>
<th>Hyperparameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>discount factor $\gamma$</td>
<td>Used to control farsightedness of agent.</td>
</tr>
<tr>
<td>target polyak $\tau$</td>
<td>Coefficient for soft target network updates.</td>
</tr>
<tr>
<td>entropy coef. $\lambda$</td>
<td>Coefficient upon maximum entropy term in policy objective.</td>
</tr>
<tr>
<td>entropy coef. end</td>
<td>Final value to linearly decay the entropy coefficient to.</td>
</tr>
<tr>
<td>entropy coef. end step</td>
<td>Time step to which entropy coefficient end is decayed to.</td>
</tr>
<tr>
<td>final epsilon $\epsilon$</td>
<td>DQN specific - Final epsilon for epsilon greedy exploration policy.</td>
</tr>
<tr>
<td>final epsilon timestep</td>
<td>DQN specific - Timestep to which epsilon is decayed to.</td>
</tr>
<tr>
<td>update freq.</td>
<td>Frequency of training procedure.</td>
</tr>
<tr>
<td>update epochs</td>
<td>Number of minibatch updates per training procedure.</td>
</tr>
<tr>
<td>learning rate $\alpha$</td>
<td>Learning rate for SGD optimizer.</td>
</tr>
<tr>
<td>learning rate end step</td>
<td>PPO specific - Timestep to which optimizer learning rate is decayed to.</td>
</tr>
<tr>
<td>minibatch size</td>
<td>Number of $(s, a, r, s', d)$ experiences over which each SGD update is computed.</td>
</tr>
<tr>
<td>replay buffer size</td>
<td>Number of experiences stored.</td>
</tr>
<tr>
<td>initial experiences</td>
<td>Number of random samples generated for replay buffer before training.</td>
</tr>
<tr>
<td>clip range</td>
<td>PPO specific - clips PPOs objective function to remove incentive for new policy to stray far from old policy.</td>
</tr>
<tr>
<td>value-function coef.</td>
<td>PPO specific - value-function coefficient placed on loss function.</td>
</tr>
<tr>
<td>gae lambda</td>
<td>PPO specific - Factor for trade-off of bias vs variance for Generalized Advantage Estimator.</td>
</tr>
<tr>
<td>num envs</td>
<td>The number of vectorized environments.</td>
</tr>
<tr>
<td>random seeds</td>
<td>The random seeds that pseudo RNGs are initialized to.</td>
</tr>
</tbody>
</table>
CHAPTER 6

Conclusion and Future Work

This thesis shows that an off-policy algorithm which learns a stochastic policy can be implemented from the theoretical update rule given in Eq. 29, and name the algorithm ‘Discrete GPO.’ The update rule was achieved using trust region methods, and holds a monotonic improvement guarantee on policy performance in the theoretical space, which makes it an attractive avenue for approximate algorithm implementations. It is demonstrated through simulated experiments that Discrete GPO can learn new policies in a very stable and efficient manner in environments with well defined state features, and that Discrete GPO does not suffer from the performance collapse problems of other off-policy methods such as DQN. In this setting, it is clear that Discrete GPO competes with the best implementation of the state of the art algorithm PPO. Discrete GPO is tested in a more challenging Atari environment, where significant approximation errors are more likely, and complex exploration is required. While Discrete GPO is unable to learn a highly rewarding policy like PPO, it still manages to avoid performance collapse when compared to DQN.

There are several avenues for which this research could be continued. First, more experimentation in other Atari environments would provide a better sense of the algorithms true capability in complex domains. It is not uncommon for algorithms to struggle in specific environments, while performing well in others. Furthermore, more hyperparameter search could be conducted to find values that work well for the vision domain. Second, for more challenging environments like Atari, the addition of a prioritized experience replay buffer [37] may help reduce poor policy updates that are induced by approximation error in the critic net-
work. Prioritized experience replay replaces the sampling of mini-batches from the replay buffer from a uniform distribution to a weighted distribution, where experiences with larger approximation error are weighted to be more likely to be sampled in future mini-batch updates. Third, better exploration techniques could be explored. Instead of using a maximum entropy term which can be difficult to tune, an intrinsic curiosity reward [38] could be added at each step. An intrinsic curiosity reward signal motivates an agent to select actions of which a learned generative model of the environment poorly predicts the next-state of. Lastly, an on-policy version of this algorithm could be explored. While this would remove the benefits of off-policy learning, it would be interesting to compare an on-policy update approach to state of the art on-policy algorithms like PPO.
LIST OF REFERENCES


