REPUBLIC OF TURKEY
YILDIZ TECHNICAL UNIVERSITY
GRADUATE SCHOOL OF SCIENCE AND ENGINEERING

OVERCOMING HARD EXPLORATION PROBLEM IN MARKOV DECISION PROCESS USING INVERSE REINFORCEMENT LEARNING AND HUMAN DEMONSTRATIONS

Wadhah Zeyad Tareq TAREQ

DOCTOR OF PHILOSOPHY THESIS
Department of Computer Engineering
Computer Engineering Program

Supervisor
Assoc. Prof. Dr. Mehmet Fatih AMASYALI

April, 2022

REPUBLIC OF TURKEY
YILDIZ TECHNICAL UNIVERSITY
GRADUATE SCHOOL OF SCIENCE AND ENGINEERING

OVERCOMING HARD EXPLORATION PROBLEM IN MARKOV DECISION PROCESS USING INVERSE REINFORCEMENT LEARNING AND HUMAN DEMONSTRATIONS

A thesis submitted by Wadhah Zeyad Tareq TAREQ in partial fulfillment of the requirements for the degree of **DOCTOR OF PHILOSOPHY** is approved by the committee on 01.04.2022 in Department of Computer Engineering, Computer Engineering Program.

Assoc. Prof. Dr. Mehmet Fatih AMASYALI
Yildiz Technical University
Supervisor

Approved By the Examining Committee

Assoc. Prof. Dr. Mehmet Fatih AMASYALI, Supervisor
Yildiz Technical University _____

Prof. Dr. Sırma YAVUZ, Member
Yildiz Technical University _____

Prof. Dr. Hatice KÖSE, Member
Istanbul Technical University _____

Assoc. Prof. Dr. Ayşe Betül OKTAY, Member
Yildiz Technical University _____

Asst. Prof. Dr. Akhtar JAMIL, Member
Istanbul Sabahattin Zaim University _____

I hereby declare that I have obtained the required legal permissions during data collection and exploitation procedures, that I have made the in-text citations and cited the references properly, that I haven't falsified and/or fabricated research data and results of the study and that I have abided by the principles of the scientific research and ethics during my Thesis Study under the title of Overcoming Hard Exploration Problem in Markov Decision Process Using Inverse Reinforcement Learning and Human Demonstrations supervised by my supervisor, Assoc. Prof. Dr. Mehmet Fatih AMASYALI. In the case of a discovery of false statement, I am to acknowledge any legal consequence.

Wadhah Zeyad Tareq TAREQ

Signature

Dedicated to
my father, my mother,
my wife, and my sisters.

ACKNOWLEDGEMENTS

I would like to express my deepest appreciations to my adviser Assoc. Prof. Dr. Mehmet Fatih AMASYALI for his guidance, valuable suggestions and creative comments throughout the research. It would not be possible to complete this thesis without his persistent help and patience. I consider myself very fortunate for being able to work with a very considerate and encouraging adviser like him.

I also want to show my sincere gratitude to the members of the thesis committee, Prof. Dr. Sırma YAVUZ and Prof. Dr. Hatice KÖSE. I owe many thanks to them for their valuable comments and suggestions while studying on the thesis.

Wadhah Zeyad Tareq TAREQ

TABLE OF CONTENTS

LIST OF ABBREVIATIONS vii

LIST OF FIGURES ix

LIST OF TABLES xi

ABSTRACT xii

ÖZET xiv

1 INTRODUCTION 1

1.1 Literature Review 1

1.1.1 Deep Reinforcement Learning 1

1.1.2 Learning from Demonstrations 2

1.1.3 Inverse Reinforcement Learning 3

1.2 Objective of the Thesis 3

1.3 Hypothesis 4

2 BACKGROUND 5

2.1 Machine Learning 5

2.1.1 Supervised Learning 5

2.1.2 Unsupervised Learning 6

2.1.3 Reinforcement Learning 6

2.2 Markov Decision Process (MDP) 8

2.3 Q-Learning 9

2.4 Convolutional Neural Networks 10

2.4.1 Input layer 11

2.4.2 Convolutional Layer 11

2.4.3 Pooling Layer 12

2.4.4 Fully-connected Layer 12

3 DEEP REINFORCEMENT LEARNING AND INVERSE REINFORCEMENT LEARNING 13

3.1 Deep Q-Networks 13

3.1.1 Preprocessing . . . 14
3.1.2 Model architecture . . . 15
3.1.3 Learning by interacting . . . 16
3.1.4 Training details . . . 17
3.2 Deep Q-network Extensions . . . 18
3.2.1 Double Q-Learning . . . 19
3.2.2 Prioritized Experience Replay . . . 19
3.2.3 Dueling Network Architectures . . . 20
3.2.4 Rainbow . . . 21
3.3 Deep Q-learning from Demonstrations . . . 22
3.3.1 The Pre-training Phase . . . 22
3.3.2 The Interacting Phase . . . 23
3.4 Inverse Reinforcement Learning . . . 24
3.4.1 Margin optimization . . . 26
3.4.2 Entropy optimization . . . 27
3.4.3 Bayesian update . . . 27
3.4.4 Classification and Regression . . . 29

4 PROPOSED SYSTEM 31
4.1 Inclusion External Reward . . . 31
4.1.1 Pre-Training Phase . . . 32
4.1.2 Interacting Phase . . . 34
4.2 Inclusion Inverse Reinforcement Learning . . . 35
4.2.1 Preparation of Reward Function Learning Datasets . . . 36
4.2.2 Reward Function Architecture . . . 37
4.3 The Integrated Agent . . . 37

5 EXPERIMENTAL SETUP 40
5.1 Environment Setup . . . 40
5.2 Experimental Results and Discussion . . . 42
5.2.1 Pre-training Phase . . . 42
5.2.2 Interacting Phase . . . 52

6 RESULTS AND DISCUSSION 57
6.1 Suggestions . . . 58

REFERENCES 59

PUBLICATIONS FROM THE THESIS 63

LIST OF ABBREVIATIONS

DRL	Deep Reinforcement Learning
DIRL	Deep Invirase Reinforcement Learning
RL	Reinforcement Learning
IRL	Invirase Reinforcement Learning
DQN	Deep Q-Network
CNN	Convolutional Neural Network
DDQN	Double Deep Q-Network
PER	Prioritized Experience Replay
DDDQN	Dueling Double Deep Q-network
A3C	Asynchronous Advantage Actorcritic
DQfD	Deep Q from Demonstrations
LfSD	Learning from Single Demonstration
IL	Imitation Learning
PDDQN	Prioritized Double DQN
GANs	Generative Adversarial Networks
VI	Value Iteration
iter	Iteration
NLP	Natural Language Processing
ALE	Arcade Learning Environment
RMSProp	Root Mean Square Propagation
TD	Temporal Difference
REIRL	Relative Entropy Inverse Reinforcement Learning
BIRL	Bayesian Inverse Reinforcement Learning

MCMC	Markov Chain Monte Carlo
GPIRL	Gaussian Process Inverse Reinforcement Learning
MLIRL	Maximum Likelihood Inverse Reinforcement Learning
SCIRL	Structured Classification Inverse Reinforcement Learning
CSIRL	Cascaded Supervised Inverse Reinforcement Learning
FIRL	Feature Inverse Reinforcement Learning
RF	Reward Function

LIST OF FIGURES

Figure 2.1 Reinforcement learning entities and their communications 7
Figure 2.2 The agent–environment interaction in a Markov decision process [27] 8
Figure 2.3 The Architecture of Convolutional Neural Networks [33] 11
Figure 2.4 The structure of convolutional and pooling layers [34] 12
Figure 3.1 The Breakout game in the Markov decision process form [41] . . . 14
Figure 3.2 A fully preprocessed state for breakout. 15
Figure 3.3 Schematic illustration of the convolutional neural network used in DQN [2] 16
Figure 3.4 Basic (top) and dueling (bottom) DQN architecture [7] 21
Figure 3.5 RL (left) vs IRL (right). 25
Figure 4.1 Demonstration building procedure 32
Figure 4.2 The performance of our agent in phase two for a hard exploration environment. The agent starts learning from self-generated data after performing 20,000 steps. The results are normalized between 0 and 1 35
Figure 4.3 The left plots show the standard MDP for solving Reinforcement Learning problems. The right plots show our new framework by adding the Reward Function 36
Figure 4.4 Reward function dataset building procedure 36
Figure 4.5 The difference between our agent architecture (left) and the DQfD agent architecture (right) 39
Figure 5.1 Screenshots from five Games: (Left-to-right) Montezuma's Revenge, Pitfall, Private Eye, Breakout, and Pong. 41
Figure 5.2 The difference between 8 actions and 18 action results. (a) plot shows the maximum scores. (b) plot shows the average scores . . . 44
Figure 5.3 The difference between 0.9 PER and 0.6 PER results. (a) plot shows the maximum scores. (b) plot shows the average scores. 45
Figure 5.4 Maximum and average scores for phase one agent on Montezuma's Revenge game 46
Figure 5.5 Maximum and average scores for phase one agent on the Pitfall game 47

Figure 5.6 Maximum and average scores for phase one agent on the Private Eye game 47
Figure 5.7 Maximum and average scores for phase one agent on the Breakout game 48
Figure 5.8 Maximum and average scores for phase one agent on the Pong game 48
Figure 5.9 Comparing phase one performance for both agents. (a) plot shows the average scores. (b) plot shows the maximum scores 50
Figure 5.10 The performance of our agent in phase two. (a) plots show online rewards of Montezuma's Revenge game. (b) plots show online rewards of the Breakout game 52
Figure 5.11 Online rewards for DQfD agent, standard PDDQN agent, and our agent on the Montezuma's Revenge game. We removed the zero reward between the episodes 55
Figure 5.12 Online rewards for DQfD agent, standard PDDQN agent, and our agent on the Breakout game. We removed the zero reward between the episodes 55
Figure 5.13 The ratio of how often the demonstration data was sampled versus the self-generated data in the DQfD algorithm while we trained it . 56

LIST OF TABLES

Table 2.1 Summary of comparison for the Q-learning and Value Iteration methods 10
Table 3.1 List of DQN hyper-parameters and their values 18
Table 4.1 The demonstration actions number, episodes number, total steps, and scores for each game 33
Table 5.1 The parameters used for training and testing the algorithms. 42
Table 5.2 Comparing our agent phase one performance with the baseline performance 51
Table 5.3 Reward Function learning dataset details and test accuracy 53
Table 5.4 Reward Function learning F1 scores for each action 53

ABSTRACT

Overcoming Hard Exploration Problem in Markov Decision Process Using Inverse Reinforcement Learning and Human Demonstrations

Wadhah Zeyad Tareq TAREQ

Department of Computer Engineering
Doctor of Philosophy Thesis

Supervisor: Assoc. Prof. Dr. Mehmet Fatih AMASYALI

In this study, performance enhancement of Deep Reinforcement Learning (DRL) algorithms on hard exploration environments have been proposed by using external reward systems. For this, first, human demonstrations were used as an external reward. A complete training phase (Pre-training phase) includes the human demonstration is performed to imitate the human performance and get the advantage of embedded rewards. After that, an independent Reward Function using Inverse Reinforcement Learning (IRL) principles is established to gain better performance on an additional training phase (Interacting phase).

This thesis proposes to train a notable and simple Deep Reinforcement Learning algorithm using external rewards in the Pre-training phase. During building the human demonstrations dataset, the actions made by the human are tagged as correct, and all other possible actions are tagged as wrong. Prioritized Double Deep Q-Networks (PDDQN) algorithm was chosen to train on this dataset. It has become possible for this trained model to predict correct action in hard exploration environments. In addition, the important samples selection ratio has been optimized to improve the performance of the algorithm. As a result, the PDDQN algorithm outperformed all baseline algorithms in hard exploration environments.

This thesis proposes a reward function for the Interacting phase using the principles of Inverse Reinforcement Learning. It has been determined that the difference between the environmental rewards and external rewards in human demonstrations leads to

a decrease in the efficiency of the algorithm in the interaction phase. In order to solve this problem, an Inverse Reinforcement Learning reward function, which is one of the common techniques developed in recent years, has been created to predict human/expert behaviors by using rewards. The reward function is trained with a set of data extracted from phase one human demonstrations, and then the rewards it predicts can be used in conjunction with environment rewards. With this feature, better performance has been seen in the experiments.

Keywords: Deep reinforcement learning, learning from human demonstration, inverse reinforcement learning, atari games

ÖZET

Markov Karar Sürecinde Zor Keşif Probleminin Ters Pekiştirmeli Öğrenme ve İnsan Gösterimleri Kullanılarak Aşılması

Wadhah Zeyad Tareq TAREQ

Bilgisayar Mühendisliği Anabilim Dalı
Doktora Tezi

Danışman: Doç. Dr. Mehmet Fatih AMASYALI

Bu çalışmada, Derin Pekiştirmeli Öğrenme (Deep Reinforcement Learning, DRL) algoritmalarının zorlu keşif ortamlarında dışsal bir ödül sistemi kullanılarak performanslarının iyileştirilmesi önerilmiştir. Bunun için öncelikle insan gösterimleri dışsal bir ödül olarak kullanılmıştır. Algoritmanın insan performansını taklit etmesi ve dışsal ödüllerden yararlanması için insan gösterimlerini içeren bir eğitim aşamasını (Ön eğitim aşaması) tamamlaması gerekmektedir. Bunun ardından, algoritmanın daha iyi performans elde etmesi için ek bir eğitim aşamasında da (Etkileşim aşaması) Ters Pekiştirmeli Öğrenme (Inverse Reinforcement Learning, IRL) ilkeleri kullanılarak bir ödül fonksiyonu kurulmuştur.

Bu tez, Ön eğitim aşaması için, dışsal ödüller kullanarak basit ve etkili bir Derin Pekiştirmeli Öğrenme algoritması eğitmeyi önermektedir. İnsan gösterimleri içeren eğitim kümesini oluştururken insan tarafından yapılan eylemler doğru diğer tüm olası eylemler yanlış olarak etiketlenmiştir. Bu veri kümesini eğitmek için Öncelikli Çift Derin Q-Ağları (Prioritized Double Deep Q-Networks, PDDQN) algoritması seçilmiştir. Eğitilen bu modelin zorlu keşif ortamlarında doğru eylemi tahmin etmesi mümkün hale gelmiştir. Buna ek olarak, algoritmanın performansını artırmak için önemli görülen örneklerin seçilme oranı optimize edilmiştir. Sonuç olarak, PDDQN algoritması zorlu keşif ortamlarında tüm temel algoritmalardan daha iyi performans göstermiştir.

Bu tez, Etkileşim aşaması için, Ters Pekiştirmeli Öğrenme ilkelerini kullanan bir ödül

fonksiyonu önermektedir. İnsan gösterimlerindeki dışsal ödüllerle ortam ödülleri arasındaki farkın etkileşim aşamasında algoritmanın verimliliğinde bir azalmaya yol açtığı belirlenmiştir. Bu sorunu çözmek için son yıllarda geliştirilen yaygın tekniklerden biri olan Ters Pekiştirmeli Öğrenme ile ödüller kullanarak insan/uzman davranışlarını tahmin eden bir ödül fonksiyonu oluşturulmuştur. Ödül fonksiyonu birinci aşama insan gösterimlerinden çıkarılan veriler kümesiyle eğitilmekte ve daha sonra tahmin ettiği ödüller gerçek ortam ödülleriyle birlikte kullanılabilmektedir. Bu özelliği sayesinde yapılan deneylerde, daha iyi performans sağladığı görülmüştür.

Anahtar Kelimeler: Derin pekiştirmeli öğrenme, insan gösteriden öğrenme, ters pekiştirmeli öğrenme, atari oyunları

1
INTRODUCTION

1.1 Literature Review

Our research is based on Deep Reinforcement Learning (DRL) and Deep Inverse Reinforcement Learning (DIRL) algorithms. In this study, we have modified the classical algorithms for getting higher performance by using human demonstrations. This literature survey is categorized into three groups. The first group is about the appearance and the development of Reinforcement Learning (RL) algorithms in video games. The second group is regarding using human demonstrations in DRL algorithms. The last group is about Inverse Reinforcement Learning (IRL) and DIRL.

1.1.1 Deep Reinforcement Learning

RL allows agents to explore the environment and learn from trial and error without the need for the demonstrator. Many RL agents successfully built to learn how to play Atari games. The first deep RL agent was introduced by [1]. The agent learned control policies to play six different games from game frames and scores only. After that, [2] introduced a notable RL agent, known as Deep Q-Network (DQN), that can play 49 different Atari games. DQN merges the Q-learning [3] with the Convolutional Neural Network (CNN). The result showed that the DQN agent was able to reach the expert performance on many games. Two main components enabled the agent to perform well: the target network and the experience replay.

Later, the RL community has made many changes and extensions to the DQN algorithm to improve its performance and stability. Using the Double Q-learning algorithm [4] rather than the Q-learning algorithm was the first change to the DQN. The Double DQN (DDQN) [5] showed that the DQN suffers from significant overestimation in some games, and the Double Q-learning reduces that overestimation and leads to better performance. Prioritized experience replay (PER) [6] replaced random sampling with sampling experience with probability. This change allowed the DQN to sample experiences that have more information to learn than the other. The

Dueling Double Deep Q-network (DDDQN) [7] architecture suggests decoupling the double Q-learning into two estimators: state and action. The state estimator tells us it is (or is not) a valuable state regardless of the effect of the actions at that state. Asynchronous Advantage Actorcritic (A3C) [8] presented asynchronous four standard reinforcement learning algorithms and showed that parallel actor-learners stabilize training, allowing all four methods to train neural network controllers successfully. Distributional Q-learning [9] learns a categorical distribution of discounted returns instead of estimating the mean. Noisy DQN [10] uses stochastic network layers for exploration. Finally, Rainbow [11] studied the efficiency of the six previous extensions in one algorithm. Their algorithm provided an agent with performance better than each extension separately.

1.1.2 Learning from Demonstrations

Despite these achievements and enhancements, the RL agent performs poorly in environments where the feedback is unavailable or intermittent. Generally, the RL agent learns good policies in many games where there is feedback from the environment. Many environments do not come with such features. Montezuma's Revenge and Pitfall are good examples of such environments. In these games, the feedback comes after performing a sequence of actions. For covering this gab, many works suggested using human demonstrations in the learning. Human demonstrations provide many other advantages like reducing the training time and solving problems of the environment that do not come with an accurate simulator.

Deep Q from Demonstrations (DQfD) [12] introduced a DRL agent that uses human demonstrations to learn to play Atari games. Their algorithm accelerated the learning process and automatically assessed the necessary ratio of demonstration data. Also, DQfD achieved acceptable results in sparse rewards environments. Another study [13] trained an agent to accomplish the same states seen in a YouTube video of the game, where [14] combines a sophisticated version of Q-learning to maximize the likelihood of actions taken in a demonstration. Learning from Single Demonstration (LfSD) [15] also proposed using the human demonstration to cover the sparse rewards problem in Montezuma's Revenge games by starting their agent from demonstration states rather than from the game standard start states. This process aims to break down the complex exploration problem into a curriculum of sub tasks. The Go-explore [16] agent played well in Montezuma's Revenge and Pitfall games by remembering the state it visited. The agent is starting the exploration from a new promising state. After that, it is building the policy by imitation learning.

1.1.3 Inverse Reinforcement Learning

The IRL first appeared to solve the problem of the new state in the Imitation Learning (IL) approaches. In IL, when the agent visited a new state, it became impossible to return to the demonstrated states. IRL was first introduced by Ng and Russell [17]. They proposed to solve the reward engineering problem by inferring the reward function from the expert demonstrations. The main challenge was the ambiguity. In some demonstrations, many reward functions could produce the expert policy. In [18], they developed an approach that can get the same reward as the expert policy rewards. The later works are Bayesian IRL and Maximum Entropy IRL. Bayesian IRL [19] adopted the ambiguity by calculating the distributed rewards rather than focusing on the given function. This method produces the same guarantee as in [18]. On the other hand, Maximum Entropy IRL [20] builds a reward function that meets the expert features. This function guarantees the higher-entropy stochastic policy. This allowed increasing the generalization in the environments using many different planning dynamics.

As we mentioned before, the researchers developed a few IRL algorithms to play video games. The work by Uchibe [21] used logistic regression to classify the transitions into two groups: expert and non-expert. The classifier is used instead of the reward function to train standard deep RL algorithms. The results showed that their performance rarely outperforms the IL. The second work [22] which named CNN-AIRL. They modified the Adversarial IRL algorithm [23]. Their modification includes adding CNN to the AIRL baseline, normalizing the environment reward, and increasing the size of the discriminator data set. Additionally, they represented the state with low-dimensional by autoencoder architecture built especially for video games. Their algorithm achieves good performance on the simple Catcher video game while lower than the expert performance on the Atari game.

1.2 Objective of the Thesis

High-dimensional inputs and scale of rewards represent a challenge for DRL algorithms. DQN overcame these challenges by learning directly from the raw pixel and clipped game score. However, the DQN agent failed in many environments due to the distribution of rewards. Such environments are known as hard-discovering environments. Montezuma's Revenge, Private Eye, and Pitfall are popular examples of these environments. For example, Montezuma's Revenge requires hundreds of actions to produce a reward. As a result, random action (DQN working method) makes a reward every half a million steps. This scaling limits the abilities of the current DRL techniques. In this thesis, we have aimed to enable DRL algorithms to perform

well in hard exploration games by using external rewards involved within the human demonstrations to be available at all times during the learning.

1.3 Hypothesis

To enable classical DRL algorithms to play hard exploration games, we assume it is better to reward each action selected by the algorithms. We adoptive the Prioritized Double DQN (PDDQN) algorithm and trained it with three steps to prove the hypothesis. Firstly, building human demonstrations with an external reward to supply each action with a reward. Secondly, train PDDQN with these demonstrations to measure the effect of these external rewards. Thirdly, using IRL to enable PDDQN to learn from self-generated data by interacting with the environment. These hypotheses can be clarified with the supports of the detailed information given below.

Human demonstrations are the primary key used with DRL algorithms to enable them to play hard exploration games. The demonstration is built by human and consist of the current state of the game, the action (selected by a human), game feedback (reward), and the next state of the game. In this study, the main difference in the demonstration is the reward. Rather than using the game rewards, the human sets the correct action reward to a constant $r = 1$ and sets rewards to a constant $r = 0$ to all other actions in the action space.

The PDDQN is a combination of DQN [2], DDQN [5], and PER [6]. PDDQN provides good performance on games where rewards are available. In hard exploration games, this algorithm suffers from the disappearance and absence of rewards. In contrast to the traditional way, the demonstration with an external reward covers the problem of reward disappearance and lack. This proposed demonstration makes hard exploration games possible for the PDDQN and any DRL algorithm.

As famous, agents interacting with the game environment are essential for measuring the performance and enhancing the RL algorithms. The difference in the distribution of rewards between the demonstration and the game made interacting with the environment impossible. This study proposed to use IRL to generate reward function from the demonstration. This reward function makes interacting with the game environment possible and enables the PDDQN algorithm to continue learning and enhancement.

2
BACKGROUND

This chapter presents the background for the contents in the following chapters and a set of definitions for some essential topics. The chapter starts with a summary of machine learning, which is the global context of the study in this thesis. The difference between machine learning subfields. Then, the Markov Decision Process (MDP) architecture and details. Finally, the fundamentals of Q-learning and general information about the Convolutional Neural Networks (CNN).

2.1 Machine Learning

Machine learning is concerned with creating applications that automatically improve with experience. Recently, machine learning engineering developed several applications starting from data-mining applications that learn to detect fraudulent bank transactions to learn user's preferences by information-filtering systems. The field has also seen significant progress in autonomous vehicles that can learn to drive in cities and highways. All these applications resulted from the significant advances in algorithms that form the foundations of this field [24]. There are three popular sub-fields of machine learning: Supervised Learning, Unsupervised Learning, and Reinforcement Learning [25].

2.1.1 Supervised Learning

Supervised learning is the most well-known machine learning sub-field. Supervised learning seeks to build an approach that finds appropriate output according to some input depending on training data. There are many examples of supervised learning applications:

- Text classification
- Image classification and object location

- Regression problems
- Sentiment analysis

Despite the difference between these applications, but the same mechanism can solve them. The supervisor has provided the application with labeled examples, and the application learns from these examples. Many examples represent the input, and the application generates an output depending on the given output. That is why it is called supervised learning.

2.1.2 Unsupervised Learning

In unsupervised learning, the data has no labels assigned before. The algorithm aims to learn the hidden structure of the examples. Data clustering is a common example of unsupervised learning approaches. The algorithm finds the relationships among particular examples and combines these examples into a set of clusters.

Generative Adversarial Networks (GANs) [26] is another popular unsupervised learning method. GANs is a framework that uses the adversarial process for computing the generative models. The idea of GANs is to generate two models. The first model generates fake data, and the second model tries to distinguish the artificially generated data from the original data in the data set. Over time, the two models are becoming more qualified by acquiring more experience in their tasks by capturing particular patterns of the dataset.

2.1.3 Reinforcement Learning

RL is the third subfield and lies between supervision and unsupervised learning methods. It doesn't work with labeled examples as in supervised learning. However, it is not completely blind as in unsupervised learning due to the reward system. RL uses many supervised learning methods, such as function approximation with deep neural networks, backpropagation, and stochastic gradient descent, to learn data representation. Good RL examples are an automatic helicopter trying to make a roll, news recommendation to users, and an algorithm learning playing chess game to defeat a professional human player. There are two primary entities in RL: Agent and Environment. Also, there are communication channels between these entities: Actions, Rewards, and Observations, as is seen in figure 2.1. In the next few paragraphs, we will explain these entities in detail.

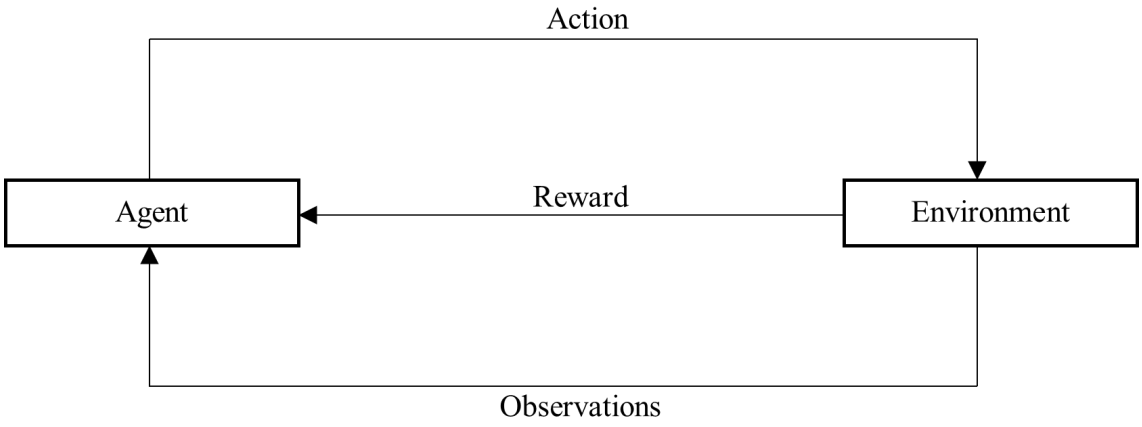


Figure 2.1 Reinforcement learning entities and their communications

The **Environment** is everything outside of an agent. The agent interacts with its environment by actions only, and these actions cannot change the dynamics or rules of the environment. An **Agent** is software written to solve a problem without any prior knowledge. The agent selects action and performs that action in the environment to receive rewards to evaluate its performance. The agent could be an algorithm trying to learn to play a chess game or an extension software in the web that tells the browser where to drag the mouse cursor, where to click, and what to write in each text-box.

Actions are movements or capabilities that the agent can do in the environment. Actions are pre-established rules that the agent cannot change over the running time. There are two types of actions: discrete and continuous. Discrete actions are a finite set of independent things with predefined values, such as moving one step for left or right in a video game. Continuous actions have a range of values attached to a finite set of actions. A car's actions, such as the steering wheel and gas pedal, is an example of continuous actions.

The **Reward** is a value the agent obtains periodically from the environment. It's a number that could be small or large and positive or negative. The purpose of reward is to determine the efficiency of the agent's behavior. The availability of rewards is different from the environment to environment. It's common to receive a reward for every interaction. Nevertheless, in some environments, a reward can appear once in an agent's lifetime. In this case, all rewards except the last one will be zero.

Observation of the environment is the second source of information that the agent can get. The observations tell the agent the changes that occurred in the background, and the agent uses them to select the next action. In the chess game, observation is the new position of each pawn on the game board after executing one step.

2.2 Markov Decision Process (MDP)

Markov Decision Process (MDPs) [27] are a standard formalization for describing sequential decision-making problems. MDPs are meant to be a straightforward framing of learning from interaction to achieve a goal. For this reason, MDPs become a mathematically idealized form of the reinforcement learning problem for precise theoretical statements. In MDPs, an agent and environment interact at a sequence of discrete-time steps $t = 0, 1, 2, 3, \ldots$ As it is seen in Figure 2.2, at each time step t, the agent obtains a state $s_t \in S$, which is a representation of environment's situation, and depending on this situation, the agent chooses an action, $a_t \in A(s)$. After that, and depending on the agent's action, the environment feedback the agent with a reward which is a numerical value, $r_{t+1} \in R$, and move the agent to a new state, s_{t+1}. The environment and agent in conjunction shape a sequence or circuit that forms such as this:

$$s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, \ldots \tag{2.1}$$

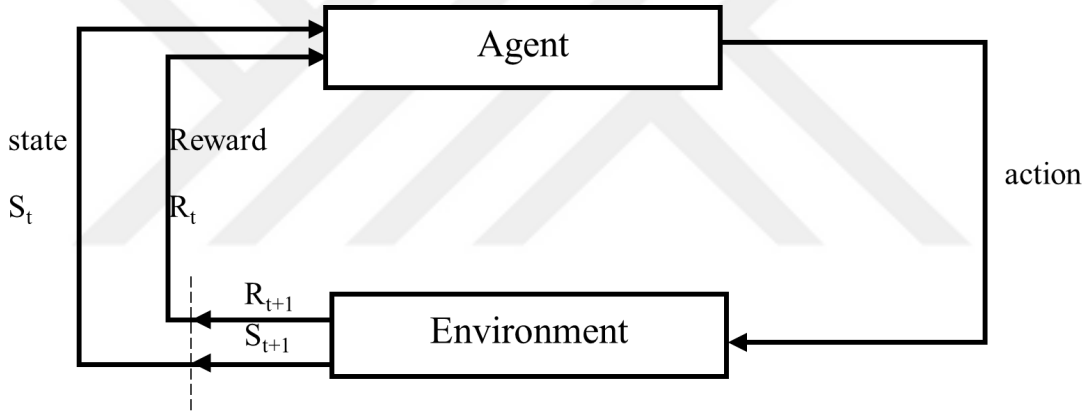


Figure 2.2 The agent–environment interaction in a Markov decision process [27]

The chosen action determines the probability of the new state s_{t+1}. In math, this probability represents by the state transition function $T(s_{t+1}|s, a)$. This function satisfies the Markov Property. Markov Property proposes that the current state is the only parameter to determine the future dynamics of the system. That means each state in the system must be distinguishable and unique. The agent's main problem in MDPs is finding the function that defines which action will be chosen in the current state. This function is known as the policy, and it represents by $\pi(s)$. The agent aims to find the optimal policy $\pi^*(s)$ that will maximize the cumulative reward [25] [28]:

$$R = \sum_{t=0}^{\infty} \gamma_t r(s, a, s_{t+1}) \tag{2.2}$$

The γ is known as a discount factor and takes values starting from 0 to 1. The main

purpose of γ is to reduce the effects of late rewards.

Finding optimal policy achieving through finding the optimal value function. The procedure of finding the optimal value function is by starting in a state and calculating the future predicted discounted reward. Richard Bellman [29] has introduced a dynamic programming algorithm known as the Value Iteration (VI) to find the optimal value function. The prime concept of the VI algorithm is the Bellman equation. The Bellman equation says that the optimal value is the value of action with the maximum discounted future reward (the action with maximum Q-value). The Q-value for a pair of the current state and action is the expected immediate reward summed with discounted future reward. The formulas of VI and Q-value are:

$$V(s) = max_a Q(s,a) \tag{2.3}$$

$$Q(s,a) = \sum_{s_{t+1}} T(s_{t+1}|s,a)[R(s,a,s_{t+1}) + \gamma V(s_{t+1})] \tag{2.4}$$

2.3 Q-Learning

Q-Learning [30] is a learning algorithm that suggests determining the behavior from direct experience with the environment. A learning algorithm generally applies when the learner does not have prior knowledge about the problem or the environment. As the name suggests, Q-learning estimates the optimal Q-values of an MDP, which means that behavior can be learned by taking actions greedily with respect to the learned Q-values. In the Q-learning algorithm, the most common way to choose an action in the current state (s) is to use a greedy policy. ϵ is a fraction between 0 and 1. Based on the policy, the agent randomly selects a fraction of time among all actions, whereas the action with respect to the Q-value estimates a fraction of $(1-\epsilon)$ time. The update rule for Q-learning is below [28]:

$$Q(s,a) := Q(s,a) + \alpha[R(s,a,s_{t+1}) + \gamma max_{a_{t+1}} Q(s_{t+1},a_{t+1}) - Q(s,a)] \tag{2.5}$$

The Q-value is updated by the last state-action pair (s_{t+1}, a_{t+1}) concerning the next state s_{t+1} and environmental reward $R(s,a,s_{t+1})$. The α is the learning rate and takes values arranged from 0 to 1.

Q-learning differs from the Value Iteration by calculating the Q-value. In Value Iteration, we have the reward and the transition probabilities for every action in the state, and we estimate the Q-value depending on these inputs. In Q-learning, the reward and the transition probabilities are unavailable. The agent must discover the

reward by selecting an action and observing the reward and the next state. To reach all possible states, the agent keeps trying by selecting random actions on the current state. Over time, the agent starts to move close to the true Q-value. The comparison between Q-learning and Value Iteration algorithm is in Table 2.1.

Table 2.1 Summary of comparison for the Q-learning and Value Iteration methods

Parameter	Value Iteration	Q-Learning
T matrix	■	□
R matrix	■	■
γ	■	■
α	□	■
ϵ	■	■
ϵ_{decay}	□	■
$iter_{max}$	■	■
Inner loop operator	Bellman	Max
Stop condition	$iter_{max}$	$iter_{max}$
Convergence	■	□

It is usual to use a function approximator to determine the Q-value. The old reinforcement learning algorithms rely on handcrafted features, which limits the application scope. Using deep learning algorithms, especially convolutional neural networks (CNN), enabled reinforcement learning to benefit from the raw high dimensional inputs like visual and auditory information. The CNN details will explain in the next section.

2.4 Convolutional Neural Networks

Convolutional networks [31], also known as convolutional neural networks, or CNNs, are specialized neural networks for processing data with a known grid-like topology. Examples include time-series data, which can be thought of as a 1-D grid taking samples at regular time intervals, and image data, which can be considered a 2-D grid of pixels. Convolutional networks have been tremendously successful in practical applications. The name "convolutional neural network" indicates that the network employs a mathematical operation called convolution. Convolution is a specialized kind of linear operation. Convolutional networks are neural networks that use convolution in place of general matrix multiplication in at least one of their layers [32]. Figure 2.3 shows the typical architecture of CNN. Generally, there are five kinds of layers in CNN: Input Layer, Convolutional Layer, Pooling Layer, Fully-Connected Layer, and Output Layer [28].

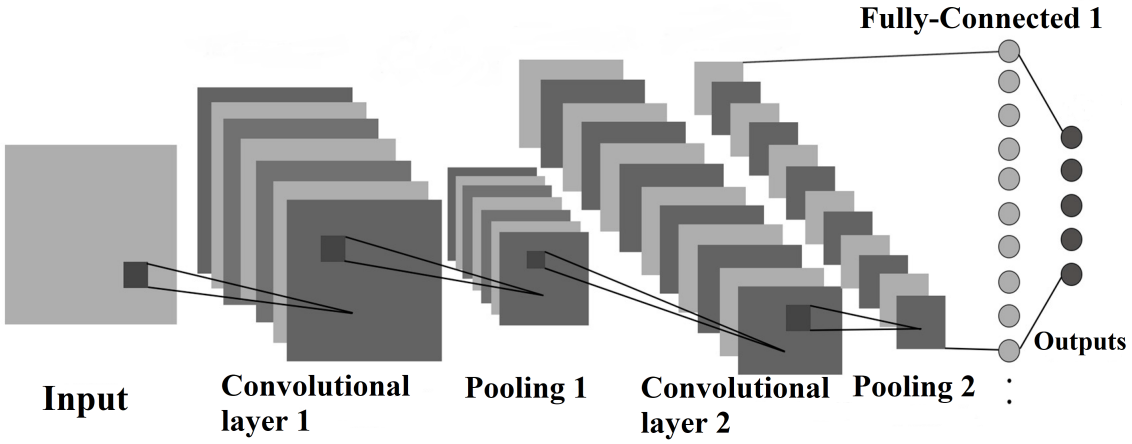


Figure 2.3 The Architecture of Convolutional Neural Networks [33]

2.4.1 Input layer

The input layer is the first layer in CNN. The input layer reserves the image as input with a W x H x C structure. The W is the width, and H is the height of the image. The C character represents the number of channels in the image. For example, the RGB image has three channels Red, Green, and Blue.

2.4.2 Convolutional Layer

The convolutional layer is one of the core layers in CNN. CNN is designed to capture the invariant features of images. Capturing the feature is done by the dot operation between the local region and weights, followed by the pooling method. The convolutional layer receives the image with size W x H x C. It is calculated the output neurons attached to the receptive fields. The output size is calculated by a quad (K, F, S, ZP), where K is the number of filters, F is the filter size, S is the stride number, and ZP is the size of zero-padding separately. The size of the output could be found by:

$$W^{'} = \frac{W - F + 2P}{S + 1} \tag{2.6}$$

$$H^{'} = \frac{H - F + 2P}{S + 1} \tag{2.7}$$

$$C^{'} = K \tag{2.8}$$

Based on the sharing policy between the weights, each down map has weights with a size of F x F x C. Thus, it will amount to F x F x C x K weights and K bias to be trained in the convolutional layer.

2.4.3 Pooling Layer

The pooling layer is the second core layer in the CNN. This layer has many different functions, such as max pooling, average pooling, and L2-norm pooling. These functions perform a non-linear down-sampling operation over the entire feature maps. The most simple and performance process is the max pooling. For these reasons, it becomes the most used function in deep learning approaches. The pooling layer receives input with W x H x C size, perform a down-sampling, and the size of the output could be found by:

$$W^{'} = \frac{W - F}{S + 1} \tag{2.9}$$

$$H^{'} = \frac{H - F}{S + 1} \tag{2.10}$$

$$C^{'} = C \tag{2.11}$$

The feature maps remain constant; thus, it is worth nothing to use the zero-padding technique. The pooling process reduces the size of parameters which reduces the computation time of the next processes. Figure 2.4 shows the architecture of the Convolutional and pooling layers.

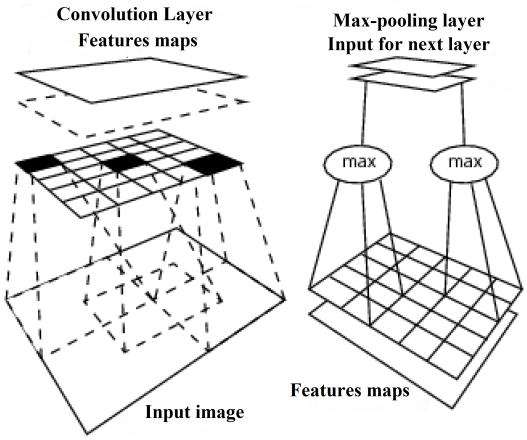


Figure 2.4 The structure of convolutional and pooling layers [34]

2.4.4 Fully-connected Layer

The last layer in CNN is the fully-connected layer. Each neuron is a fully-connected to all neurons in the previous layer. The fully-connected layer computes the final class scores where each class corresponds to a label in the training data set.

3
DEEP REINFORCEMENT LEARNING AND INVERSE REINFORCEMENT LEARNING

The deep word in deep reinforcement learning (DRL) refers to combining reinforcement learning (RL) with deep learning (DL) [35]. The benefit of including DL with RL is allowing the agent to make decisions in environments with unstructured data. DRL algorithms can take in massive inputs and decide what actions to perform to optimize an objective. Therefore, DRL agents have been used in a new set of applications such as Video Games, Natural Language Processing (NLP), Robotics, Education, and Healthcare.

Recently, video games have become a platform for evaluating DRL agents. In video games, the pixels of the game screen is the unstructured input data, and the agent's main objective is to maximize the game score. Arcade Learning Environment (ALE) [36] provided hundreds of Atari 2600 game environments to be research challenges for model imitation learning, and reinforcement learning. The essential features of ALE are differences, interests, and challenges in each environment. Furthermore, ALE provides a standard framework for testing and comparing different learning approaches.

This chapter introduces the details of the powerful algorithms that used DRL in Atari games. Moreover, we present basic information about the Inverse Reinforcement learning methods. We provide all these details because we use some of these algorithms in our proposed system.

3.1 Deep Q-Networks

DQN [2] is an artificial agent trained by Google DeepMind using reinforcement learning principles to learn decision-making from high-dimensional information. They used the advances in DL [37] [38] [39] [40] to develop the DQN agent and tested it on Atari 2600 games. By receiving the game score and screenshots, the agent's

performance exceeds all previous algorithms. Furthermore, it achieves an equivalent level to a professional human player in many games using the same algorithm, hyper-parameters, and network architecture.

The Breakout (Figure 3.1, left) is an example of the Atari 2600 games. The agent aims to get the maximum reward by shooting the ball to knock the bricks and moving the paddle to catch the ball and avoid life loss. This is a standard RL problem (Figure 3.1, right), which trains an agent to communicate with the game and achieve maximum reward by following specific policies. Through repeated training, the agent becomes more intelligent to interact with the environment to achieve high rewards [41]. The working procedure of the DQN agent is explained in the following few subsections in detail.

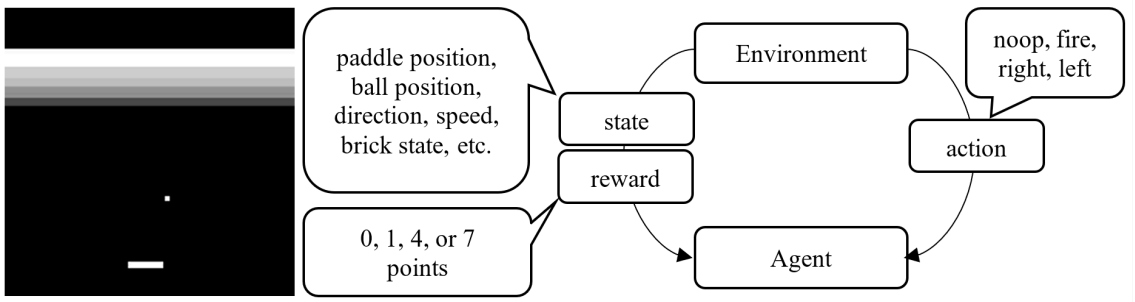


Figure 3.1 The Breakout game in the Markov decision process form [41]

3.1.1 Preprocessing

The first step in DQN is processing the images coming from the environment. The directly raw image of atari games has 210 x 160 pixels with a 128-color palette. This amount of data is required a large storage capacity and long computing time. First, The DQN takes the maximum pixel color value over the previous frame and the present frame to encode each frame. The encoding is essential to remove the flickering in video games because some game objects are seen in some frames while different objects are seen in other frames. Second, the DQN removes the luminance which is the Y channel in the RGB frame, and resizes the frame into 84x84. These steps are applying to m of frames (most recent frames) and stack them as one input to the Q-function. The preprocessing function responsible for these steps is represented by the symbol Φ in the algorithm1 described below.

The DQN used four-game frames and stacked them together as a sequence. This history sequence enables the agent to understand both the direction and acceleration of the moving objects. For example, in the Breakout game, it is impossible to determine the ball is moving up or down from one frame. But with the use of history sequence,

the action can be different from one case to another. If the ball moves down, it is better to move the paddle to bounce the ball back up. If the ball is moving up, the agent can wait a little longer or try to move in the opposite direction where the ball will eventually reach there, see Figure 3.2. The DQN algorithm is robust to different values of the history sequence. For example, it works with a series of 3 or 5 frames.



Figure 3.2 A fully preprocessed state for breakout.

3.1.2 Model architecture

The DQN used a deep convolutional neural network (CNN) to approximate the optimal action-value function. Their architecture is shown in Figure 3.3. The input to the neural network is the state representation only with an 84 x 84 x 4 image produced by the pre-processing function. The output of the CNN is 18 separate units. Each unit represents a possible action. The number of valid actions varied between 4 and 18 depending on the game's action space. The first hidden layer convolves 32 filters of 8x8 with stride 4 with the input image and applies a rectifier non-linearity [42] [43]. The second hidden layer convolves 64 filters of 4x4 with stride 2, followed by a rectifier non-linearity. This is followed by the third convolutional layer that convolved 64 filters of 3 x 3 with stride 1 followed by a rectifier. The final hidden layer is fully-connected and consists of 512 rectifier units. The output layer is a fully-connected linear layer with a single output for each valid action. Each output is responsible for predicting the Q-values of a single action from the game actions. This architecture allows calculating Q values for all available actions in the current state with a single network forward pass.

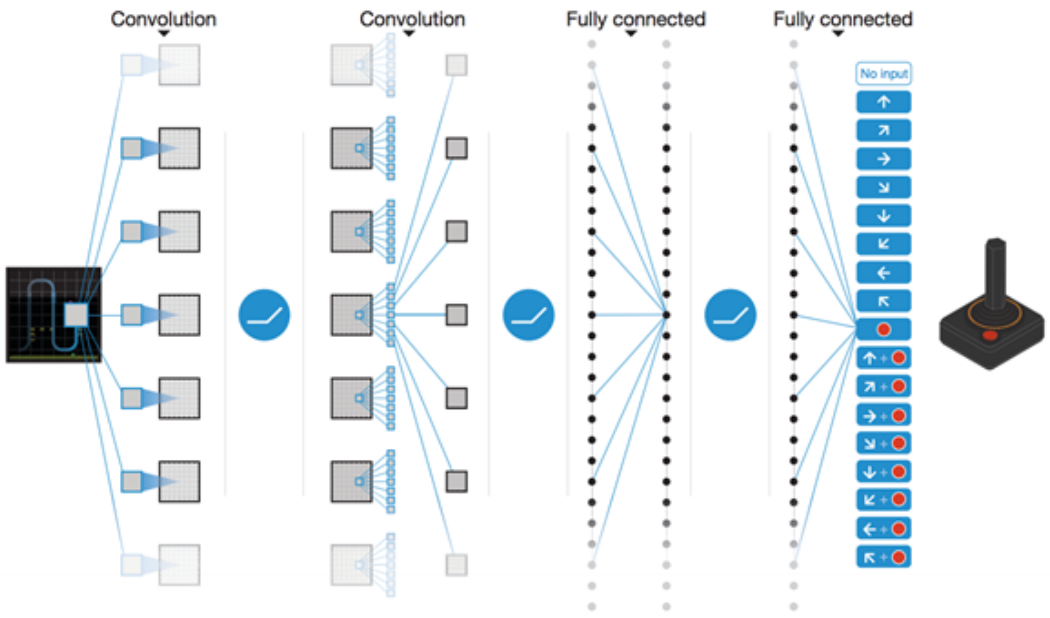


Figure 3.3 Schematic illustration of the convolutional neural network used in DQN [2]

3.1.3 Learning by interacting

First, the agent needs to interact with the environment to generate data to train on. The agent can act randomly, but that requires an extremely long time. The alternative way is using the Q function approximation as a source of behavior. At the beginning of the training, the representation of the Q is not perfect and will stuck the agent with wrong actions without trying to behave differently. The best way to balance the randomness and the Q function is by using exploration and exploitation. In the beginning, the agent needs to explore the transition and the action outcomes, so it is better to use the random behavior. After training progresses, random behavior becomes time-wasting, and it is better to use the Q approximation to interact with the environment efficiently.

This behavior is known as an epsilon-greedy or $(\epsilon - greedy)$ method that uses probability hyper-parameter ϵ to control the switching between Q policy and random. The DQN agent starts with $\epsilon = 1.0$, which means only random actions are selected. This value slowly decreases through the training to become $\epsilon = 0.1$. The epsilon-greedy method helps the agent explore the environment initially and select good actions at the end of the training. DQN uses a technique known as experience replay [44] to store the agent's experiences. At each time step t, the agent acts by using the e-greedy method. The agent stores the transition experience $e_t = (s_t; a_t; r_t; s_{t+1})$ in the replay memory $D_t = e_1; e_2; ...; e_t$.

3.1.4 Training details

As it's like in supervised learning, DQN calculates the loss function by finding the difference between the target value and the current prediction value (CNN output). The core of reinforcement learning is that we don't have the real target value. The target value can be estimated using the Bellman equation which the target is the current reward plus the discounted highest Q value for the next state.

$$Q(s,a) = r(s,a) + \gamma max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) \tag{3.1}$$

So The loss function changes at each iteration i:

$$L_i(\theta_i) = E[(r(s,a) + \gamma max_{a_{t+1}} Q(s_{t+1}, a_{t+1}; \theta_i) - Q(s,a;\theta_i))^2] \tag{3.2}$$

The problem is using the same weights for estimating the target and the current prediction of the Q value. This led to a significant correlation between the target and the weights we are changing. It means that both the Q value and the target value are shifting at every step of training. So, the agent is getting closer to the target, but the target is also moving. To make training more stable, DQN proposed to use the target network. The target network weights are copied from the current network and calculate the target Q value in the Bellman equation. The target network weights are copied once in N steps. DQN copied the weights after each 10 thousand of training (N = 10000):

$$L_i(\theta_i) = E[(r(s,a) + \gamma max_{a_{t+1}} Q(s_{t+1}, a_{t+1}; \theta_i^-) - Q(s,a;\theta_i))^2] \tag{3.3}$$

Where θ is the weight of the current CNN, θ^- is the weights of the target network. DQN trained their agent on 49 different Atari games. For all games, they used the same network architecture, hyper-parameters, and learning algorithm. All parameters are shown in Table 3.1. During the training only, DQN changed all positive rewards to 1, negative reward to -1, and kept the 0 rewards unchanged. DQN changed the rewards to limit the error derivatives' scale and make it easier to use the same learning rate across multiple games. The life counter (number of lives in the game) is used to determine the end of the episode during the training. In all these experiments, DQN used the Root Mean Square Propagation algorithm (RMSProp) [45] with mini-batches of size 32. The complete algorithm for training deep Q-networks is presented in Algorithm 1.

Table 3.1 List of DQN hyper-parameters and their values

Hyper-parameter	Value
Reply memory size	1000000
Target network update frequency (N)	10000
Discount factor (θ)	0.99
Learning rate	0.0025
Gradient momentum	0.95
Squared gradient momentum	0.95
Min squared gradient	0.01
Initial exploration	1
Final exploration	0.1
Final exploration steps	1000000

Algorithm 1 DQN

Set replay buffer D with N capacity
Initialize Q function with random weights θ
Initialize target Q function with weights $\theta^- = \theta$
for episode = 1, M **do**
    Set $s_1 = x_1$ and preprocessed $\phi_1 = \phi(s)$
    for t = 1, T **do**
        With ϵ select action randomly a_t
        otherwise select action a_t using trained policy
        Execute action a_t in emulator and observe reward r_t and image x_{t+1}
        Set $s_{t+1} = s_t$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
        Set $y_i = r_i + \gamma max_{a_{t+1}} Q(\phi_{j+1}, a_{t+1}; \theta^-)$
        Perform a gradient descent step on $(yi - Q(\theta_j, a_j, \theta))^2$ with respect to the network parameters θ
        Every 10000 steps reset $\theta^- = \theta$
    end for
end for

3.2 Deep Q-network Extensions

DQN algorithm had a notable effect on the Reinforcement Learning (RL) field by using a nonlinear approximator which provided many interests in the deep Q-learning field in particular and in deep RL overall. Since then, the RL community has made many changes and extensions to the DQN basic architecture to improve stability and convergence [25]. In the following few subsections, we'll take a deeper look at some of those ideas.

3.2.1 Double Q-Learning

The first improvement to the basic DQN came from DeepMind researchers in a paper titled Deep Reinforcement Learning with Double Q-learning [5]. The authors showed that the standard DQN suffers from overestimating the Q values. This overestimation is harmful to the agent's performance and leads to sub optimal policies. The reason for the overestimation is the max operation in the Bellman equation. The Double Q-Learning solved this problem by modifying the Bellman equation by choosing actions for the next state using the trained network but taking values of Q from the target network. So, the new expression for target Q-values will look like this:

$$Q(s,a) = r(s,a) + \gamma Q(s_{t+1}, argmax_a Q(s_{t+1}, a; \theta_i); \theta_i^-) \tag{3.4}$$

Double DQN finds better policies, obtaining new state-of-art results on the Atari 2600 domain.

3.2.2 Prioritized Experience Replay

The next idea to improve DQN was the Prioritized Experience Replay (PER) [6]. This method aimed to enhancement the sampling efficiency in the replay buffer. The standard DQN sampled the experiences from the replay buffer randomly to get the next training batch. The PER replaced this random sample policy by assigning priorities to each sample in the buffer. The priority value is the difference between the current network prediction and the target output for the same sample (Temporal Difference TD error). To find the priority of transition i:

$$p_i = |\sigma| + e \tag{3.5}$$

Where e is a small positive constant that assumes that no transition has zero probability to be taken, σ is the TD error for that transition. However, Setting the priority using TD-error has several issues. First, the TD errors are updated for the transitions that are replayed. If the transition has a low TD error at the beginning of training may not be selected for a long time. Secondly, the TD error shrinks gradually, so the transitions with high TD error often get chosen. These issues make the system prone to over-fitting. To defeat these issues, PER proposed to use a stochastic sampling method that includes both priority and random sampling. Concretely, the probability

of sampling transition i can be defined as:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{3.6}$$

where p_i is the priority of transition i. The exponent α is a hyper-parameter used to reintroduce some randomness in the transition selection for the training batch. If $\alpha = 0$, all transitions in the training batch will select randomly, and if $\alpha = 1$, only the transitions with the highest priorities will be chosen. Prioritized Double DQN (PDDQN) combined the reinforcement learning approach (represented by the Double DQN algorithm) with the prioritized replay mechanism. The standard reinforcement learning approaches sample data randomly while the new prioritized approaches sample data with prioritization.

3.2.3 Dueling Network Architectures

The next improvement to the standard DQN was proposed by [7] named Dueling Network Architectures for Deep Reinforcement Learning. The main idea of Dueling architecture is dividing the Q-values Q(s, a) into quantities: the value of being at that state V(s) and the advantage of taking that action at that state A(s, a):

$$Q(s,a) = A(s,a) + V(s) \tag{3.7}$$

After calculating these two quantities, the new Q value comes from combining them through an aggregation layer. Figure 3.4 compares the basic DQN with the Dueling architecture.

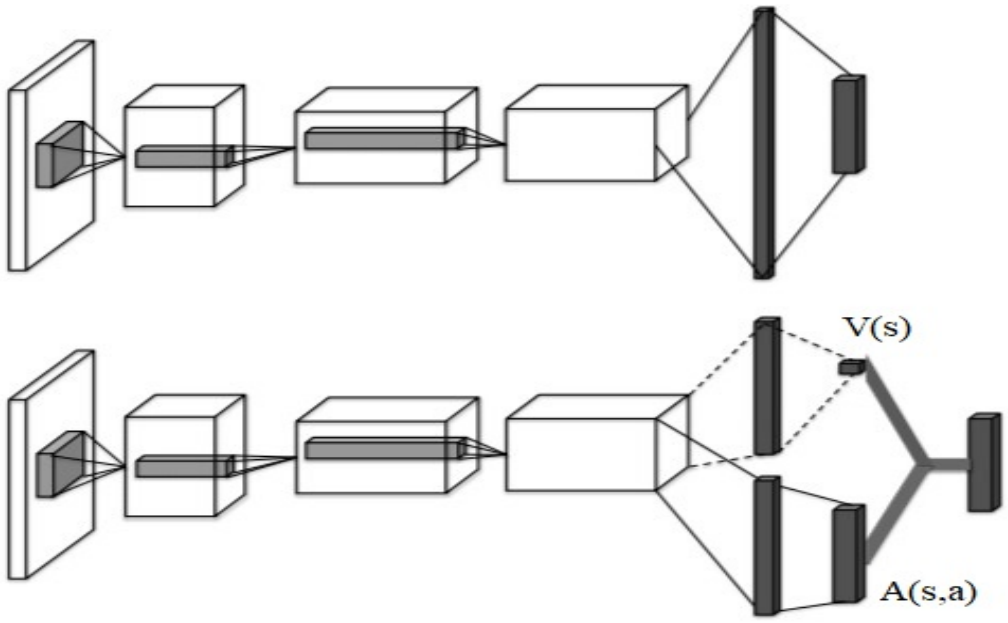


Figure 3.4 Basic (top) and dueling (bottom) DQN architecture [7]

The DQN computes the Q value for every action in the current state. Sometimes the state is not useful and there is no advantage from calculating the Q value for such state actions. For example, in video games, some states contain actions that lead to termination only. The dueling architecture decouples the estimation to discover such types of states without the need to calculate the Q value for each action in that state. As a result, the Dueling architecture is helpful for states when the actions have no impact on the environment. One popular example is the car racing game. When the way is empty, the actions responsible for moving left or right do not matter. In such a case, the dueling DQN does not calculate the Q value of all actions, just calculates the Q value of the meaningful actions.

3.2.4 Rainbow

Rainbow [11] examines six improvements to the standard DQN algorithm (some of them explained before). These improvements are Double DQN [5], Prioritized experience replay [6], Dueling network architecture [7], A3C [8], Distributional Q-learning [9], and Noisy DQN [10]. Rainbow aims to study the effect of combining these improvements. The new agent is a single integrated agent with all mentioned improvements named Rainbow. Combining all of these improvements in one hybrid method occur in three steps: First, the network architecture: the new architecture used the categorical DQN to predict the value probability distribution of action. Also, the network will have two separate paths for values and advantage distributions as

in Dueling architecture and replace the last layers in the DQN network with the NoisyLinear layer as in the Noisy network. Second, they used the prioritized replay buffer to sample transitions by priority rather than random. Finally, the Bellman equation disseminates to several steps as in the n-steps and uses the double DQN to limit the overestimation of Q values. The Rainbow agent performs better than all previous separate improvements in terms of data efficiency and final performance.

3.3 Deep Q-learning from Demonstrations

Despite the apparent successes of deep reinforcement learning, it suffers in environments where rewards are missing or rare. Such environments are known as Hard Exploration environments in which the reward distribution is irregular. The first algorithm that was able to solve the problem of the uneven distribution of rewards is Deep Q-learning from Demonstrations (DQfD) [12]. The DQfD used human demonstrations to accelerate the learning process and enable the agent to play the hard discovering environments such as Montezuma's Revenge, Private Eye, and Pitfall for the first time. The DQfD consists of two phases, the first is the pre-training phase, and the second is the interaction phase. We will explain these phases in detail.

3.3.1 The Pre-training Phase

The pre-training phase aims to imitate the demonstrator by training the agent with samples from the demonstration data. In this phase, the agent does not interact with the environment, which means that it does not receive the current state, does not choose an action, does not receive a reward, and does not move to the next state. All this information is present as a sample in the demonstration data. The mini-batches samples select from these demonstrations data to update the weights of the agent's network to satisfy the Bellman equation (target value). The agent uses many losses functions to update the network weights and biases. These functions are the double Q learning with one step ($J_{DQ}(Q)$), the double Q-learning with n-step ($J_n(Q)$), the supervised large margin ($J_E(Q)$), and the L2 regularization ($J_{L2}(Q)$).

Double Q learning with one step guarantees that the agent's network fulfills the Bellman equation. Double Q-learning with n-step (n = 10) verify reliable pre-training by reproducing the demonstrations values of all previous states. The third loss function [46] is used to make the values of the other actions be at most a margin lower than the value of the expert's action. Using the supervised large margin classification loss grounds the values of the unselect actions to logical values and makes the greedy policy induced by the value function imitate the demonstrator. Finally, the L2 regularization

loss is implemented to control the over-fitting of the weights on demonstration data with a small size. The functions loss that applied by the DQfD to update the weights are shown below:

$$J_{DQ}(Q) = (R(s,a) + \gamma Q(s_{t+1}, argmax_{a_{t+1}}; \theta_i^-) - Q(s,a,\theta_i))^2 \tag{3.8}$$

$$J_n(Q) = r_t + \gamma r_{t+1} + ... + \gamma^{n-1} r_{t+n-1} + max_a \gamma Q(s_{t+n}, a) \tag{3.9}$$

$$J_E(Q) = max_{a \in A}[Q(s,a) + l(a_E, a)] - Q(s, a_E) \tag{3.10}$$

$$J(Q) = J_{DQ}(Q) + \lambda_1 J_n(Q) + \lambda_2 J_E(Q) + \lambda_3 J_{L2}(Q) \tag{3.11}$$

The notation a_E represents the action selected by the demonstrator in current state s and the value $l(a_E, a)$ represents the margin part. The margin part value becomes 0 when the agent's action and the demonstrator's action are the same $a = a_E$. The λ parameters control the weighting between the losses.

3.3.2 The Interacting Phase

Once the first phase is complete, the DQfD agent starts interacting with the environment by selecting an action with its learned policy, receiving the reward, and moving to the next state. This data are known as self-generated data, and the agent saves it to the agent's replay buffer. When the agent's replay memory exceeds the specified size, the agent starts overwriting the previously generated samples without overwriting the existing demonstrations. In this phase, the agent updates its network with a mix of self-generated and demonstration data. The DQfD algorithm used the prioritized replay mechanism [6] to control the ratio between sampling the demonstration and self-generated data. All the losses are used to train the agent in this phase, except the supervised large margin classification loss ($J_E(Q)$) is not applied to self-generated data. It is necessary for the demonstration data only since these data cover a limited number of states and do not take all possible actions. This guarantees to assigned the other state-action pairs to real values.

In general, the main difference between the DQFD algorithm and the traditional deep reinforcement learning algorithms can be summarized in these three main points:

- Training samples: DQfD relies on a group of pre-built samples held in the agent's

memory during the training period.

- Pre-training phase: all previous algorithms trained their agents with only the interacting phase.
- Losses function: The DQfD algorithm used four losses functions while all other algorithms only used one loss.

Algorithm 2 shows the Pseudo-code of the DQfD.

Algorithm 2 DQfD

Input: D_{replay}: contains demonstration data, θ: behavior weights (random), θ^{-}: target weights (random), T: target network update frequency, k: steps of pre-training phase
for steps $= 1$, k **do**
    Select samples with prioritization from D_{replay}
    Compute loss J(Q) by the target network
    Update the θ by using the gradient descent
    if steps mod $T = 0$ **then**
        $\theta^{-} \leftarrow \theta$
    end if
end for
for steps $= 1, 2, \ldots$ **do**
    Select action using learned policy
    Play action a and observe (s_{t+1}, r)
    Save new samples (s, a, r, s_{t+1}) in the memory, overwriting previous self-generated samples if memory is full
    Select samples with prioritization from D_{replay}
    Compute loss J(Q) by the target network
    Update the θ by using the gradient descent
    if steps mod $T = 0$ **then**
        $\theta^{-} \leftarrow \theta$
    end if
    $s \leftarrow s_{t+1}$
end for

3.4 Inverse Reinforcement Learning

Inverse reinforcement learning (IRL) aims to understand the decisions of another agent from its performance and therefore avoid the manual design of its reward function [47]. IRL can use recorded demonstrations to build agents capable of performing a task without any prior knowledge about that task. For this reason, many communities such as artificial intelligence, control theory, and psychology have become more interested in IRL.

Since the IRL's agent follows the expert's policy built under the Markov decision process (MDP), the MDP can also be used as a framework for the IRL agents. The main difference is the agent learns from the expert's optimal policy rather than the environment reward. Figure 3.5 shows the difference between the MDP approach for RL and IRL. Algorithm 3 shows the standard template for most IRL methods.

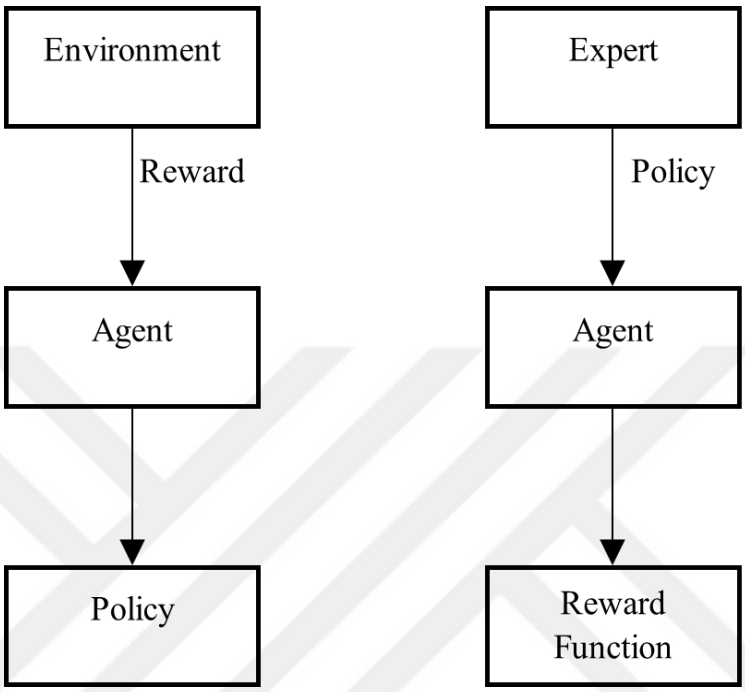


Figure 3.5 RL (left) vs IRL (right).

Algorithm 3 IRL Standard Template

1: **Input:** Trajectory (S, A, T, γ), group of samples showing the expected behavior: $D = (s_0, a_0), (s_1, a_1), ..., (s_t, a_t), s_t \in S, a_t \in A, t \in N$, or a policy that describes the expert's behavior.
2: **Output:** Reward Estimation R_E
3: Assign the expert's behavior as the optimal solution for MDP
4: Set reward function parameters using linearly weighted, distribution through rewards, or any different way;
5: Develop new policy or behavior by applying the existing reward function in the MDP.
6: Update the reward function parameters using IRL methods such as Entropy optimization, Margin optimization, etc to reduce the variation between the expert's policy and the current policy.
7: Repeat steps 5 and 6 until the policies variation becomes acceptable.

Existing techniques try to build a reward function to explain the expert's behavior described in various forms. These techniques can be a mix of weight-based functions, states to reward mapped values using probability diffusion, and others. The following subsections classify IRL techniques depending on the main principles of inverse learning into four classes which are Margin, Entropy, Bayesian, and the last one is classification and regression [48].

3.4.1 Margin optimization

The margin optimization [49] seeks to build a function that explains an expert's behavior in a way more accurate using the margin. This class includes many methods to find the reward function by maximizing some margin, and these methods are classified depending on the type of margin used.

The optimal margin optimization is the earlier and uncomplicated way to build policy from other policies. The margin can be found by the summation of the distinctions between the optimal policy action value and the following action using the same policy through all states:

$$M_{optimal} = \Sigma_{s \in S} Q^{\pi}(s, a^*) - max_{a \in A} Q^{\pi}(s, a) \tag{3.12}$$

Where Q^{π} is the Q value for the expert policy π, the a^* is the optimal action for s. The earliest method that used the margin in the equation above is Ng and Russell's [17]. They used the expert's policy as input and recover the reward function that provide the optimal output in MDP.

The second method is the margin observed from learned feature expectations, which used the feature to find a reward function that minimizes the margin between the feature expectations of a leaner policy and the feature expectations from the expert's policy. Max-margin and projection methods use the feature expectation loss [17]. In practice, the expert's policy can not reach by the learner. On the contrary, the learner takes the demonstration of the expert as input. In this method, the reward function represents by the feature functions as a sum of linear weights.

The last margin method is the learned policy over the distributed actions. This method is a second choice after the previous method and seeks to reduce the dissimilarity between the random policies per state. Since the demonstration data is obtainable rather than the expert's behavior, the dissimilarity between the random policies is found depending on the estimated state visitation times. Briefly, how many times has

the same state been visited by the same policy? Hybrid-IRL [50] used the observed and learned policy distributions to solve the problem by employing the gradient descent in the reward theories space.

3.4.2 Entropy optimization

The main disadvantage in IRL is the ambiguity, where many reward functions can be derived from the same behavior for the same expert. The margin optimization introduced previously enters a bias for the reward function. Many methods used the maximum entropy principle [51] to obtain the distribution over all reward functions. Distribution with maximum entropy is better than the other distributions. The methods that use entropy optimization can be categorized based on the distribution used by the process.

The first method is the distribution entropy over tracks or policies. The reward function builds from the policies rather than transitions with the maximum entropy in this method. The reason for doing this is that the search space grows exponentially with time, leading to excessive growth. Ziebart et al. [52] built a popular IRL technique by adding two constraints. First, the distribution over all policies should be a probability distribution. Second, the expected feature count of the learner must match the count obtained from the expert's demonstration.

Relative distribution entropy over tracks is another method of entropy optimization for IRL approaches. This method suggests minimizing the relative entropy between two different reward functions over the policies. Relative Entropy Inverse Reinforcement Learning (REIRL) [53] is a famous method that uses the relative entropy of the distribution. Their results in car racing video games showed that the relative entropy of the distribution could obtain expert reward function and learn a good policy from a small number of demonstrations.

3.4.3 Bayesian update

Another method is the Bayesian update. In this method, the state action pairs are treated as statements that enable a primary distribution of Bayesian updates over reward functions. The Bayesian update generated various principled IRL methods where no reward function has a typical structure. Let $P_r(R_E)$ be an initial allocation over the functions of reward and $P_r(T|R_E)$ be the statement probability of the reward theory. Then, reward functions allocation is received using the following Bayesian

update:

$$P_r(R_E|T) \propto P_r(T|R_E)(R_E) \tag{3.13}$$

Where (R_E) is the reward function, and T is a trajectory that contains state-action pair. Bayesian update methods categorize depends on type of observation likelihood model in the update above. The Bayesian update is executed with the same amount of trajectories in the expert demonstration. The probability function famous option is Boltzmann distribution (Gibbs distribution) with function of state-action Q-value calculated as:

$$P_r((s,a)|R_E) \propto e^{\frac{Q^*(s,a,R_E)}{\beta}} \tag{3.14}$$

The notation β management the state-action probability (higher β, the more priority selection of state-action pairs). The earliest Bayesian IRL technique is Bayesian Inverse Reinforcement Learning (BIRL) [19], which used the Boltzmann distribution. BIRL offers several various information about the reward function with continuous values. Analytically obtaining the posterior is a challenge in continuous reward functions space. To solve this problem, BIRL used the Markov chain Monte Carlo (MCMC) algorithm in policies space to obtain an approximation of the posterior for each sample. Another solutioın for this challenge is using gradient descent as in [54]. Lopes et al. [55] demonstrate that the rear calculated in BIRL can use include learning from interacting in IRL. The agent asks the supervisor for more data of state-action pairs where the entropy is high. These new state-action pairs enable the learner to improve the posterior.

The second option for the likelihood function is Gaussian process. The Gaussian process is an effective approach by allowing a nonlinear reward function $R_E = f(r, \phi)$. The structure of the reward function is provided by a function parameterized by θ. The previous rear function becomes $P_r(r, \theta|T, \phi)$ in which r is a reward connected with a feature functions ϕ at a pair of states and actions. The Gaussian Process Inverse Reinforcement Learning (GPIRL) method [56] calculates the rear as a Bayesian update together with Gaussian distribution probability. GPIRL circulates from only a slight subset of values included in the experimental samples with rare extra rewards within random states.

The last model for the likelihood function is Maximum likelihood estimation. Maximum likelihood immediately increases the probability of data for the posterior

estimation in BIRL [19] methods. Vroman et al. [57] in the technique Maximum Likelihood Inverse Reinforcement Learning (MLIRL) determine the Boltzmann exploration policy to assume feature weights of the reward that guides to the highest probability calculation. As a differentiable probability, the algorithm connects to the optimal weights by employing gradient ascent with standard criteria.

3.4.4 Classification and Regression

One of the machine learning techniques that served as a significant factor in IRL is the Classification and Regression methods. The action-value scores Classification and state-space divisions Regression tree methods show that the IRL can be launched into such framework.

The action-value scores classification developed the IRL as a multi-class classification problem. The pair of state and action can be used as data label pairs. In most domains, there are two or more activities that lead this classification into multiple classes classification. Klein et al. [58] introduced Structured Classification Inverse Reinforcement Learning (SCIRL) which is an IRL method with multi-class classification. The classifier is a reward function trained with the demonstration data. SCIRL select the large margin method of structured prediction [59] as the algorithm for classification. While SCIRL assumes the presence of a transition model to compute the scoring function, an extension called Cascaded Supervised Inverse Reinforcement Learning (CSIRL) [60] takes a step further and estimates the transition probabilities if they are not known. CSIRL uses standard regression on a simulated demonstration data set to estimate the transition model and thereafter learn the reward function using SCIRL. Notice that SCIRL is predicated on the action in the state-action pair being the desired optimal action and uses it as the label. In this context, Brown et al. [61] shows that if the input to IRL also includes a ranking of the trajectories based on the degree of suboptimality of the trajectory, the model may utilize this additional preference information to fit a neural network representation of the reward function. A cross-entropy-based loss function trains the neural network to obtain a higher cumulative reward for the trajectory that is preferred over another per the given ranking.

Representing the reward function as a linearly weighted sum of features over the entire demonstration may be ineffective or require many features when the demonstration size is large. The state-space division's regression tree is an alternate solution in which the nodes represent the features, while the leaves define the intersection of features. The state-action represents by a path of the tree, and the whole tree generates a section of the demonstration. Feature Inverse Reinforcement Learning (FIRL) [62] builds

the regression tree and the features frequently to get to the current regression tree reward function. In the last few years, Uchibe [21] used logistic regression to train three networks for value-function approximations, reward function, and the ratio of controlled and uncontrolled state transitions in the demonstrations.

4
PROPOSED SYSTEM

As we mentioned earlier, many existing systems failed in many environments due to the distribution of rewards. Such environments are known as hard-discovering environments. Montezuma's Revenge, Private Eye, and Pitfall are popular examples of these environments. For example, Montezuma's Revenge requires hundreds of actions to produce a reward. The random manner used in baselines makes a reward every half a million steps. This scaling limits the abilities of the Deep Reinforcement Learning (DRL) techniques. Algorithms built to solve this problem were complex and required hardware with high computation ability. Accordingly, we suggest a new modification to the available methods to cover the disadvantages above. This chapter will explain the details of the modification suggested in this thesis.

4.1 Inclusion External Reward

Since the agent uses one architecture to play several different games, this means that the different representation of the games or the difference in the difficulty does not affect its performance. Also, the difference in the value of rewards from one game to another does not affect the performance of the agent because normalizing the rewards to 1, 0, and -1 for all games. The only factor that led to the failure of the agents is the missing of rewards.

From this point, we suggested using an external reward makes all environments share the same reward feature. We build new demonstrations that contained these rewards. After that, we used these demonstrations to train one of the most popular algorithms to prove the efficiency of our approach. All the details of building the demonstrations and the selected algorithm will be explained in the next subsections.

4.1.1 Pre-Training Phase

The pre-training phase is the first stage for the agent to learn how to imitate the same behavior that was used to build the demonstration. First, we must build demonstrations to be used later in the training. These demonstrations will contain external rewards. A human player selects the correct action in each step and sets that action's reward to 1. The reward of the rest actions is 0. Assigning actions with 0 is essential to give all actions realistic values because the demonstration is visiting a limited number of the states and not selecting all actions in the action space. These absolute values prevent the network from updating toward ungrounded variables. The selected action and the assigned reward are saved alongside the current state and the next state. These four pieces of information represent one sample. Also, the rest actions and the assigned rewards are saved with the same current state and the next state separately. Thus, for each step, we are saving several samples equal to the number of actions in the game. The player keeps playing until filling the demonstration with a specific number of samples. Figure 4.1 shows the demonstration building procedure.

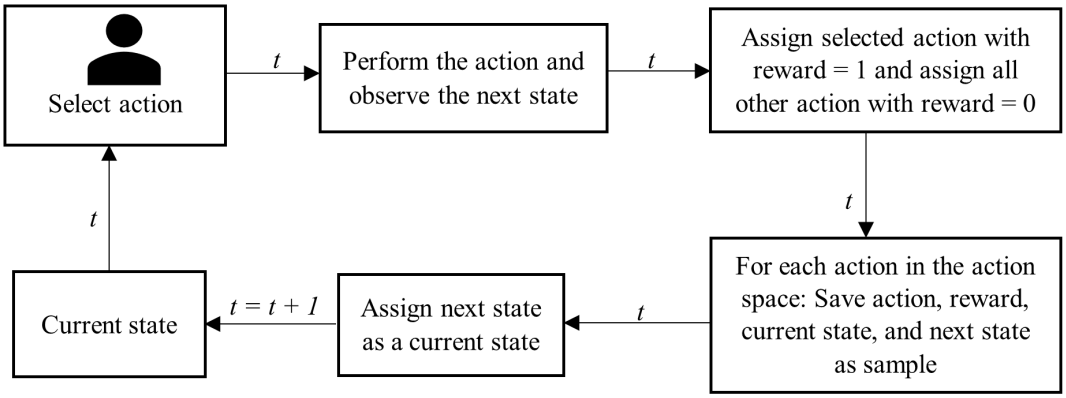


Figure 4.1 Demonstration building procedure

We built five different demonstrations for five different games to train and test our agent. These games are Montezuma's Revenge, Pitfall, Private Eye, Pong, and Breakout. Each demonstration builds by playing the game and selecting actions by the keyboard. It is not required to be a super player to select the correct actions only. The game waits for us each time to select the action by the keyboard, which gives us time to select the action correctly. During building the demonstration, we found many actions in the hard exploration games are useless. Thus, we reduce the number of actions from 18 to 8. The actions number, episodes number, total steps, and scores obtained in our demonstration for each game are shown in Table 4.1.

Table 4.1 The demonstration actions number, episodes number, total steps, and scores for each game

Game	Best Score	Worst Score	Total Steps	Episodes	Actions Number
Montezuma Revenge	32000	30000	15900	5	8
Pitfall	35711	3971	27908	5	8
Private Eye	20700	20300	8636	5	8
Breakout	79	17	10289	9	4
Pong	1	-1	6107	3	6

When the construction of the demonstration is completed, the next step is training the agent to measure the effect of the external rewards. Deep Q-Network (DQN) [2] is the notable agent that used the DRL principles to learn how to play video games. Later, two extensions were added to the DQN agent to increase its performance. These extensions are Double Deep Q-Network (DDQN) [5] and Prioritized Experience Replay (PER) [6]. In this thesis, we used the Prioritized Double Deep Q-Network (PDDQN) algorithm to build our agent. The objective of the pre-training phase is to learn to imitate the player behavior with a value function that meets the Bellman equation so that it can be updated once the agent begins interacting with the environment. During this phase, the agent samples mini-batches from the demonstration data and updates the network by applying the DDQN loss on the network weights and biases. The DDQN loss guarantee that the network can continue learning later by fulfilling the Bellman equation:

$$Q(s,a) = (r(s,a) + \gamma Q(s_{t+1}, argmax_a Q(s_{t+1}, a, \theta); \theta^-) - Q(s,a;\theta))^2 \qquad (4.1)$$

Where r (s, a) is the environment's reward for the current state s and action a, γ is the discount factor, s_{t+1} is the next state, θ is the current network weights, and θ^- is the target network weights. The left side of the subtraction represents the target value while the right side is the agent network prediction. The DDQN uses the current network to select the action, and the target network to calculate the Q value of that state-action pair.

Our agent selects samples from the demonstration data using the PER. According to the PER, each sample has a priority value calculated by taking the absolute value of subtracting the target value from the network prediction for that sample. Previous works sampled 60 percent of mini-batch samples by priority and the rest samples were selected by random. The 60 percent increase annealed linearly to reaching 100 percent, which means selecting all samples with priority. In the pre-training phase, we

selected 90 percent of mini-batch with priority and only 10 percent by random. Also, we increased this percent in the same way. The reason to select 90 percent is that all the samples in the demonstration are important to us, and we want to learn from all of them. The pseudocode of our demonstration building and pre-training phase is shown in Algorithm 4.

Algorithm 4 Phase One

Input: D^{buffer}: empty buffer for storage the demonstrations. θ: weights for behavior network (random initialization), θ^{-}: weights for target network (random initialization), T: target network update frequency, k: steps of pre-training phase s_t: human demonstrations steps, k: number of pre-training gradient updates.
for steps $= 1$, s_t **do**
    Select action by the human player.
    Play action a on state s_t and observe s_{t+1}.
    Set the reward of the selected action with r $= 1$ and the rest with r $= 0$.
    Store (s_t, a, r, s_{t+1}) into D^{buffer}.
    $s_t \leftarrow s_{t+1}$
end for
for steps $= 1$, k **do**
    Sample a mini-batch of n transitions from D^{buffer} with prioritization.
    Calculate loss using the DDQN loss.
    Perform a gradient descent step to update θ.
    if steps mod $T = 0$ **then**
        $\theta^{-} \leftarrow \theta$
    end if
end for

4.1.2 Interacting Phase

Once the pre-training phase is complete, the agent starts acting on the environment by selecting action using its learned policy, collecting self-generated data, and learning from that data. The objectives of phase two are to explore new states and make the agent performance more stable. However, once the agent started interacting with the environment, the learned policy performance decreased. The reason here is the reward distribution. In phase one, the agent received 0 or 1 reward for each action. On the other hand, phase two depends on environment rewards. These rewards have different time intervals variations from game to game. Figure 4.2 shows the effect of reward distribution on agent performance in Montezuma's Revenge.

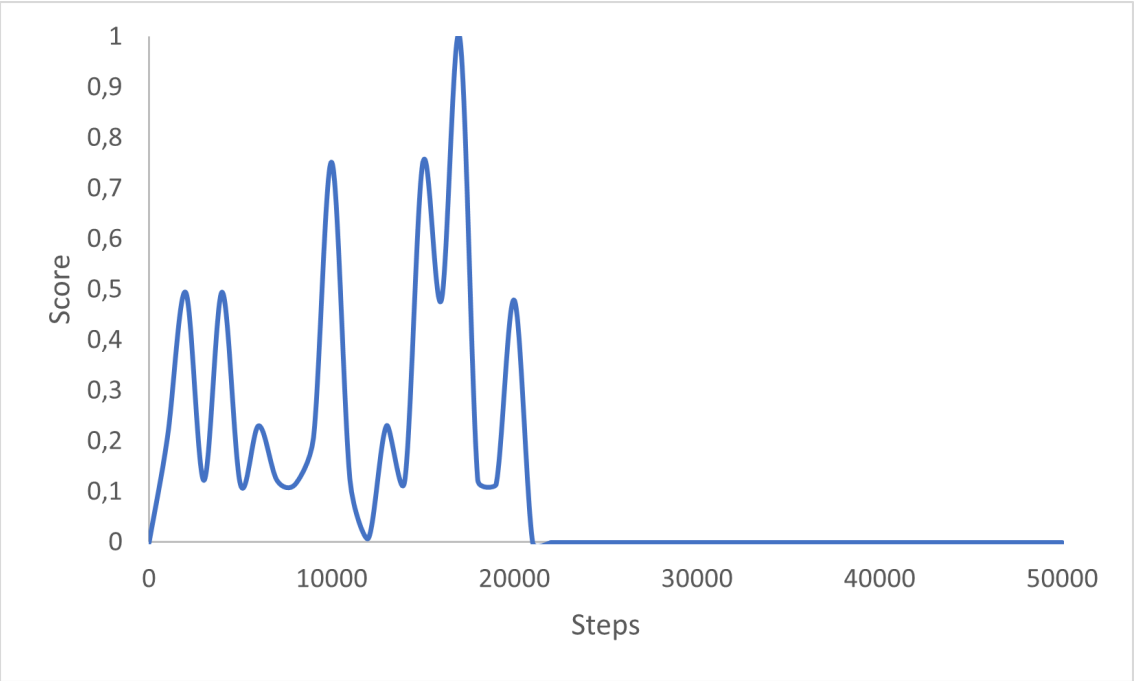


Figure 4.2 The performance of our agent in phase two for a hard exploration environment. The agent starts learning from self-generated data after performing 20,000 steps. The results are normalized between 0 and 1

Figure 4.2 proves that the agent performance gets effect by the newly generated data. The agent runs on the environment for 50000 steps which is enough to show the problem. To solve this problem, we propose to use the concepts of inverse reinforcement learning to extract new auxiliary rewards that help the agent during phase two. Inverse reinforcement learning generates a reward function from the demonstration. This reward function makes interacting with the game environment possible and enables the PDDQN algorithm to continue learning and enhancement.

4.2 Inclusion Inverse Reinforcement Learning

Inverse Reinforcement Learning (IRL) main concept is learning the Reward Function (RF) from the demonstration data. The RF explains the policy that the expert followed during building the demonstrations. The agent will select the action and the RF will provide a reward for the selected action. This reward will be added to the environment reward. Using both the reward from the RF and the environments reward provide 3 main advantages:

- Uniform the reward distribution in the two phases.
- Take advantage of the environment's reward in simple environments.

- Enable the agent to discover new states beyond the demonstration states.

Figure 4.3 shows the differences between the standard RL framework and our framework.

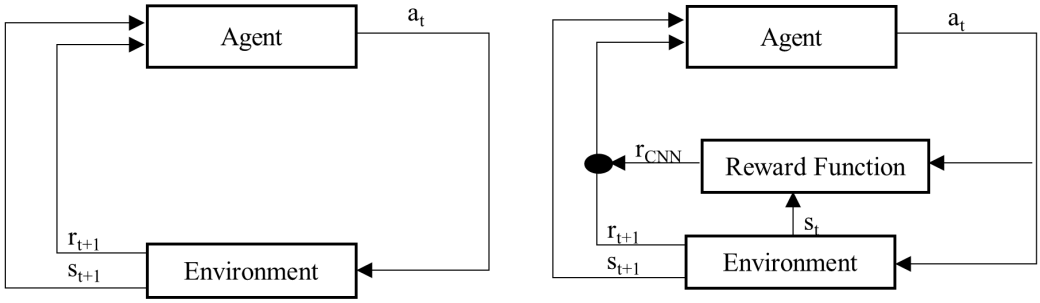


Figure 4.3 The left plots show the standard MDP for solving Reinforcement Learning problems. The right plots show our new framework by adding the Reward Function

4.2.1 Preparation of Reward Function Learning Datasets

The training of reward function learning datasets have been extracted from the previous phase demonstration. The RF's main objective is to evaluate the selected action according to the state in which it was selected. So, the input of the RF is the current state, and the output is the reward for each action in that state. Recall our demonstration construction, we set the reward of the selected action equal to 1 and the other action with a reward equal to 0. That's means we have several samples for the same state. To cover this problem, we selected the sample with a reward equal to 1 only. Such samples contain the current state and the action number. The current state in that sample is considered as an input to the RF. A zeros array with a size equal to the number of actions of the environment is a target output for the RF. Only the action in that sample is assigned with 1 in the target output. The processor of building the RF dataset is shown in Figure 4.4.

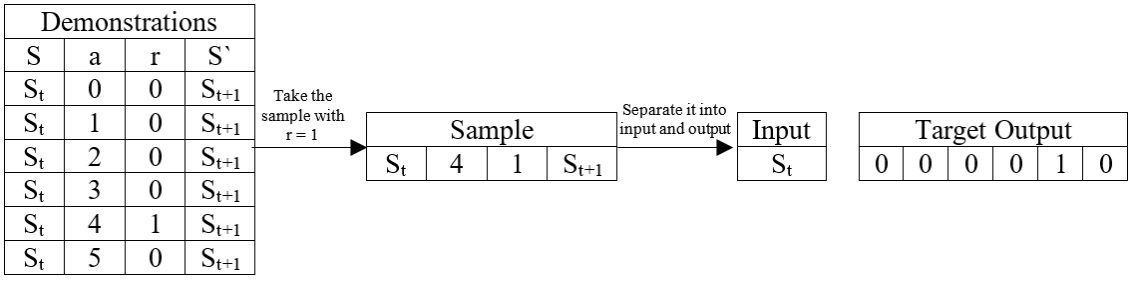


Figure 4.4 Reward function dataset building procedure

4.2.2 Reward Function Architecture

We build a separate Convolutional Neural Network (CNN) to serve as an RF. The RF has the same architecture as the agent regular CNN architecture from phase one. The main differences are the output activation function and the loss function. In the agent's CNN, we used a linear activation function. In the reward function, we used the SoftMax function. For the loss function, we used Categorical Cross-Entropy rather than the Huber loss. The input to the RF is the same input to the agent network (4 pre-processed states stacked together). The output of the RF is an array with zero values for all actions except the correct action for that state which will assign with a reward equal to 1. Algorithm 5 shows the pseudocode for training the RF network.

Algorithm 5 Training the Reward Function

1: **Input:** R^{buffer}: the reward dataset. θ: weights for behavior network (random initialization), k: number of training gradient updates.
2: **for** steps $= 1$, k **do**
3: Sample a mini-batch of samples from R^{buffer} with random.
4: Calculate loss using the Categorical cross-entropy.
5: Perform a gradient descent step to update θ.
6: **end for**

4.3 The Integrated Agent

The integrated agent performs both the pre-training and interacting phases. Initially, the agent starts to imitate the demonstration behavior by sampling a mini-batch from the demonstration to train the PDDQN algorithm. The total training steps in this phase are 250,000 steps. Once the imitating learning is done, the agent trains the reward function on the dataset extracted from the same demonstration for 20,000 steps. The last task of the agent is interacting with the environment. In this phase, the agent starts selecting action using the learned policy from phase one and saving the new samples in an empty folder. As in phase one, the sample contains the current state, action number, reward value, and the next state. The reward in the sample is a result of summation environment reward with the reward from the reward function.

The last thing to mention is comparing our agent with the Deep Q-learning from Demonstrations [12] (DQfD) agent. Among all previous DRL agents, the DQfD agent is the most similar agent to our agent in the architecture. We compare most of our results with the DQfD agent results. The DQfD agent has both the pre-training and the interacting phases. The DQfD agent learns from both the demonstration data and the self-generated data. The DQfD agent differs from ours in rewards only. The DQfD agent uses the environment rewards in both phases while our agent uses the

external reward in phase one and the external and environment rewards in phase two. Algorithm 6 shows the pseudocode of our integrated agent, and Figure 4.5 compares our architecture with the DQfD architecture for both phases.

Algorithm 6 Our Agent with Phase One and Phase Two

Input: D^{buffer}: buffer contains the demonstration data. A^{buffer}: empty buffer for storage agent self-generated data. θ_1: weights for behavior network in phase one (random initialization), θ_1^-: weights for target network in phase one (random initialization), θ_2: weights for behavior network in phase two (weights will copy from phase one network), θ_2^-: weights for target network in phase two (weights will copy from phase one network), θ_3: Reward Function weights, T: target network update frequency, k: steps of pre-training phase
Pre-training Phase
for steps $= 1$, k **do**
    Sample a mini-batch of n transitions from D^{buffer} with prioritization.
    Calculate loss using the DDQN loss.
    Perform a gradient descent step to update θ_1.
    if steps mod $T = 0$ **then**
        $\theta_1^- \leftarrow \theta_1$
    end if
end for
Interacting Phase
for steps $= 1, 2, \ldots$ **do**
    Select action a by the trained policy from phase one.
    Play the action a and observe (s_{t+1}, r).
    Clip positive reward at 1 and leave 0 rewards unchanged.
    Calculate reward r_{RF} for s_t and a by the Reward Function with θ_3.
    Calculate the final reward $r = r + r_{RF}$
    Store (s_t, a, r, s_{t+1}) into A^{buffer}
    Select samples with prioritization from A^{buffer}
    Compute loss using DDQN loss
    Update the θ_2 by using the gradient descent
    if steps mod $T = 0$ **then**
        $\theta_2^- \leftarrow \theta_2$
    end if
end for

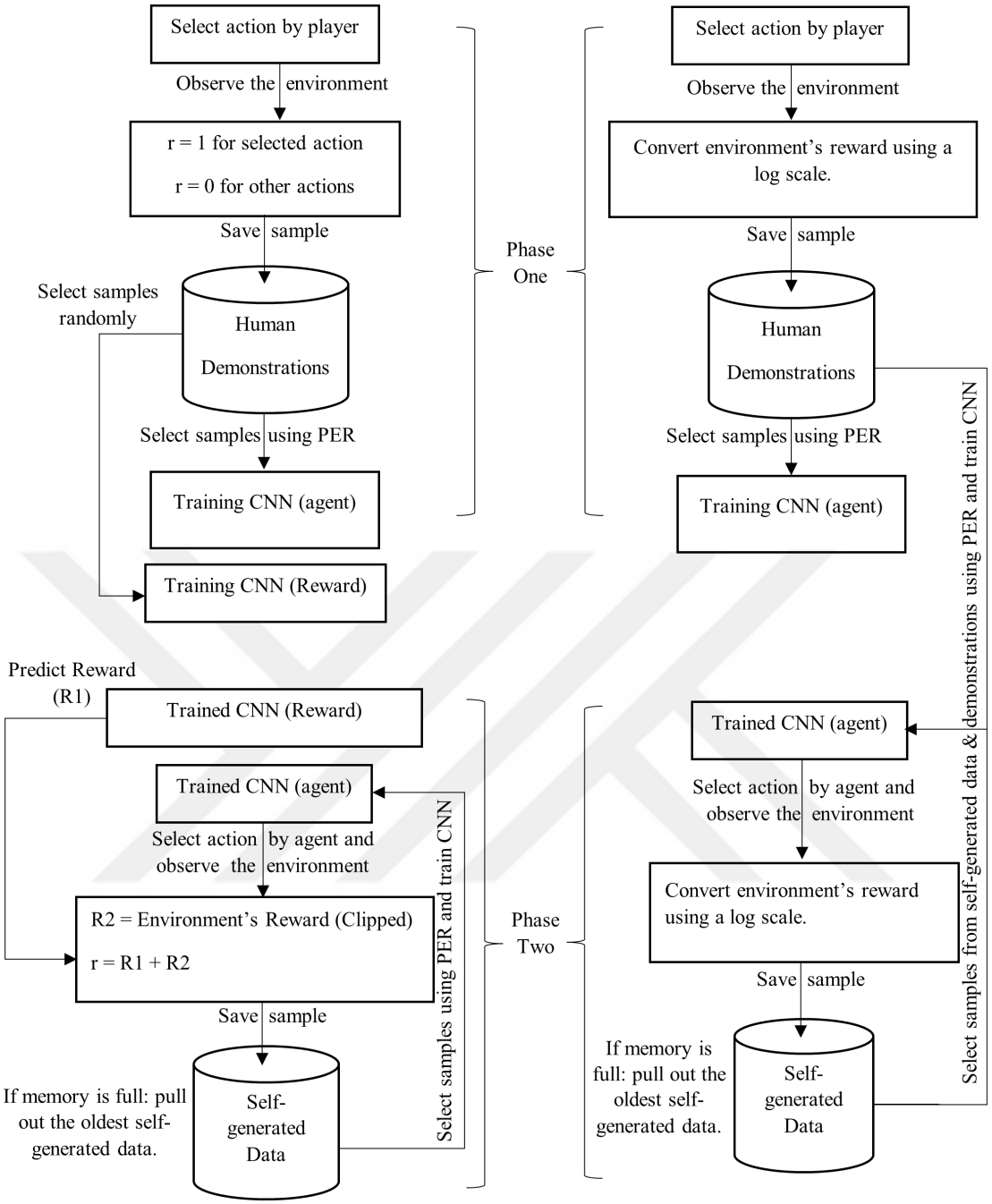


Figure 4.5 The difference between our agent architecture (left) and the DQfD agent architecture (right)

5
EXPERIMENTAL SETUP

In the essence of this thesis study, enabling simple and basic algorithms to play hard exploration games has been proposed by using external rewards. The thesis stands on the basis that the missing reward is the main challenge for most deep reinforcement learning techniques. The external reward solves this challenge by covering the absence of environmental rewards and reducing the time required to reach acceptable performance. The external reward has been used in two forms. The first form is implying it within the demonstrations and the second form is predicting it using the deep inverse reinforcement learning principles. The environment setup, parameters, and the results including the comparing results are explained detailly in the next sections.

5.1 Environment Setup

We trained and tested our agent on the Arcade Learning Environment (ALE) [37]. ALE become a standard benchmark for Reinforcement Learning (RL) algorithms since Deep Q-Network (DQN) used it. ALE is a set of video games known as Atari games that contain many different games each game is considered as an environment in which the agent can be trained and tested. We selected five different games from the classic Atari games. These games are Montezuma's Revenge, Pitfall, Private Eye, Breakout, and Pong. The first three games are popular examples of hard exploration environments. The other games are simple exploration games. Figure 5.1 shows screenshots from the five games that we used for training and testing our approach.

Montezuma's Revenge, Pitfall, and Private Eye are hard exploration games in which the player performs a sequence of actions to accomplish one reward. In these games, the player goes throw a number of rooms to collect rewards and avoid a number of enemies and traps. The place of rewards and other items is fixed. Therefore, the structure of the room remains constant regardless of the player's action.

On the other hand, Breakout and Pong are simple exploration games in which the player receives a reward for each action that has been selected. In these games, the player controls a paddle by two actions right or left to hit a ball to hit the ceiling or pass through another player. The player's action determines the direction and speed of the ball. Therefore, each state of the game changes depending on the selected action.

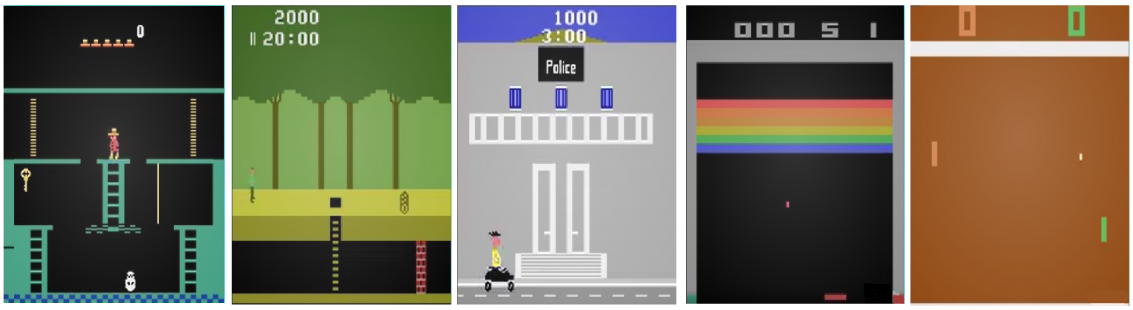


Figure 5.1 Screenshots from five Games: (Left-to-right) Montezuma's Revenge, Pitfall, Private Eye, Breakout, and Pong.

The agent receives the game screenshots as an environment state. The Atari game screenshot has 210 x 160 pixels with 128 colors which can be demanding in terms of computation and memory requirements. DQN applied a pre-processing step down-sampled the screenshot into an 84x84 grey-scale image. Since that, all Deep Reinforcement Learning (DRL) agents, including our agent, followed the same pre-processing step. As we mentioned earlier, every single state stacked 4 screenshots to determine the direction and speed of animated entities in the game.

There are two ways to evaluate DRL agents in ALE environments. The first way is evaluating the agent during the training by reporting game scores. In general, this way occurs when the agent interacts with the environment. The instant rewards are reported without any change regardless of how the agent is representing the reward internally. The second way is by taking the last or the best weight produced from the training stage and testing it by playing the game number of episodes. The game score is reported after finishing each episode. Also, the reward is reported without any change regardless of how the agent is representing the reward internally. We evaluated our agent in both ways. During the pre-training phase, there is no interaction with the environment so we saved several weights and tested them once the training is finished. In the interacting phase, we reported the game scores achieved by the agent immediately.

Depending on the evaluation methods mentioned above, we compared our agent performance with pre-published agents: DQN, DDQN, Prioritized DDQN, Dueling DDQN, Distributional DQN, Noisy DQN, Rainbow, and DQfD. It is impossible to train and test all these algorithms on our devices due to the hardware features and training

time requirements. To achieve a fair comparison, we followed the same evaluation settings of the published results that we took from [11] except for the DQfD's agent. The DQfD's agent is the most similar agent to our agent. Thus, we trained and tested the DQfD's agent in our device to get accurate results to enable us to make accurate comparisons between our agent and the DQfD's agent.

Finally, Table 5.1 shows the hyper-parameters used for the training and testing of the baseline, DQfD agent, and our agent. Some of these hyper-parameters were used in all algorithms and some of them were used in some algorithms depending on each algorithm. The Baseline hyper-parameters took from the same source of the results. The last thing to be mentioned is that each training occurs on a single Nvidia GeForce GTX 1060 graphics processing unit (6 GB memory version) with about 12 GB random access memory and an Intel Core i5 ninth generation processor.

Table 5.1 The parameters used for training and testing the algorithms.

Hyper-parameter	**Baselines**	**DQfD**	**Our Agent**
Grey-scaling	Yes	Yes	Yes
State size	(84, 84)	(84, 84)	(84, 84)
Frames stacked	4	4	4
Reward clipping	[-1, 1]	log scale	[0, 1]
Discount factor	0.99	0.99	0.99
Memory size	1M	1M	80k
Mini-batch size	32	32	32
Target network update	10k	10k	10k
Prioritized replay ratio	0.6	0.6	0.9 / 0.6
Exploration	Yes	Yes	N/A
Pre-training steps	N/A	750k	250k
Interacting steps	200M	200M	500k
Reducing actions number	N/A	N/A	$18 \rightarrow 8$
Reward function training steps	N/A	N/A	20k

5.2 Experimental Results and Discussion

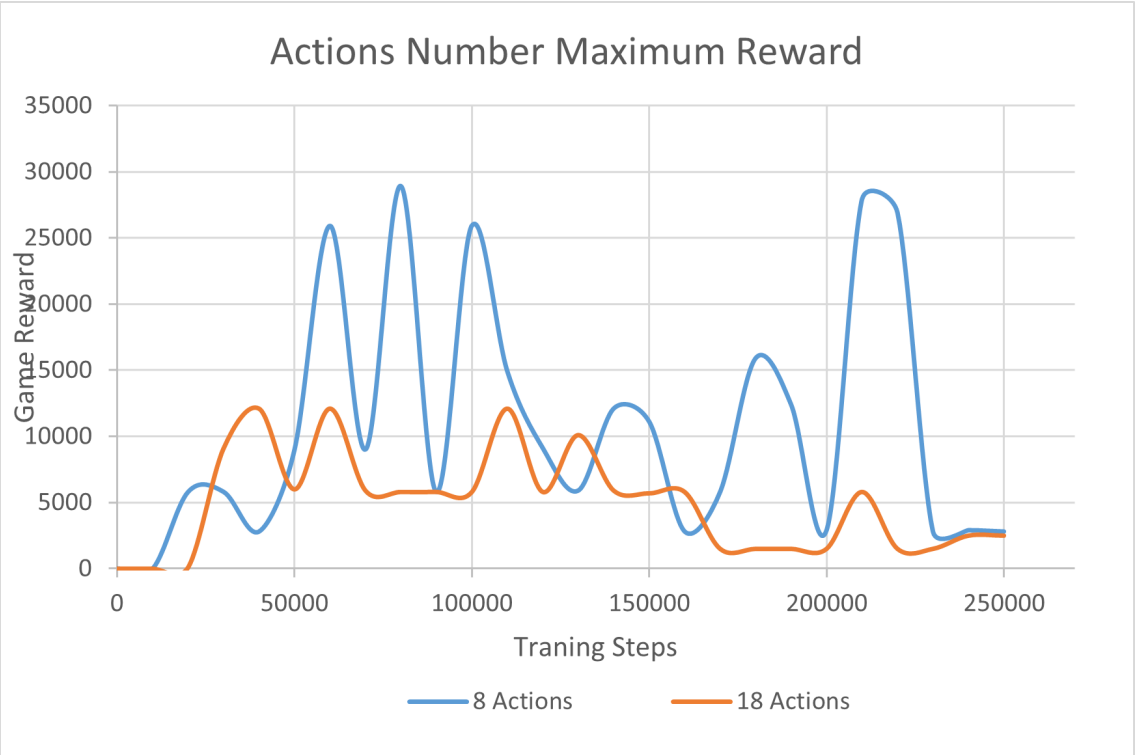
5.2.1 Pre-training Phase

All results in phase one are reported after finishing the training. The agent is trained for 250 thousand steps and after each 10 thousand, we are saving the network weights. Once the training is finished, we will have 25 different weights. we tested each weight on 10 episodes and took the average and maximum score for these 10 episodes.

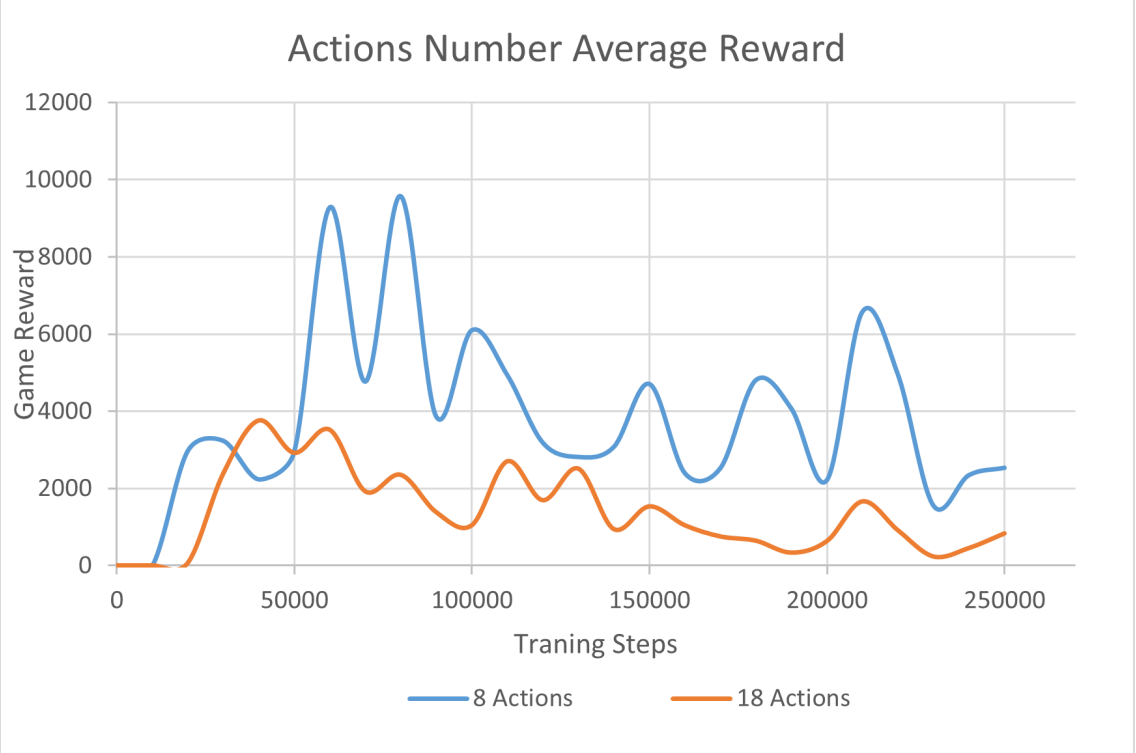
Initially, we compared the difference between the standard hyper-parameters and the new hyper-parameters that we used. For all these comparisons, we used

Montezuma's Revenge game which is a popular game in the RL community. The first hyper-parameter is reducing the number of actions. During building the demonstrations, we found that many actions are useless, and using these actions will be led to an increase in the number of samples which in turn increases the requirements of the memory size and training time. So, we trained and tested our agent on 8 actions (A sufficient number of actions to play the game and finish the first level) rather than the standard 18 actions. Figure 5.2 shows the results of testing the agent with eight actions and eighteen actions. Training the agent with 8 actions rather than 18 led to scoring 17000 scores higher and enable the agent to finish the first level of the game. We have used this number of actions in all the remaining hard exploration games (Pitfall and Private Eye).

The second hyper-parameter is the important sampling ratio. The standard ratio used in most RL algorithms is 0.6. In our demonstration, all samples are important and we want to learn from all of them. Thus we increased the important sampling ratio from 0.6 to 0.9. Figure 5.3 shows the results of testing the agent after training it with 0.9 and 0.6 rates. When we used a 0.9 ratio, the required training steps to get the maximum score are reduced from 120000 to 50000 only.

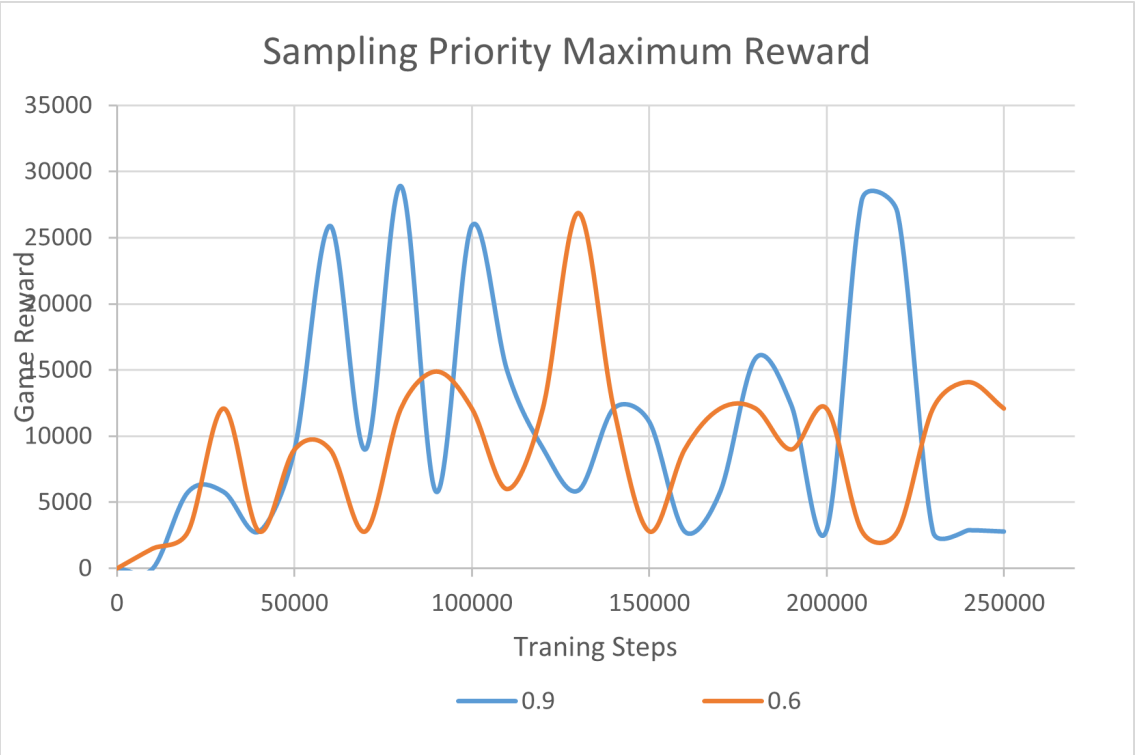


(a)

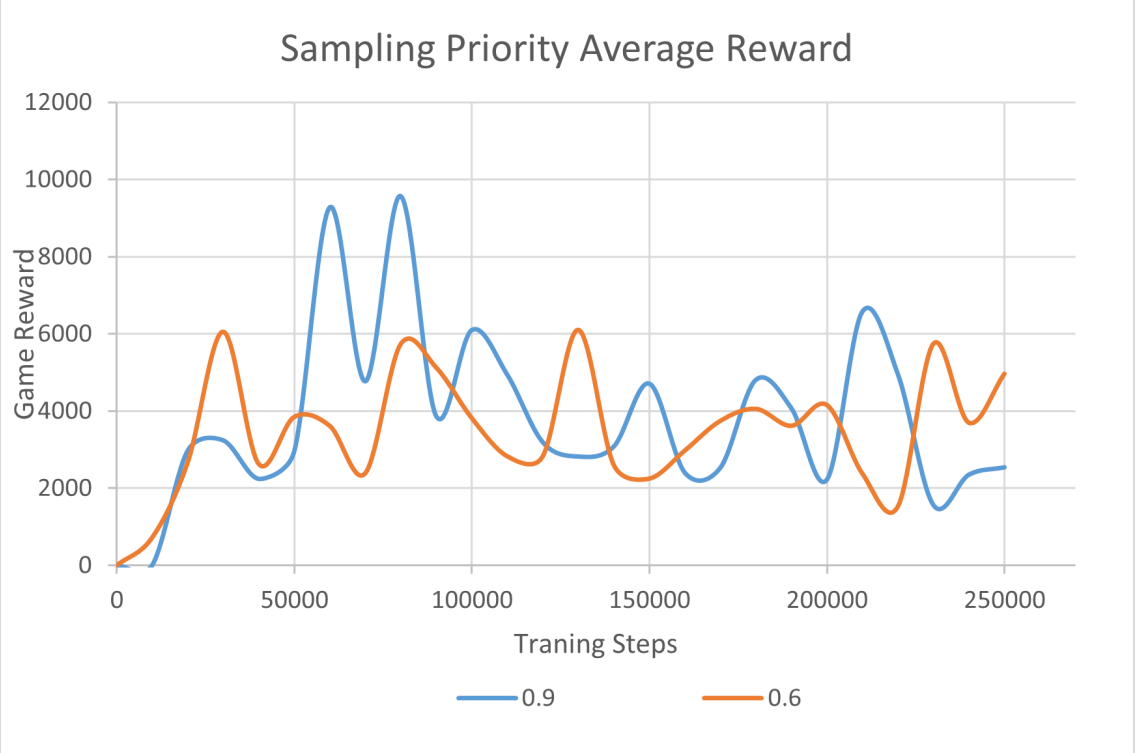


(b)

Figure 5.2 The difference between 8 actions and 18 action results. (a) plot shows the maximum scores. (b) plot shows the average scores



(a)



(b)

Figure 5.3 The difference between 0.9 PER and 0.6 PER results. (a) plot shows the maximum scores. (b) plot shows the average scores.

After proving the efficacy of the new hyper-parameters, the agent trained using the new hyper-parameters with the hyper-parameters introduced in Table 5.1. The agent trained on the five games we mentioned before. The results of these games are shown in figures 5.4, 5.5, 5.6, 5.7, and 5.8. The gray area represents the minimum and maximum scores of the demonstration used in training for each game.

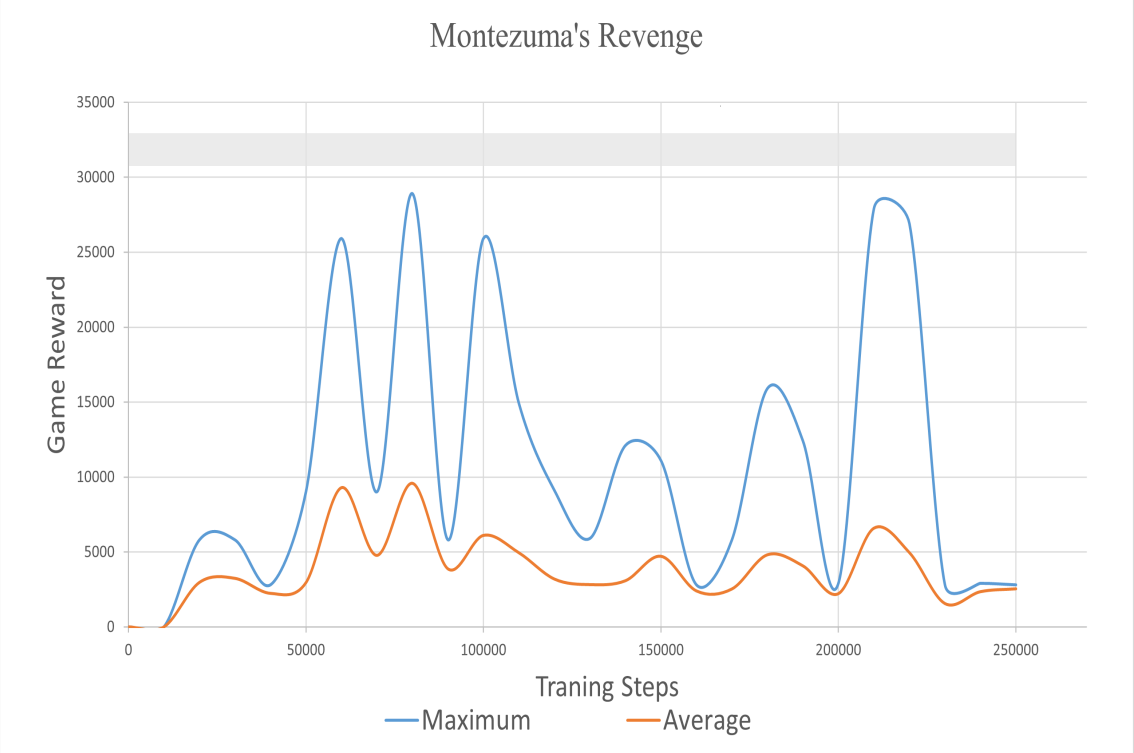


Figure 5.4 Maximum and average scores for phase one agent on Montezuma's Revenge game

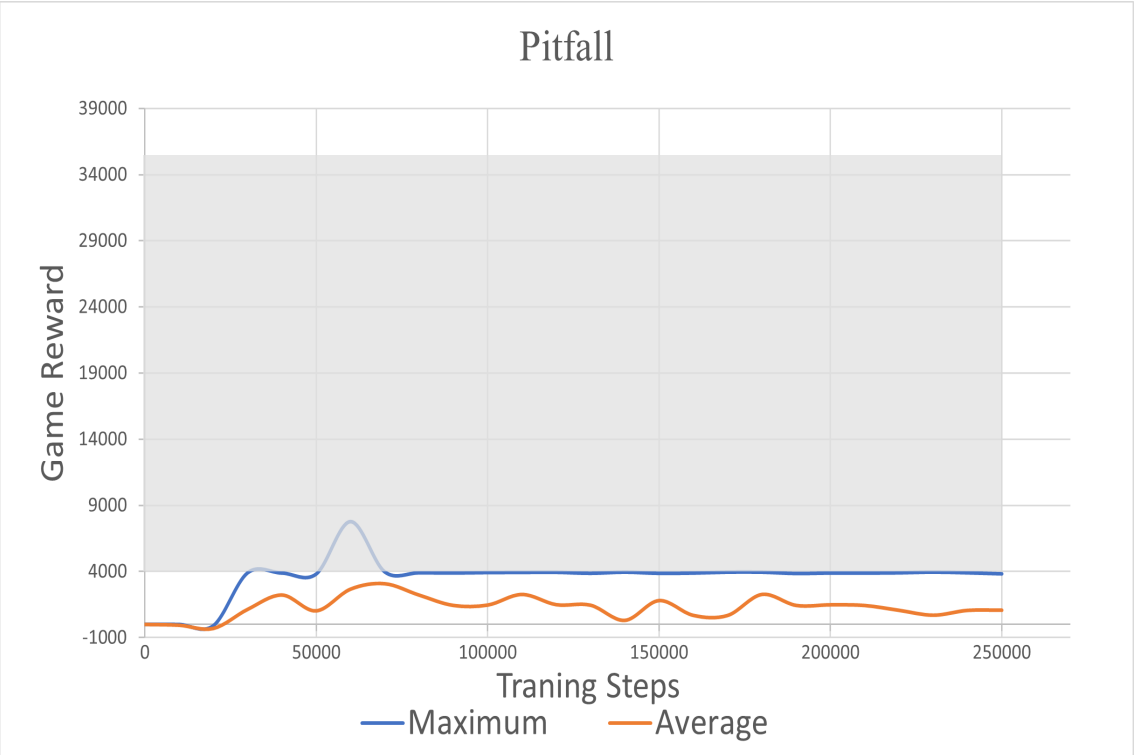


Figure 5.5 Maximum and average scores for phase one agent on the Pitfall game



Figure 5.6 Maximum and average scores for phase one agent on the Private Eye game

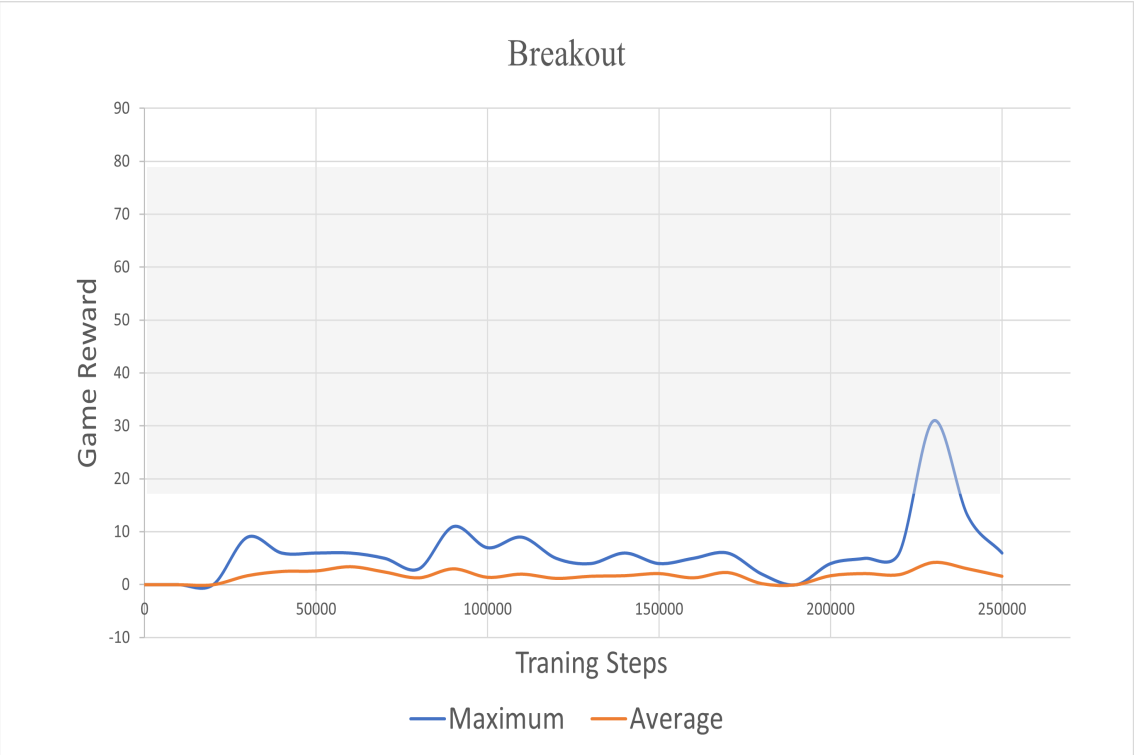


Figure 5.7 Maximum and average scores for phase one agent on the Breakout game

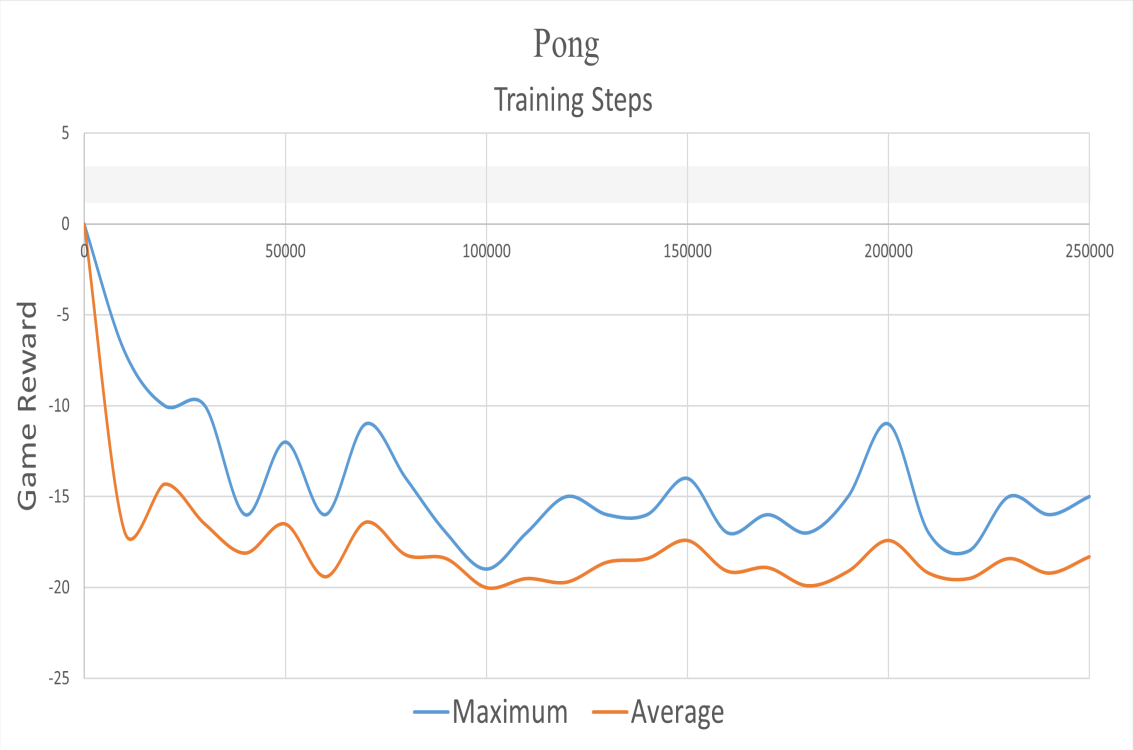


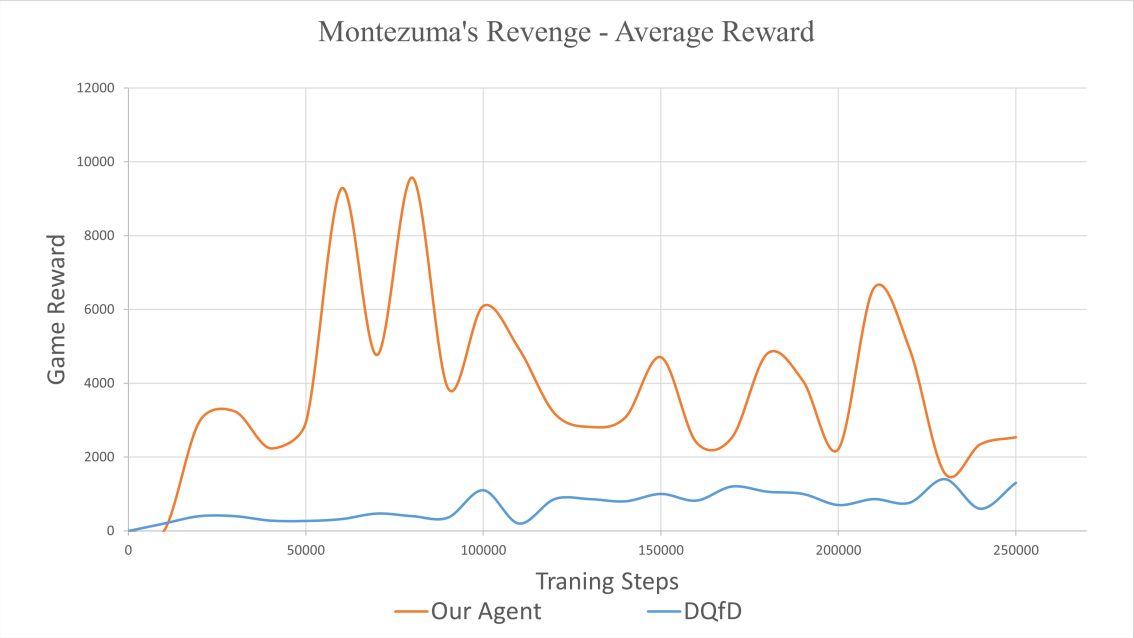
Figure 5.8 Maximum and average scores for phase one agent on the Pong game

These results are reported in two ways, the average of the ten episodes and the maximum score of these ten episodes for each weight. In the first game, Montezuma's Revenge, our agent scored 28.900 after only 80.000 steps of training. This score is meaning that the agent successfully finished the first level of the game which contains 24 different rooms. This weight is the best snapshot of our agent in Montezuma's Revenge game. For the rest of the weights, the results varied between 3000 and 29,000 scores. The second game is the Pitfall game. In this game, our agent has a more stable performance compared with the previous game. The agent scored 7782 after 60.000 training steps. This weight is the best snapshot of our agent in the Pitfall game. For the rest of the weights, the maximum score was 4000 scores for all weights. In the Private Eye game, the third hard exploration environment, our agent's performance is excellent. The agent takes only 40.000 steps to score higher than the demonstration's maximum score. The demonstration's maximum score is 20.700 while the agent's maximum score is 28.374. The fourth weight is the best snapshot of our agent in this game. For the rest of the weights, the maximum score was between 20000 and 21000 scores.

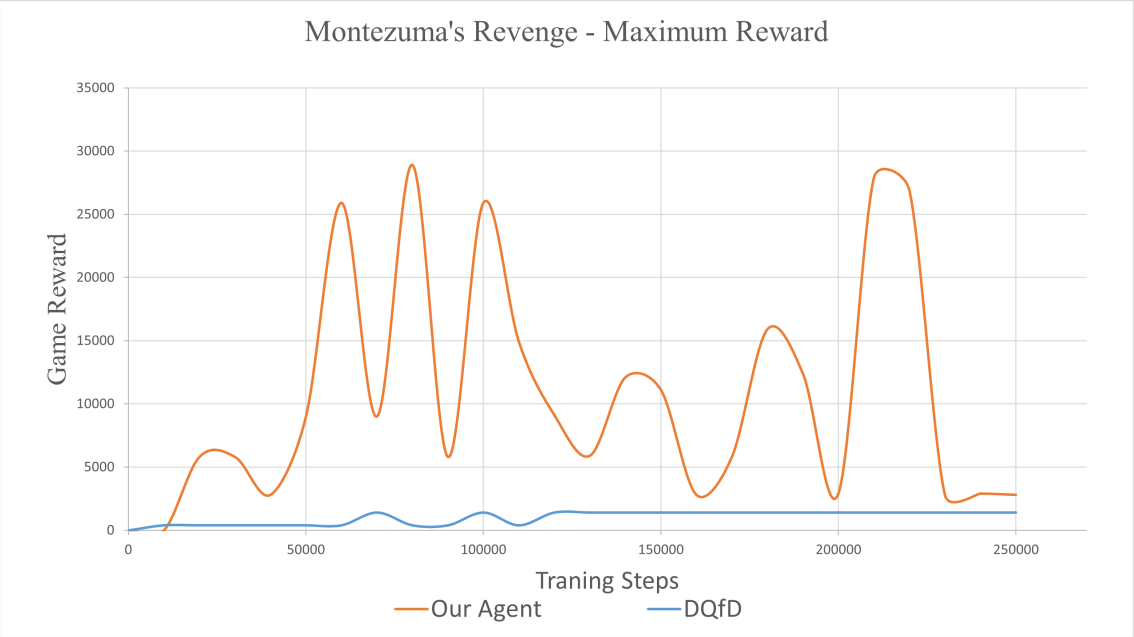
The last two games are Breakout and Pong games. These games are popular examples of simple environments. Most classical RL algorithms achieved a high score in both games. However, these algorithms required millions of training steps to reach these scores. In general, classical RL algorithms required 200 million steps to train their agents. On the other hand, our agent required 250 thousand steps which are very small training steps compared to classical RL agents. Moreover, the states of these two simple environments change depending on each action, and it is impossible to cover all these states with the demonstrations. Our agent's performance is acceptable compared to the time of training and demonstration size. In the Breakout game, our agent scored 31 as a maximum score after 240.000 steps of training. This weight is the best snapshot of our agent in the Breakout game. The agent required all this time to learn as much as possible from the demonstrations before scoring that score. The reason here is the variation between samples in the demonstration data. In the Pong game, our agent scored -10 (which means 11 scores from the total scores of 21) after 20.000 steps of training. This weight is the best snapshot of our agent in the Pong game. For the rest of the weights, the results varied between -18 and -10 scores. This game has an opponent player and the state relies on both our agent's action and the other player's action. Therefore, the results were varied and low.

Figure 5.9 is comparing the phase one results for both our agent and DQfD's agent that we trained on Montezuma's Revenge game. Both agents trained with the same period and tested on the same settings. Recall our agent results, the agent's maximum score is 28.900, and the maximum average score of ten episodes is 9570. For the DQfD's

agent, the agent's maximum score is 1400 which means that the agent was able to pass two or three rooms only, and the maximum average score of ten episodes is 1400.



(a)



(b)

Figure 5.9 Comparing phase one performance for both agents. (a) plot shows the average scores. (b) plot shows the maximum scores

Finally, Table 5.2 shows the comparison between DQN, DDQN, Prioritized DDQN, Dueling DDQN, Distributional DQN, Noisy DQN, Rainbow, DQfD, and our agent. To perform a fair comparison, we followed the same evaluating procedure of the published results that we took from [11]. The table is showing the raw scores for all games averaged over 200 episodes for the best snapshot obtained during the training. In each episode, the agent starts with 30 no-op steps. The 30 no-op is a random number between 1 and 30 of do-nothing action. The no-op is important to provide a random starting position for the agent. We report the published scores for DQN, DDQN, Prioritized DDQN, Dueling DDQN, Distributional DQN, Noisy DQN, and Rainbow. For DQfD and Our agents, we report our evaluations of the agents.

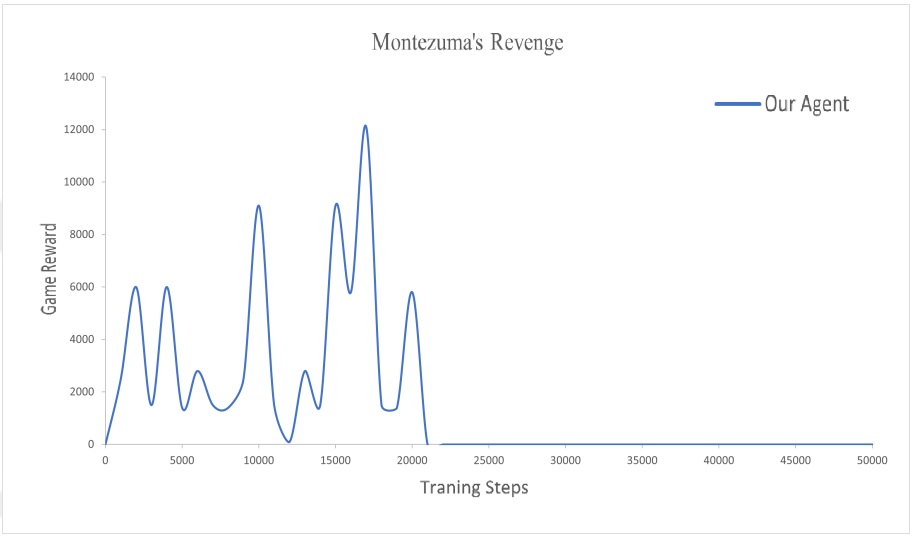
In hard exploration environments (Montezuma's Revenge, Pitfall, and Private Eye), our agent performance is better than all previous baselines with a small period of training time. The table shows the significant difference between the baselines results and our agent results. In simple environments, the performance of the baselines is excellent. Even these games were the main reason for the popularity of RL algorithms. Our agent performance is weak. Nevertheless, it is still better than the performance of the DQfD agent in both games Pong and Breakout.

Table 5.2 Comparing our agent phase one performance with the baseline performance

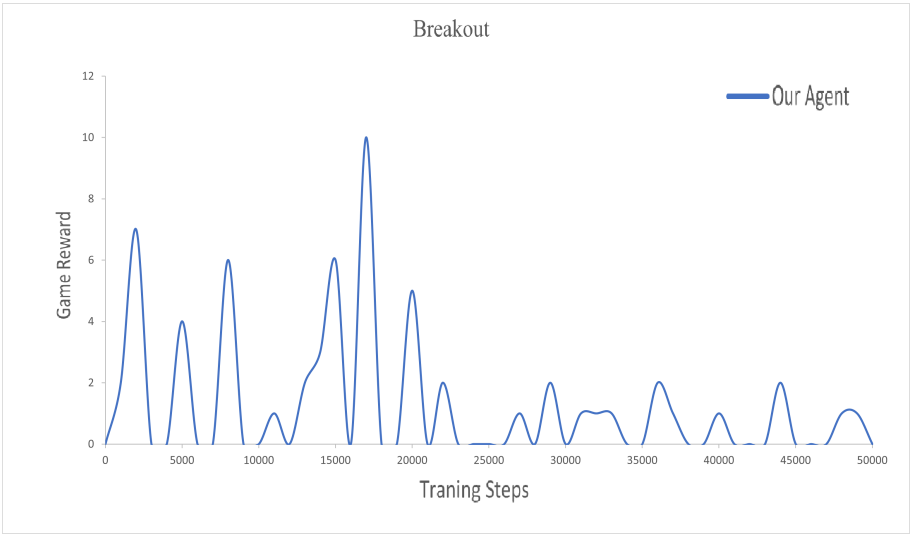
Game	Montezuma's Revenge	Pitfall	Private Eye	Pong	Breakout
DQN	0.0	-286	146	19.5	385.5
DDQN	0.0	-29.9	129	20.9	418.5
PDDQN	0.0	0.0	200	20.7	381.5
Duel DDQN	0.0	0.0	103	**21.0**	345.3
Distrib. DQN	384	-2.1	15172	20.8	**612.5**
Noisy DQN	206	-18.2	3966	**21.0**	459.1
Rainbow	384	0.0	4234	20.9	417.5
DQfD	206	265	8438	-19.5	1.99
Our Agent	**5035**	**2034.3**	**23152**	-14.5	2.39

5.2.2 Interacting Phase

The first interacting result we want to share is the direct agent performance after finishing the pre-training phase. These results are showing the effect of environment reward distribution and how our agent started losing its performance once the learning from the interaction is started. Figure 5.9 shows the agent performance on two games Montezuma's Revenge and Breakout. These games are a good example for both simple exploration environments (Breakout), and hard exploration environments (Montezuma's Revenge). In both games, the agent starts to learn from self-generated data after performing 20,000 steps to show the effect of reward distribution clearly.



(a)



(b)

Figure 5.10 The performance of our agent in phase two. (a) plots show online rewards of Montezuma's Revenge game. (b) plots show online rewards of the Breakout game

It is clear how the reward distribution harms the agent's performance. At the beginning of the interaction, the agent selects actions using the pre-trained policy. This policy enabled the agent to play and score efficiently in the first 20 thousand steps. During these steps, the agent is saving the new data as self-generated data. After the 20 thousand steps, the agent starts updating its network using the self-generated data. In the Breakout game, the agent scores are reduced, and in the Montezuma's Revenge game the scores become equal to zero.

The second part of interacting results is the Reward Function (RF) training and testing results. As we mentioned before, the RF trained with a new dataset had been extracted from the phase one demonstrations. The RF's convolutional neural network (CNN) is trained with 20.000 steps for each game. Later, the RF for each game was tested with a new dataset. This dataset differs from the training dataset. We built a new test dataset by playing each game for one episode. The idea behind building a new dataset is to find out if the reward function suffers from overfitting or underfitting. The details of the training dataset, test dataset, and test accuracy are shown in Table 5.3. Furthermore, we calculated the F1 score to measure the test accuracy. The F1 score results are shown in Table 5.4.

Table 5.3 Reward Function learning dataset details and test accuracy

Game	**Training Dataset**		**Test Dataset**		**Test Accuracy**
	Size	Episode	Size	Episode	
MR	60877	5	9705	1	0.8646
Pitfall	132844	5	25135	1	0.9148
Private Eye	39098	5	8282	1	0.8443
Breakout	31297	9	3282	1	0.6453
Pong	26783	3	6245	1	0.7969

Table 5.4 Reward Function learning F1 scores for each action

Accuracy	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**Weighted**
MR	0.86	0.63	0.91	0.88	0.85	0.85	0.78	0.74	0.86
Pitfall	0.93	0.64	0.78	0.93	0.	0.59	0.76	0.	0.91
Private Eye	0.41	0.	0.54	0.87	0.92	0.	0.55	0.57	0.83
Breakout	0.68	0.	0.29	0.29	-	-	-	-	0.56
Pong	0.87	0.64	0.51	0.6	0.	0.	-	-	0.81

The first table shows the test accuracy which represents the ratio of predicted values that match with actual values. For the hard exploration environments, the RF accuracies were between 84% and 91% which were a good percentage. For the simple environments, in which the state-space is infinite, the RF accuracies were between 64% and 79% which were an acceptable percentage. It is known that there are many classes each class represents one action in the datasets. To find the number of true predictions for each class, we calculated the F1 score. The F1 Score is a metric that represents the ratio between the exactness and completeness of the classifier. In another way, the ratio between Precision and Recall. The F1 Score value could be found by:

$$F1 = 2 * \frac{precision * recall}{precision + recall} \tag{5.1}$$

The last results are the performance of our agent with an integrated model in which the agent uses the self-generated data including the Inverse Reinforcement Learning (IRL) rewards. Our integrated approach has only been tested on two games: Montezuma's Revenge and Breakout. The reason that we tested it on two games is due to the training time that is required. For comparison, we trained our integrated agent, the integrated DQfD agent, and the standard Prioritized Double DQN (DDQN) agent on these two games. The standard PDDQN is the algorithm that we used to build our approach. The PDDQN agent differs from our agent because it does not have demonstration data, pre-training phase, and reward function. The PDDQN agent took about one week to finish the training for each game. The DQfD agent took about four days for each game, while our agent required 20 days to complete the training for each game. The reason for the difference in training time is due to the number of predications in the algorithms. In each step, our agent makes two predictions, the first prediction is selecting the correct action, and the second is predicting the reward for that action. On the other hand, the DQfD and PDDQN agents predict the correct action only. The difference in training time is the only disadvantage in our work. Figures 5.11 and 5.12. Showing the online results for the three algorithms in Montezuma's Revenge and Breakout games respectively.

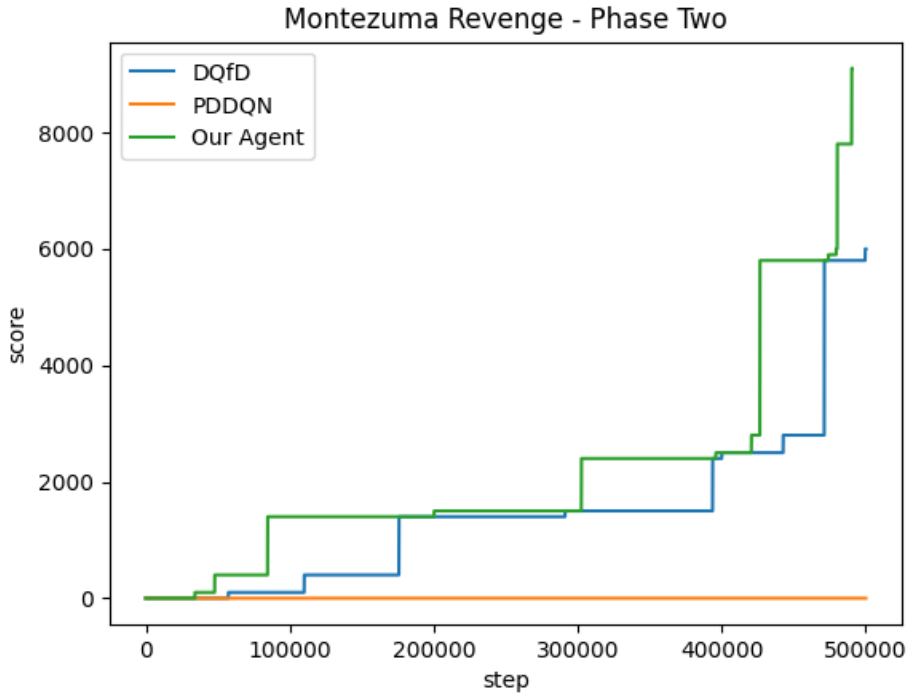


Figure 5.11 Online rewards for DQfD agent, standard PDDQN agent, and our agent on the Montezuma's Revenge game. We removed the zero reward between the episodes

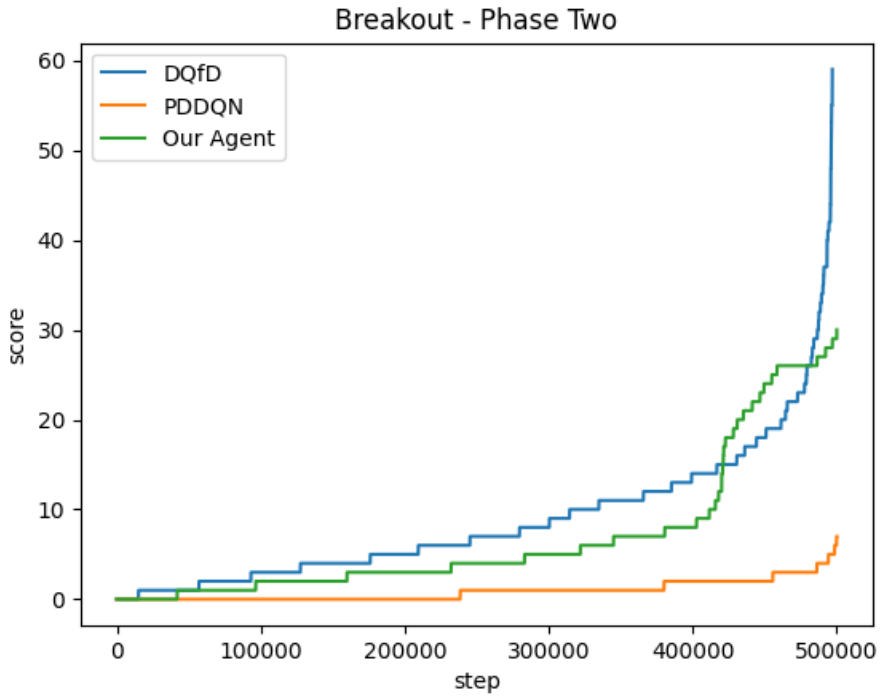


Figure 5.12 Online rewards for DQfD agent, standard PDDQN agent, and our agent on the Breakout game. We removed the zero reward between the episodes

As usual, our agent performance is better than the other agents on Montezuma's Revenge game. The RF rewards enable the agent to keep learning without decreasing performance. In the beginning, the agent starts learning from the self-generated data before reaching any reward. For this reason, in the beginning, the agent score is zero due to the change that occurs in the network weights. Our agent starts scoring faster than the DQfD agent and finishes the training with a higher score. The PDDQN's agent starts and finishes the training without scoring any positive score. For the Breakout game, our agent score is in the middle between the DQfD's and PDDQN's. In general, the PDDQN's agent is scoring higher than our agent in this game but after millions of training steps. For the first 500 thousand steps, our agent's performance is better and was able to score triple the PDDQN's score. The DQfD's agent was better than our agent from the beginning to the end. Except for the period between 425 thousand and 475 thousand in which our agent starts scoring higher than the DQfD's agent.

The last plot we want to show is the ratio between the demonstration and the self-generated data in the mini-batch. As we mentioned before, the DQfD agent trained its current network using both demonstrations data and self-generated data. We calculate the ratio for both games Montezuma's Revenge and Breakout as shown in Figure 5.13. Unlike our agent, the DQfD agent relies on demonstrations all-time of training. Furthermore, it uses the demonstrations only at the beginning of the training without returning to the self-generated data. Our agent uses the self-generated data only, which is the main concept in learning from interaction and makes it different from imitation learning.

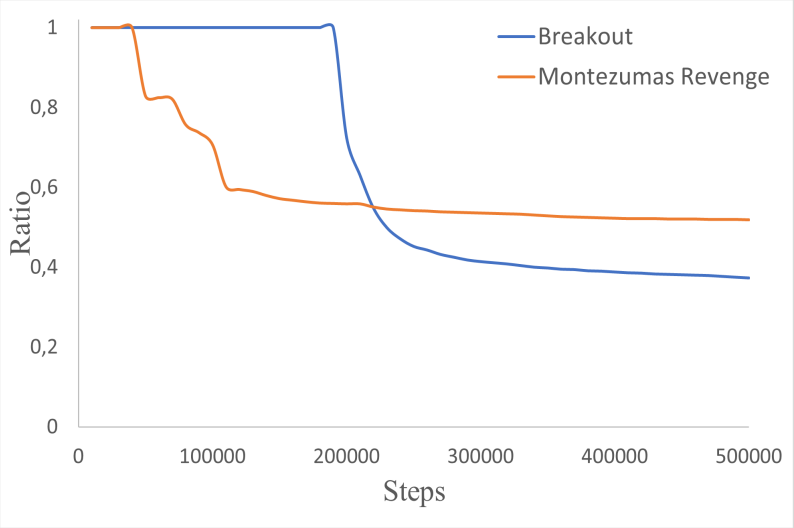


Figure 5.13 The ratio of how often the demonstration data was sampled versus the self-generated data in the DQfD algorithm while we trained it

6
RESULTS AND DISCUSSION

In this research, a new solution for enhancement Deep Reinforcement Learning algorithms has been suggested. A novel way of addressing the problem of rewards scarcity in hard exploration environments. The proposed research contributes an external reward system used independently or alongside the environment reward.

Our mechanism proposed a strategy of combining Reinforcement Learning, Learning from Demonstration, and Inverse Reinforcement Learning. Learning from demonstration helps us to reduce the learning duration and gives us the ability to employ external rewards. The only problem is the difference in the distribution between external rewards and environmental rewards. A Reward Function based on Inverse Reinforcement Learning principles is proposed for two reasons. The first one is to uniform the distribution of rewards between the training phases. The second reason is to enable the agent to interact with the environment to take the advantage of the environment's reward to discover new states beyond the demonstration states.

The training and testing were carried out on 5 different environments of both types hard exploration and simple environments. Experimental results show that the proposed method has better performance than the previous methods on hard exploration environments. Nevertheless, our mixed method has a little bit more time complexity since it contains extra predications. On the other hand, the performance in simple environments is not as efficient but still acceptable.

6.1 Suggestions

Although our system has achieved high success rates in its experimental results, there are still many improvements and experiences to make. These experiments and improvements can be summarized in the following points:

1. Training and testing our method on more games. The deep reinforcement learning community used to train their agents on 49 games. With a few improvements that can be added to the device used, we can train at least another five games besides the current games. Training these games will prove that our approach is reliable and flexible at the same time. Moreover, these improvements allow us to train the agent with larger memory and a longer training period which in turn improves the agent performance.

2. Using other techniques or algorithms. In this work, we have purposely used the simplest deep reinforcement learning approach to show up the efficiency of our enhancements. In the future, the system can be improved by using more complex techniques such as Dueling Deep Q-Network [7], Distributional Q-learning [9]. These techniques will raise the performance of our approach.

3. The last enhancement to be made is enabling the Reward Function to learn in parallel with the agent. In our approach, the agent interacts with the environment and learns from self-generated data. During the interaction phase, the agent evaluates its behavior depending on incoming rewards from the environment and the pre-trained reward function. The weights of this function never change during the interaction phase. One of the possible enhancements is updating the network weights using the agent's self-generated data. In this way, the agent continues to benefit from the external rewards and the reward function learns from more data that covers states not found in demonstrations.

REFERENCES

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[2] 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*, vol. 518, no. 7540, pp. 529–533, 2015.

[3] C. J. C. H. Watkins, "Learning from delayed rewards," 1989.

[4] H. Hasselt, "Double q-learning," *Advances in neural information processing systems*, vol. 23, pp. 2613–2621, 2010.

[5] H. Van Hasselt, A. Guez, D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, 2016.

[6] T. Schaul, J. Quan, I. Antonoglou, D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[7] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, N. Freitas, "Dueling network architectures for deep reinforcement learning," in *International conference on machine learning*, PMLR, 2016, pp. 1995–2003.

[8] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*, PMLR, 2016, pp. 1928–1937.

[9] M. G. Bellemare, W. Dabney, R. Munos, "A distributional perspective on reinforcement learning," in *International Conference on Machine Learning*, PMLR, 2017, pp. 449–458.

[10] M. Fortunato, M. G. Azar, B. Piot, J. Menick, I. Osband, A. Graves, V. Mnih, R. Munos, D. Hassabis, O. Pietquin, *et al.*, "Noisy networks for exploration," *arXiv preprint arXiv:1706.10295*, 2017.

[11] M. Hessel, J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, D. Silver, "Rainbow: Combining improvements in deep reinforcement learning," in *Thirty-second AAAI conference on artificial intelligence*, 2018.

[12] T. Hester, M. Vecerik, O. Pietquin, M. Lanctot, T. Schaul, B. Piot, D. Horgan, J. Quan, A. Sendonaris, I. Osband, *et al.*, "Deep q-learning from demonstrations," in *Thirty-second AAAI conference on artificial intelligence*, 2018.

[13] Y. Aytar, T. Pfaff, D. Budden, T. L. Paine, Z. Wang, N. de Freitas, "Playing hard exploration games by watching youtube," *arXiv preprint arXiv:1805.11592*, 2018.

[14] T. Pohlen, B. Piot, T. Hester, M. G. Azar, D. Horgan, D. Budden, G. Barth-Maron, H. Van Hasselt, J. Quan, M. Večerık, *et al.*, "Observe and look further: Achieving consistent performance on atari," *arXiv preprint arXiv:1805.11593*, 2018.

[15] T. Salimans, R. Chen, "Learning montezuma's revenge from a single demonstration," *arXiv preprint arXiv:1812.03381*, 2018.

[16] A. Ecoffet, J. Huizinga, J. Lehman, K. O. Stanley, J. Clune, "Go-explore: A new approach for hard-exploration problems," *arXiv preprint arXiv:1901.10995*, 2019.

[17] A. Y. Ng, S. J. Russell, *et al.*, "Algorithms for inverse reinforcement learning.," in *Icml*, vol. 1, 2000, p. 2.

[18] P. Abbeel, A. Y. Ng, "Apprenticeship learning via inverse reinforcement learning," in *Proceedings of the twenty-first international conference on Machine learning*, 2004, p. 1.

[19] D. Ramachandran, E. Amir, "Bayesian inverse reinforcement learning.," in *IJCAI*, vol. 7, 2007, pp. 2586–2591.

[20] Y. Tassa, T. Erez, E. Todorov, "Synthesis and stabilization of complex behaviors through online trajectory optimization," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, IEEE, 2012, pp. 4906–4913.

[21] E. Uchibe, "Model-free deep inverse reinforcement learning by logistic regression," *Neural Processing Letters*, vol. 47, no. 3, pp. 891–905, 2018.

[22] A. Tucker, A. Gleave, S. Russell, "Inverse reinforcement learning for video games," *arXiv preprint arXiv:1810.10593*, 2018.

[23] J. Fu, K. Luo, S. Levine, "Learning robust rewards with adversarial inverse reinforcement learning," *arXiv preprint arXiv:1710.11248*, 2017.

[24] T. M. Mitchell, *Machine Learning*. McGraw Hill, 1997.

[25] M. Lapan, *Deep Reinforcement Learning Hands-On: Apply modern RL methods, with deep Q-networks, value iteration, policy gradients, TRPO, AlphaGo Zero and more*. Packt Publishing Ltd, 2018.

[26] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, Y. Bengio, "Generative adversarial nets," *Advances in neural information processing systems*, vol. 27, 2014.

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

[28] X. X. Wang, "Deep reinforcement learning:case study with standard rl testing domains," 2016.

[29] R. Bellman, "A markovian decision process," *Journal of mathematics and mechanics*, vol. 6, no. 5, pp. 679–684, 1957.

[30] C. J. Watkins, P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[31] M. Hamadouche, C. Dezan, D. Espes, K. Branco, "Comparison of value iteration, policy iteration and q-learning for solving decision-making problems," in *2021 International Conference on Unmanned Aircraft Systems (ICUAS)*, IEEE, 2021, pp. 101–110.

[32] Y. LeCun *et al.*, "Generalization and network design strategies," *Connectionism in perspective*, vol. 19, pp. 143–155, 1989.

[33] S. Albelwi, A. Mahmood, "A framework for designing the architectures of deep convolutional neural networks," *Entropy*, vol. 19, no. 6, p. 242, 2017.

[34] A. Baraldi, D. Tiede, "Autocloud+, a "universal" physical and statistical model-based 2d spatial topology-preserving software for cloud/cloud–shadow detection in multi-sensor single-date earth observation multi-spectral imagery—part 1: Systematic esa eo level 2 product generation at the ground segment as broad context," *ISPRS International Journal of Geo-Information*, vol. 7, no. 12, p. 457, 2018.

[35] I. Goodfellow, Y. Bengio, A. Courville, *Deep learning*. MIT press, 2016.

[36] V. François-Lavet, P. Henderson, R. Islam, M. G. Bellemare, J. Pineau, "An introduction to deep reinforcement learning," *arXiv preprint arXiv:1811.12560*, 2018.

[37] M. G. Bellemare, Y. Naddaf, J. Veness, M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *Journal of Artificial Intelligence Research*, vol. 47, pp. 253–279, 2013.

[38] Y. Bengio, *Learning deep architectures for AI*. Now Publishers Inc, 2009.

[39] A. Krizhevsky, I. Sutskever, G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[40] G. E. Hinton, R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *science*, vol. 313, no. 5786, pp. 504–507, 2006.

[41] J. Wang, L. Gou, H.-W. Shen, H. Yang, "Dqnviz: A visual analytics approach to understand deep q-networks," *IEEE transactions on visualization and computer graphics*, vol. 25, no. 1, pp. 288–298, 2018.

[42] K. Jarrett, K. Kavukcuoglu, M. Ranzato, Y. LeCun, "What is the best multi-stage architecture for object recognition?" In *2009 IEEE 12th international conference on computer vision*, IEEE, 2009, pp. 2146–2153.

[43] V. Nair, G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Icml*, 2010.

[44] L.-J. Lin, "Reinforcement learning for robots using neural networks (no. cmu-cs-93-103)," *Carnegie-Mellon Univ Pittsburgh PA School of Computer Science*, 1993.

[45] G. Hinton, N. Srivastava, K. Swersky, "Neural networks for machine learning lecture 6a overview of mini-batch gradient descent," *Cited on*, vol. 14, no. 8, p. 2, 2012.

[46] B. Piot, M. Geist, O. Pietquin, "Boosted bellman residual minimization handling expert demonstrations," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, Springer, 2014, pp. 549–564.

[47] S. Russell, "Learning agents for uncertain environments," in *Proceedings of the eleventh annual conference on Computational learning theory*, 1998, pp. 101–103.

[48] S. Arora, P. Doshi, "A survey of inverse reinforcement learning: Challenges, methods and progress," *Artificial Intelligence*, p. 103 500, 2021.

[49] N. D. Ratliff, J. A. Bagnell, M. A. Zinkevich, "Maximum margin planning," in *Proceedings of the 23rd international conference on Machine learning*, 2006, pp. 729–736.

[50] G. Neu, C. Szepesvári, "Apprenticeship learning using inverse reinforcement learning and gradient methods," *arXiv preprint arXiv:1206.5264*, 2012.

[51] E. T. Jaynes, "Information theory and statistical mechanics," *Physical review*, vol. 106, no. 4, p. 620, 1957.

[52] B. D. Ziebart, A. L. Maas, J. A. Bagnell, A. K. Dey, *et al.*, "Maximum entropy inverse reinforcement learning.," in *Aaai*, Chicago, IL, USA, vol. 8, 2008, pp. 1433–1438.

[53] A. Boularias, J. Kober, J. Peters, "Relative entropy inverse reinforcement learning," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, JMLR Workshop and Conference Proceedings, 2011, pp. 182–189.

[54] J. Choi, K.-E. Kim, "Map inference for bayesian inverse reinforcement learning.," in *NIPS*, 2011, pp. 1989–1997.

[55] M. Lopes, F. Melo, L. Montesano, "Active learning for reward estimation in inverse reinforcement learning," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, Springer, 2009, pp. 31–46.

[56] S. Levine, Z. Popovic, V. Koltun, "Nonlinear inverse reinforcement learning with gaussian processes," *Advances in neural information processing systems*, vol. 24, pp. 19–27, 2011.

[57] M. Babes, V. N. Marivate, K. Subramanian, M. L. Littman, "Apprenticeship learning about multiple intentions," in *ICML*, 2011.

[58] E. Klein, M. Geist, B. Piot, O. Pietquin, "Inverse reinforcement learning through structured classification," *Advances in neural information processing systems*, vol. 25, 2012.

[59] B. Taskar, V. Chatalbashev, D. Koller, C. Guestrin, "Learning structured prediction models: A large margin approach," in *Proceedings of the 22nd international conference on Machine learning*, 2005, pp. 896–903.

[60] E. Klein, B. Piot, M. Geist, O. Pietquin, "A cascaded supervised learning approach to inverse reinforcement learning," in *Joint European conference on machine learning and knowledge discovery in databases*, Springer, 2013, pp. 1–16.

[61] D. Brown, W. Goo, P. Nagarajan, S. Niekum, "Extrapolating beyond suboptimal demonstrations via inverse reinforcement learning from observations," in *International conference on machine learning*, PMLR, 2019, pp. 783–792.

[62] S. Levine, Z. Popovic, V. Koltun, "Feature construction for inverse reinforcement learning.," in *NIPS*, Citeseer, vol. 23, 2010, p. 1342.

PUBLICATIONS FROM THE THESIS

Papers

1. W. Z. Tareq, M. F. Amasyali, "A New Reward System Based on Human Demonstrations for Hard Exploration Games", CMC-Computers, Materials Continua, vol. 70, no. 2, PP. 2401–2414, 2022.

Conference Papers

1. W. Z. Tareq, M. F. Amasyali, "Merging Reinforcement Learning and Inverse Reinforcement Learning via Auxiliary Reward", The 4th International Conference on Artificial Intelligence in Information and Communication System (ICAIIC2022), Jeju Island, Korea, 2022, PP. 292–297.

Awards

1. "Best Paper Award", The 4th International Conference on Artificial Intelligence in Information and Communication System (ICAIIC2022), Jeju Island, Korea, 2022.