Shahid Chamran University of Ahvaz

Department of Computer Engineering

In Partial Fulfillment of the Requirements for the Degree
Bachelor Computer Engineering

Playing Video Games with Reinforcement Learning

by

Arshia Moradi

Supervised by

Assistant Professor Dr.Mohammad Javad Rashti

December 2023

Abstract

This thesis investigates the implementation, challenges, and capabilities of deep Q-network (DQN) agents in the context of playing Atari games. The study explores the foundational concepts of reinforcement learning, Q-learning, and neural networks, laying the groundwork for a comprehensive understanding of DQN agents. A detailed literature review surveys existing research, highlighting advancements, challenges, and applications in reinforcement learning.

We explored the capabilities and limitations of DQN agents, emphasizing their success in controlled gaming environments and acknowledging challenges in more complex scenarios. The study serves as a foundation for further exploration into policy gradient methods, distributional reinforcement learning, and the adaptation of strategies for more intricate environments.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Games have always been an integral part of our lives, from childhood activities like tag to engaging in board games such as Monopoly or Chess with friends and family. The act of play stands as a fundamental aspect of the human experience. In the late 20th century, video games emerged, bringing about a significant change. The introduction of video games allowed us to create intricate environments that demand strategic thinking for success.

In recent years we've witnessed the development of intelligent agents capable of learning how to play these video games. Several studies have demonstrated that leveraging reinforcement learning (RL), a subset of machine learning algorithms, enables the creation of self-learning agents proficient in playing and learning strategies within complex environments.

In reinforcement learning, our agent learns through trial and error. Each action taken by the agent receives a reward from the environment. The goal is to find a set of strategies[1] that maximize the received reward[18]. In supervised machine learning, data and corresponding correct labels are necessary. However, in reinforcement learning, we only require knowledge of the reward and action mechanism of the environment.

Video games offer an intriguing environment for testing and developing algorithms. By simulating real-world problems in video games, we can optimize decision-making processes, enhance problem-solving skills, and refine algorithms for real-world applications. The dynamic and interactive nature of video games provides a controlled yet complex space to experiment with different strategies and approaches.

Furthermore, the challenges presented in video games span a wide spectrum, from strategic decision-making to rapid response scenarios. This versatility makes video games a valuable playground for testing the adaptability and robustness of algorithms across various problem domains.

1.1 Motivation

We have always found great joy in playing video games and were intrigued by the prospect of employing artificial intelligence to enhance the gaming experience. Two pivotal elements in crafting a good game are adept-level design[3] and compelling NPCs[2]. Leveraging reinforcement learning agents, we can infuse games with more engaging and lifelike NPCs, enhancing the overall player experience.

Furthermore, the idea of trying to learn how to play from other players seemed intriguing. By learning how someone plays a game, we discover a part of their consciousness. Collecting sufficient data from a player's interactions with a game allows us to immortalize and preserve that unique aspect of their life.

[1] Later referred to as a policy
[2] Non-Playable Characters

1.2 Objectives

The primary objective of this research is to explore reinforcement learning algorithms and examine their applicability in playing simple video games. The overarching goal is to train a basic reinforcement learning (RL) agent to navigate pixel-style video games, with a focus on Atari 2600 games, as highlighted in recent literature. The subsequent aim is to assess the agent's potential in learning and adapting to more complex game environments. To achieve this overarching goal, the research seeks to address the following key questions:

1. What is reinforcement learning, and what are the fundamental algorithms used in this field?
2. What are the inherent limitations of reinforcement learning?
3. In what ways can reinforcement learning algorithms be optimized to enhance their performance and adaptability?
4. How can RL algorithms be leveraged to emulate and reproduce the performance of another player effectively?

1.3 Report Structure

This report is organized in the following manner: Chapter 2 establishes the foundational knowledge related to reinforcement learning and deep Q-networks. Chapter 3 surveys existing literature and research on DQN agents, focusing on advancements, challenges, and applications. The chapter highlights key studies that have contributed to the understanding of reinforcement learning agents, especially in the context of playing games. Chapter 4 details our implementation of the reinforcement learning agent based on the concepts presented in this report. Chapter 5 analyzes and discusses the findings from the implemented agent. Finlay chapter 6 summarizes the key findings and insights from the research.

Chapter 2

Theoretical Background

2.1 Markov Decision Process

A Markov Decision Process (MDP) is a mathematical framework for modeling decision-making, where an agent in a specific state selects actions that transition the system to a new state with associated probabilities. The agent receives rewards based on the chosen action and aims to discover a policy that maximizes the expected cumulative reward over time.[11].

A 4-tuple represents the Markov Decision Process as (S, A, P_a, R_a), where S denotes the set of states representing the environment. The action space, denoted as A, encompasses all possible actions an agent can take in any state. Each specific action, denoted as $a \in A$, determines the probability, P_a, of transitioning from state $s \in S$ to the next state s'. The immediate reward, denoted as $R_a(s, s')$, is the reward received immediately after taking action a in state s and transitioning to state s'.

2.1.1 Markov Property

The Markov Property states that the future state in a stochastic process depends solely on the current state and the action taken, independent of the sequence of events that preceded it[18].

The interactions between the environment is a sequence of state, action, and rewards

$$\langle S_0, A_0, R_1, S_1, A_1, R_2, \ldots \rangle$$

If the chain satisfies the Markov property, it is a Markov decision process. Most reinforcement learning algorithms need the Markov Property as a prerequisite.

2.2 Reinforcement Learning Components

2.2.1 Environment

The environment is the world our agent interacts with. It receives the agent's actions as input and returns a new state with the associated rewards. Whether the new state differs from the previous one depends on the specific action taken.[18].

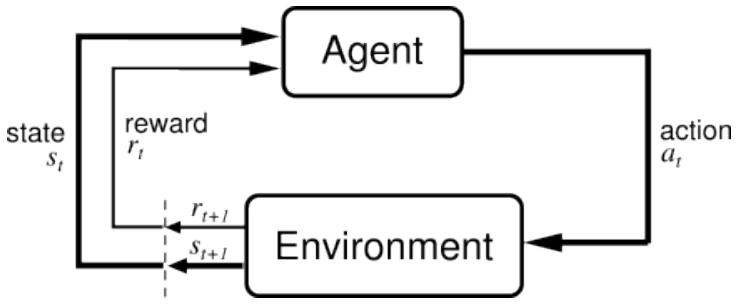


Figure 2.1: A representation of the reinforcement learning model [18]

2.2.2 States

States (S) represent the various configurations or situations in the environment. In the context of Atari2600 games, the states are the frames of each game. Another example is a robot equipped with sensors capable of detecting its proximity to a wall.

There are differences between the two previous examples: in Atari2600 games, frames offer a direct representation of the environment, whereas the robot's states are not a direct reflection of reality; instead, they are a constructed reality based on its sensor inputs.

Another key difference is the Markov Property. In the case of the robot, the state may show that it is in a hallway, but the exact location within the hallway remains uncertain. We may consider adding the history of observations into the state representation. This approach can enhance the Markovian nature of the environment, but determining the optimal level of historical information to include poses computational and storage challenges.

2.2.3 Transition Model

The transition model is the representation of how the world changes in response to the agent's action. In a Markov setting, it's shown as a distribution over the next state $P(s|s', a)$. The transition model might be wrong or incomplete, but the agent can use it to learn about the environment.

2.2.4 Policy

A policy π is a mapping from states to action, defined mathematically as a function with state s as input value and action $a \in A$ as output:

$$\pi : \ S \to A$$

The policy dictates how the agent selects actions. In stochastic environments, the policy is defined as the distribution of actions over states $\pi(a|s)$.

2.3 Value Function

The value function in reinforcement learning represents the expected sum of rewards the agent receives while interacting with the environment. The discounted sum of rewards, denoted as G_t, at time step t is defined by:

$$\begin{aligned} G_t &= r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots + \gamma^T r_T \\ &= \Sigma_{t'=t}^{T} \gamma^{t'-t} r_{t'} \end{aligned} \tag{2.1}$$

Here, r_t is the reward at time step t, and T is the time step at which the sequence terminates. The discount factor γ trades off the importance of immediate and future rewards, falling within $[0, 1]$. Agent's

goal is to maximize G_t by finding an optimal policy. There are two primary types of value function: the state value function and the action value function.

2.3.1 State Value Function

The state value function $V_\pi(s)$ represents the expected cumulative reward the agent can get from state s onward by following the policy π. Mathematically, it is defined as the expected value of the sum of discounted rewards:

$$\begin{aligned} V_\pi(s) &= \mathbb{E}_\pi \left[G_t | S_t = s \right] \\ V_\pi(s) &= \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_{t+1} | S_0 = s \right] \end{aligned} \tag{2.2}$$

2.3.2 Action Value Function

The action value function $Q_\pi(s, a)$ represents the cumulative reward when starting at state s, taking action a, and following policy π. It is defined as:

$$Q_\pi(s, a) = \mathbb{E}_\pi \left[G_t | S_t = s, A_t = a \right] \tag{2.3}$$

2.3.3 Advantage Function

The advantage function in reinforcement learning represents the advantage of taking a specific action in a given state compared to the average or expected value of all actions in that state.

The advantage function is the difference between the action value function $Q_\pi(s, a)$, and the state value function $V_\pi(s)$:

$$A(s, a) = Q_\pi(s, a) - V_\pi(s) \tag{2.4}$$

2.3.4 Bellman Equation

A Bellman equation is a mathematical equation that expresses a relationship between the value of a decision or action and the expected cumulative rewards associated with that decision[18].

By using the Bellman equation, we can calculate the value of the state s_1 based on another state s_2. Using this, we can find the best policy without calculating the value functions for each state. We can show the Bellman equation for both state value and action value functions:

$$\begin{aligned} V_\pi(s) &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma V_\pi(s')] \\ Q_\pi(s, a) &= \sum_{s',r} p(s', r|s, a)[r + \gamma \sum_{a'} \pi(a'|s') Q_\pi(s', a')] \end{aligned} \tag{2.5}$$

Here $\pi(a|s)$ is the distribution of actions over states and $p(s', r|s, a)$ is the probability of going to state s' from state s with action a and reward r.

2.4 Reinforcement Learning Algorithms

There are a vast number of reinforcement learning algorithms. These algorithms can be broadly categorized into two main types: value iteration methods and policy gradient methods.

2.4.1 Value Iteration

Value iteration methods are a class of reinforcement learning algorithms designed for iteratively updating the value functions associated with states or state-action pairs. The core concept involves estimating the expected cumulative rewards associated with different actions in various states. This estimation is called the target value.

When the agent selects action a in state s, the approximated action value $Q(s, a)$ undergoes an update towards a target value. To ensure the exploration of most states, an ϵ-greedy policy is used. This policy introduces stochasticity to decision-making, allowing for more effective exploration of the state space. The ϵ-greedy policy is expressed:

$$action = \begin{cases} \arg\max_{a \in A(s)} Q(s, a) & \text{with probability } 1 - \epsilon, \\ \text{random } a \in A(s) & \text{with probability } \epsilon. \end{cases} \tag{2.6}$$

Monte Carlo methods, Temporal Difference learning, and Q-learning are prominent examples falling under the umbrella of value iteration. The main difference between these algorithms is how and when they update their target values.

Monte Carlo

In Monte Carlo (MC) method, the approximated value is calculated when the sequence is terminated. It means that Monte Carlo methods only work in episodic Markov decision processes. At the end of each episode, the approximated values for the actions and states get updated towards the actual reward received.

$$\text{MC}_{\text{target}} = \sum_{t=0}^{T} \gamma^t r_t$$

This means that MC updates occur less frequently, and the method assumes that all states contribute equally to the final reward. Monte Carlo methods often perform well even when the Markov property is not strictly satisfied.

Temporal Difference Learning

Temporal difference learning (TD) combines Monte Carlo method with concepts of dynamic programming. The value function is updated at each time step, and it doesn't require the termination of an episode. The update is based on the difference between the current estimate and a target value, which is a combination of the immediate reward and the estimated value of the next state. This method enables learning from incomplete sequences:

$$\text{TD}_{\text{target}} = r_{t+1} + \gamma Q(s_{t+1}, a_{t+1})$$

This form of temporal difference learning is called TD(0) because we only use the next state. In Temporal Difference with N-Step Bootstrapping, referred to as TD(N), the learning agent updates its estimates of state values not just based on immediate rewards and the estimated value of the next state (as in standard TD), but it also considers the values of states N steps ahead in the future. This allows for a balance between the bias-variance trade-off.

Q-learning

Q-learning is a model-free reinforcement learning algorithm that falls under the category of temporal difference methods. It aims to learn the optimal action-value function, denoted as $Q(s, a)$, by iteratively updating the Q-values based on observed experiences.

Using the Bellman Equation 2.5 properties we can say that if we have the optimal value $Q^*(s', a')$ for the next state s' over all possible actions A, the optimal strategy is to select action a', maximizing the expected value of the update target $r + \gamma Q^*(s', a')$[7]. After updating the action value functions iteratively, the algorithm converges to the optimal value function as $i \leftarrow \infty$[18].

Algorithm 1: Q-learning Algorithm

Data: Initialize $Q(S_t, A_t)$
for *every episode* **do**
  Observe state S_t;
  while S_t *is not a terminal state* **do**
    Select action A_t and evaluate Q;
    Take action a;
    Observe r, S_{t+1};
    $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[r + \gamma \max_{a'} Q(S_{t+1}, a') - Q(S_t, A_t)\right]$;
    $S_t \leftarrow S_{t+1}$;
  end
end

Calculating the true values for $Q^*(s, a)$ is impractical, instead we use an approximation to estimate the action value functions based on a set of features θ, $Q(s, a, ; \theta) \approx Q^*(s, a)$. We call these value function approximation.

This approximation can be done using a linear function approximator, but sometimes we use a non-linear function approximator such as a neural network[7].

2.4.2 Policy Gradient Methods

Policy gradient methods are a class of reinforcement learning algorithms that directly learn a policy to maximize the expected cumulative reward, Equation 2.1. Unlike value-based methods, which aim to estimate the value function, policy gradient methods focus on learning the policy itself.

We denote the policy represented as a parameterized function $\pi_\theta(a|s)$ where θ represents the parameters of the policy and $\pi(a|s)$ is the distribution of actions over states. The goal of policy gradient methods is to find the optimal parameters θ^* that maximize the expected cumulative reward.

The general form of the policy gradient is given by:

$$\nabla J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s).Q^{\pi_\theta(s,a)}\right] \tag{2.7}$$

REINFORCE: Monte Carlo Policy Gradient

REINFORCE can be seen as the policy gradient version of Monte Carlo value iteration. In this method, we use the Monte Carlo estimates of expected return to iteratively refine the policy. The update rule involves scaling the logarithm of the probability of the taken action by the associated cumulative reward. However, the reliance on Monte Carlo estimates introduces challenges related to high variance, potentially leading to sluggish convergence.

$$\nabla J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t).G_t\right] \tag{2.8}$$

Actor-Critic Methods

Actor-critic methods combine policy gradient with a value function estimate to reduce variance. The actor updates the policy using the advantage function $A^{\pi_\theta}(s_t, a_t)$ representing the difference between the

estimated value and the actual return.

$$\nabla J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t).A^{\pi_\theta}(s_t, a_t)\right] \tag{2.9}$$

Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) is a policy optimization algorithm in reinforcement learning designed to address some of the challenges associated with traditional policy gradient methods. PPO optimizes the policy while constraining the size of policy updates, introducing a surrogate objective function with a clipping term. The objective is to balance the trade-off between exploration and exploitation.

$$\text{maximize } J(\theta) = \mathbb{E}_{\pi_\theta}\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \cdot A^{\pi_{\theta_{\text{old}}}}(s_t, a_t) - \beta \cdot \text{clip}\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}, 1-\epsilon, 1+\epsilon\right) \cdot A^{\pi_\theta}(s_t, a_t)\right] \tag{2.10}$$

2.5 Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) is an interdisciplinary field that combines principles from artificial intelligence (AI), machine learning, and neural networks to empower agents with the ability to learn and make decisions in complex environments. Unlike traditional reinforcement learning approaches, which often rely on handcrafted features or state representations, DRL leverages the representation learning capabilities of deep neural networks to automatically discover relevant features from raw data. This enables DRL agents to tackle high-dimensional and unstructured input spaces, such as images or raw sensor data, making them particularly adept at tasks like playing video games, robotic control, and more.

At the core of DRL lies the integration of deep neural networks, typically deep artificial neural networks, with reinforcement learning algorithms. These networks, often referred to as Deep Q Networks (DQN) or policy networks, learn to approximate the value functions or policies crucial for decision-making in reinforcement learning settings.

One of the pioneering achievements in DRL is the success of algorithms like Deep Q Network (DQN) in mastering a variety of Atari 2600 games directly from raw pixel inputs[7]. This breakthrough showcased the potential of combining deep learning techniques with reinforcement learning, sparking widespread interest and exploration within the research community. Since then, DRL has evolved, giving rise to various advancements, including algorithms like Trust Region Policy Optimization (TRPO)[14], Proximal Policy Optimization (PPO)[13], and the impressive achievements of deep reinforcement learning in areas like autonomous vehicles, natural language processing, and robotics.

2.5.1 Deep Q Networks

The idea behind deep Q-learning is to use a neural network to find the best features for the value function approximator. We refer to this function approximator with weights θ, Q-network, denoted as $Q(s, a, \theta)$[7]. To train the network we minimize the loss function $L_i(\theta_i)$ for each iteration i:

$$L_i(\theta_i) = \mathbb{E}_{s,a,r,s'}\left[\left(y_i - Q(s, a, \theta_i)\right)^2\right] \tag{2.11}$$

where y_i is the target of for the update:

$$y_i = r + \gamma \text{max}_{a'} Q(s', a'; \theta_i^-) \tag{2.12}$$

One innovation that was proposed by [7, 5] is instead of using the weights of the network we freeze the parameters of the network for several iterations and use this second network for calculating the

Q-targets. That's why in Equation 2.12 we use θ^- as the network weight. We update the online network $Q(s, a, \theta)$ after each iteration using gradient descent:

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a,r,s'} \left[(y_i - Q(s, a; \theta_i)) \nabla_{\theta_i} Q(s, a; \theta_i) \right] \tag{2.13}$$

Another key point in [7] was the use of experience replay. We store the agent's experience at each time step t, $e_t = (s_t, a_t, r_t, s_{t+1})$ in a data set $D = \langle e_1, \ldots, e_N \rangle$. We call D the replay memory. During the training of the Q-network, instead of just using the current experience, we sample from the replay memory at random. Then we calculate the loss function for each episode and perform the gradient descent.

Algorithm 2: Deep Q-Learning with Experience Replay

Data: Neural network parameters θ, replay buffer D
Result: Trained Q-network

```
Initialize Q-network with parameters θ
Initialize replay buffer D
for each episode do
    Observe initial state s
    while s is not terminal do
        Select action a using ϵ-greedy policy
        Execute action a, observe reward r and next state s'
        Store (s, a, r, s') in replay buffer
        Sample mini-batch from replay buffer
        Calculate temporal difference error and update Q-network parameters
        Update target network parameters periodically
        s ← s'
    end
end
```

Double Deep Q-Learning

Double Deep Q-Learning is an extension of deep Q-learning that addresses the overestimation bias in Q-values. The method introduces a second neural network, often referred to as the target network, to estimate the Q-values for target calculation. This method was proposed by [5]. This helps stabilize training and mitigate overestimation issues. The new DoubleDQN target is:

$$y_t = r_{t+1} + \gamma Q(s_{t+1}, \text{argmax}_a Q\left(s_{t+1}, a; \theta_t\right), \theta_t^-) \tag{2.14}$$

Prioritized Experience Replay

One of the challenges in training a deep Q-network is the efficient use of the replay memory. The traditional experience replay mechanism, as introduced by[7] samples experiences uniformly from a replay buffer during the training process. However, not all experiences contribute equally to the learning process. We want to use samples that have more information more frequently.

The concept of Prioritized Experience Replay[12] (PRE) was introduced to enhance learning efficiency. This method assigns different priorities to experiences based on their temporal difference (TD) errors, emphasizing the experiences that lead to higher learning progress.

The TD error for a particular experience is the discrepancy between the predicted Q-value and the target Q-value. Experiences with larger TD errors indicate a greater learning potential. In PER, these experiences are prioritized for inclusion in the training process, allowing the agent to focus on the most informative transitions.

The prioritization of experiences is achieved by assigning a priority value to each experience in the replay buffer. The priority is computed as a function of the TD error, with higher TD errors leading to higher priorities.

$$\begin{aligned} p(i) &= |\mathbf{TD}_{\text{error}}(i)| \\ P(i) &= \frac{p_i^\alpha}{\Sigma_k p_k^\alpha} \end{aligned} \tag{2.15}$$

Where $p(i)$ is the priority of experience i, α is a hyperparameter that controls the degree of prioritization, and $P(i)$ is the probability of selecting the experience i.

When prioritized experiences are sampled for training, a bias is introduced due to the uneven distribution of priorities. To address this, PER employs importance sampling weights to adjust the contribution of each sampled experience to the weight updates. The weight assigned to a particular experience is given by:

$$w(i) = \left(\frac{1}{N} . \frac{1}{P(i)}\right)^\beta \tag{2.16}$$

Here N is the total number of experiences in the replay buffer, and β is a hyperparameter that controls the extent of the weight correction.

An important detail highlighted by [12] was the process of annealing the bias in prioritized experience replay. This addresses the bias introduced during the sampling process of prioritized experiences. The bias arises because experiences are sampled with probabilities proportional to their priorities, potentially causing a mismatch between the sampled experiences and the true underlying distribution. To mitigate this bias, an annealing scheme is often employed.

In this context, "annealing" refers to gradually reducing the extent of prioritization over the course of training. Initially, the prioritization is high, emphasizing the importance of experiences with large temporal difference (TD) errors. To implement annealing, a parameter β is introduced, which controls the extent of the bias correction. We start with an initial value of β_0 and gradually increase β over time until reaching $\beta = 1$ at the end of the learning process. This gradual increase in β helps strike a balance between exploration and exploitation, providing a nuanced approach to prioritized sampling.

Dueling DQN

A problem with playing Atari2600 games with DQN, which we will discuss more in Chapter 3, is they aim to estimate the Q-value for each action directly. However, in many scenarios, it is not necessary to know the value of each action independently. Instead, what is often more valuable is understanding the value of being in a particular state and the relative advantage of each action in that state.

To achieve this we decouple the Q-function into two components: the value function $V(s)$, and the advantage function $A(s, a)$:

$$Q(s, a) = V(s) + (A(s, a) - \frac{1}{|A|} \sum_{a'} A(s, a')) \tag{2.17}$$

Where $|A|$ is the number of all possible actions. This separation allows the network to learn the value of being in a particular state and the additional value of each action in that state independently.

Multi-Step Learning

In standard Q-learning, the update is based on the immediate reward and the estimate of the Q-value for the next state. However, this approach can be limited in capturing the true impact of an action, especially in environments with delayed rewards. Multi-step learning addresses this limitation by incorporating cumulative rewards over multiple time steps[18], providing a more comprehensive view of the consequences of an action.

The Q-value update in n-step learning is as follows:

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{n-1} R_{t+n} + \gamma^n Q(S_{t+n}, A_{t+n}), \tag{2.18}$$

The Q-values in the network then can be updated using the squared temporal difference error for n-step return

$$L_\theta^{(n)}(t) = \Big(G_t^{(n)} - Q(s_t, a_t; \theta)\Big)^2 \tag{2.19}$$

Choosing the value of n can impact the learning. A balance between capturing sufficient information about the future and avoiding an excessive increase in variance is needed. The introduction of multi-step returns can lead to faster learning[18].

Noisy Nets

Balancing exploration and exploitation has always been a major challenge in machine learning. Deep Q-networks use a ϵ-greedy algorithm, Equation 2.6, to address this problem. However, the ϵ-greedy policy has its limitations. This is shown in environments where the rewards are scarce, or the number of actions to get to the reward is long. An example of these environments is the game Montezuma's Revenge.

Noisy Networks[4] offer a way of injecting randomness directly into the parameters of neural networks. In a Noisy Network, stochasticity is added to the neural network's weights through the use of factorized Gaussian noise. Specifically, instead of having a single weight parameter for each connection, Noisy Networks introduce two additional parameters per weight, representing the means and standard deviations of the noise

The linear layer of a neural network is defined as $y = wx + b$, where x is the layer input, w is the weight matrix, and b is the bias. The corresponding noisy linear layer is defined as:

$$y = (\mu_w + \sigma_w \odot \epsilon_w)x + (\mu_b + \sigma_b \odot \epsilon_b) \tag{2.20}$$

Where $\mu_w + \sigma_w \odot \epsilon_w$ are noisy weights and $\mu_b + \sigma_b \odot \epsilon_b$ is noisy bias. σ_w and σ_b are random variables, and $\odot$ denotes the element-wise product.

Chapter 3

Previous Works

There have been remarkable success stories in the field of reinforcement learning, ranging from agents mastering classic games like Backgammon and Go to the more recent achievements of conquering complex video games like Dota 2 and Starcraft 2. This chapter provides an exploration some of reinforcement learning agents, highlighting their achievements and contributions to the field. The focus then shifts to a more detailed examination of deep Q-network implementations, shedding light on their architecture, training processes. It's important to note that due to time constraints, this report does not delve into the detailed implementations of policy gradient algorithms.

3.1 Reinforcement Learning Agents

3.1.1 TD-Gammon

One of the first success stories in the field of reinforcement learning is TD-gammon[20]. Introduced by Gerald Tesauro, TD-Gammon played a pivotal role in showcasing the potential of reinforcement learning to excel in games with strategic depth. Employing a similar algorithm to Q-learning, TD-Gammon utilized a multi-layer perceptron with one hidden layer to approximate the state value function.

3.1.2 AlphaGo

AlphaGo[16] and its successor, AlphaGo Zero[17], are the two other successes in reinforcement learning agents, learning to play games. Developed by Google DeepMind, they achieved remarkable proficiency in the board game Go. AlphaGo employs the Monte Carlo tree search algorithm (MCTS), incorporating machine learning techniques and a combination of human and computer game training data. In a historical best-of-five series against the top Go player, Lee Sedol, AlphaGo secured victory with the final score of 4-1.

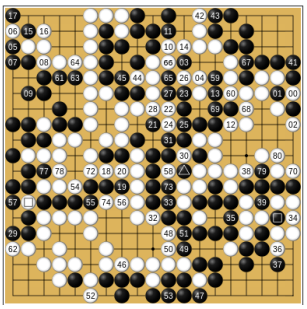


Figure 3.1: The fifth game played between AlphaGo and Lee Sedol. Lee Sedol resigned after the 280 moves.

AlphaGo Zero is the most recent implementation of AlphaGo. Unlike AlphaGo, AlphaGo Zero doesn't need human data to train on and can learn to play games by self-play. AlphaGo Zero has outperformed its previous versions, even learning to play other games like chess and rivaling state-of-the-art chess engines like Stockfish.

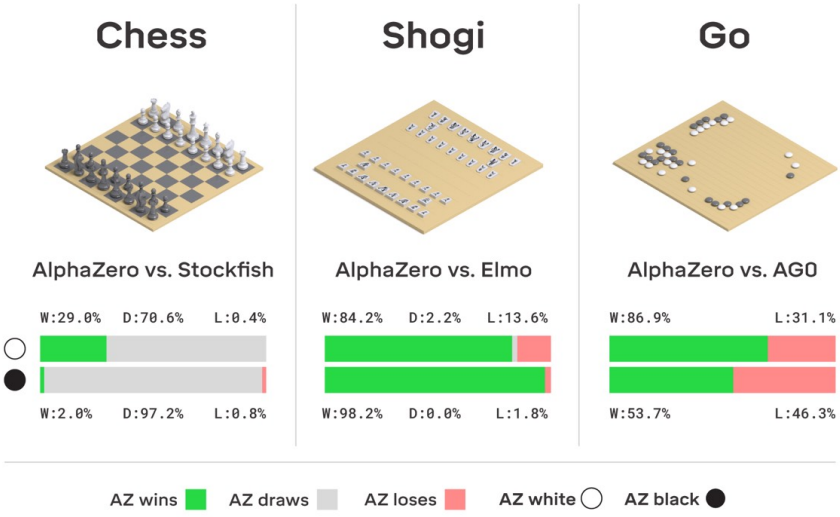


Figure 3.2: Comparison between AlphaGo Zero in other games[9]

3.1.3 OpenAI Five in Dota2

OpenAI Five[10] is a reinforcement learning agent that took on the complex and dynamic world of Dota 2. Dota 2 is a popular multiplayer online battle arena (MOBA) video game. In Dota 2, two teams of five players each, known as "heroes," compete against each other with the objective of destroying the opposing team's Ancient, a large structure located within their base. The game is played on a symmetrical map with three main lanes, each guarded by defensive structures and towers.

Dota 2 presents unique challenges due to its vast strategic depth, real-time decision-making requirements, and the coordination needed among team members. The significance of OpenAI Five's endeavor lies in its ability to master a highly intricate and strategic game that requires not only individual skill but also team coordination. Dota 2 demands a level of teamwork and adaptability that makes it a particularly challenging domain for reinforcement learning.

OpenAI Five made headlines by competing against professional human players in a series of matches and being the first agent that defeated world champions. The outcomes of these matches offered valuable insights into the capabilities of reinforcement learning in handling complex, real-world scenarios.

3.1.4 AlphaStar

AlphaStar[22], created by DeepMind, revolutionized reinforcement learning by conquering the intricate real-time strategy game StarCraft II. Released in 2019, AlphaStar marked a breakthrough in artificial intelligence's ability to navigate the complexities of strategic decision-making in a dynamic environment. StarCraft II is a real-time strategy game. It demands not only rapid decision-making but also a keen understanding of long-term strategy, resource management, and adaptability to unforeseen circumstances. AlphaStar surpassed these challenges by employing deep neural networks that learned optimal strategies through extensive self-play and reinforcement learning. Its triumphs in head-to-head matches against professional human players illustrated the adaptability of machine intelligence in mastering the nuances of a complex and dynamic gaming environment, providing a testament to the capabilities of deep reinforcement learning beyond traditional board and video games.

3.2 Deep Q-Networks in ATARI

The work by Google DeepMind[7] introduced *Deep Q-Networks*, leveraging neural networks to approximate the Q-function. DQN achieved remarkable success in playing a variety of Atari 2600 games, demonstrating the potential of deep learning in reinforcment learning (RL).

Since then, different improvements have been introduced to DQN. Subsequent chapters will delve into various enhancements to DQN discussed in Chapter 2, shedding light on the experimental methodologies and implementations of these methods.

3.2.1 Environment

Before subjecting our DQN to testing, we need an environment for our agents to interact with. In the work by DeepMind [7], The Arcade Learning Environment (ALE) [1] was used. ALE, an Atari 2600 emulator, is a platform for playing various Atari games.

The total actions that an agent can make in ALE is a subset of the entire Atari action space. This action space changes depending on the game. Here is the complete set of Atari actions:

Value	Meaning	Value	Meaning	Value	Meaning
0	NOOP	1	FIRE	2	UP
3	RIGHT	4	LEFT	5	DOWN
6	UPRIGHT	7	UPLEFT	8	DOWNRIGHT
9	DOWNLEFT	10	UPFIRE	11	RIGHTFIRE
12	LEFTFIRE	13	DOWNFIRE	14	UPRIGHTFIRE
15	UPLEFTFIRE	16	DOWNRIGHTFIRE	17	DOWNLEFTFIRE

Table 3.1: Atari2600 action space based on ALE environment

In each game, the agent has a certain number of lives. Every time a life is lost, an episode ends. On each step, the information about the termination of the episode alongside observation and reward is sent to the agent. An environment reset is when the agent has lost all its life, and a new games has to being. The observations are game frames, which are 210 x 160 pixel images with a 128 color palette.

3.2.2 Processing

Using the game frames as a direct input can be computationally demanding. The prepossessing reduces the channels and dimensions of the input frames. The raw farms are first converted to gray-scale and down-sampled to a 110 x 84 image. The final image is cropped to 84 x 84 for the input of a deep neural network[7].

After the conversion of input frames, we implement frame skipping, selecting every k-th frame and ignoring the others. This technique enhances the agent's ability to comprehend motion[19] and reduces duplicate information. Subsequently, a max pool operation is performed for each frame and its predecessor. The final result is illustrated in Figure. 3.3.

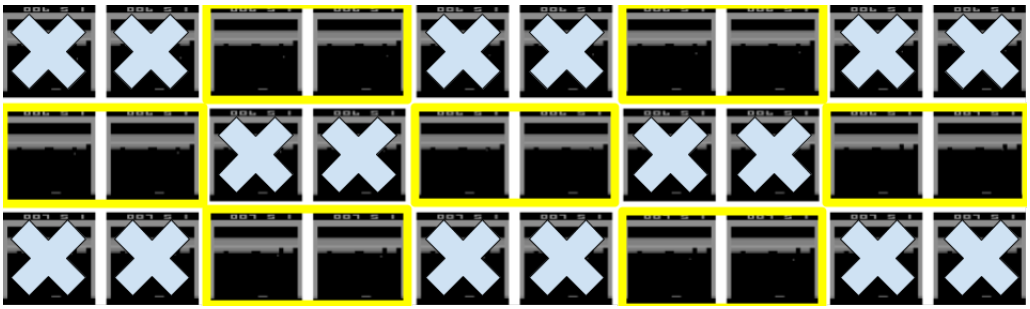


Figure 3.3: Demonstration of reprocessing done by DeepMind on ALE frames in the game of Breakout [19]

By ignoring the X-out frames in Figure 3.3, the final state is the four consecutive frames. If f is the output frame of the ALE, and s is the input state of the algorithm, then $s_i = (f_i, f_{i+1}, f_{i+2}, f_{i+3})$. This means that the input of our network has dimensions of $(4, 84, 84)$.

3.2.3 Network Architecture

The deep network architecture used in the DQN algorithm is mostly the same as DeepMind. The input to the neural network is an 84 x 84 x 4 image. The first layer is a convolutional layer with 16 8 x 8 filters with stride 4 followed by a ReLU[1] activation function. The second layer is also convolutional with 32 4 x 4 filters with stride two, followed by a ReLU activation function. The final hidden layer is fully connected that consists of 256 units with ReLU activation. The output layer is a fully connected linear layer with a single output for each valid action[7].

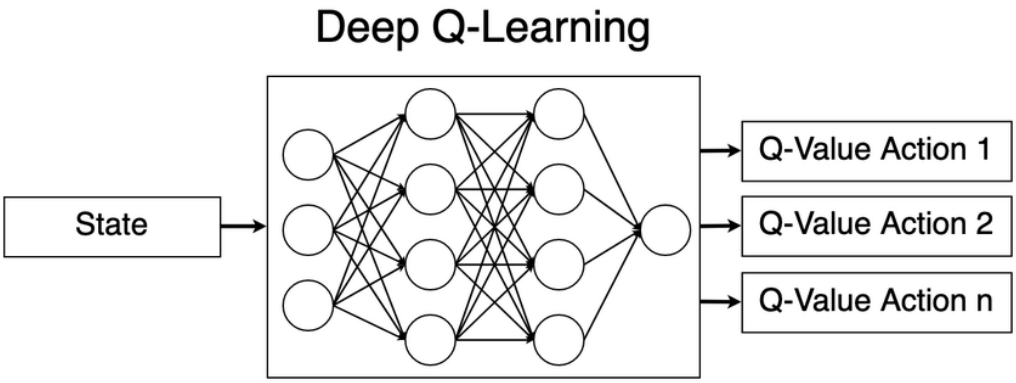


Figure 3.4: General architecture of a deep Q-learning network[15]

The overhaul architecture of the network has not changed but some adjustments and improvements are made.

[1] Rectified Linear Unit

Double DQN

In the double DQN network most of the architecture stays the same, expect the convolutional layers. Instead of two, there are three convolutional layers. The first layer convolves the input with 32 filters of size 8 (stride 4), the second layer has 64 layers of size 4 (stride 2), and the final convolution layer has 64 filters of size 3 (stride 1). The number of units in the final hidden layer is also changed to 512 units[5].

Dueling DQN

The dueling double DQN architecture uses the same one as double DQN. However, the output stream of the final hidden layer is decoupled into the advantage stream and value stream. The final output of the module is calculated based on Equation 2.17.

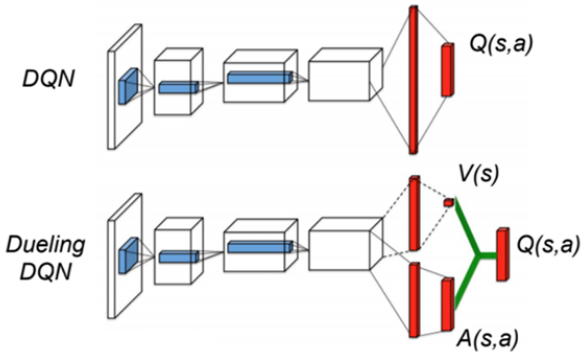


Figure 3.5: Single stream Q-network (top) and the dueling Q-network (bottom)[23]

Rainbow

Rainbow[6] is the latest deep Q-network agent that combines all the improvements in deep reinforcement learning. The main difference in the architecture is the use of Noisy Linear instead of a normal linear function and used distributional RL.

3.2.4 Training and Evaluation

The training method and most hyperparameters used are the same throughout different implementations. The models are trained over 200M frames or 50M steps. The agent is evaluated every 1M steps, and the best policy across the evaluations are kept is as the output of the learning process. The size of the experience replay memory is 1M tuples, and the network is trained every 4 steps by a mini-batch sample of size 32 from the replay memory. The exploration method used is a ϵ-greedy policy with ϵ starting from 1 and decreasing linearly to 0.1 over 1M steps. The discount factor $\gamma = 0.99$ and the learning rate $\alpha = 0.00025$. The number of steps for updating the target network is $\tau = 10,000$. The optimization is done using RMSProp with momentum parameter 0.95.

For stability and because we want to use the same model across all the Atari games, the rewards are clipped between $[-1, 1]$. This makes the learning more stable, and the same learning rate can be used across multiple games[7].

The main evaluation metric for the model is the total reward the agent gets for an episode averaged over several episodes[1]. We test this by agent playing the game using the outputs of its network and employing an ϵ-greedy method with $\epsilon = 0.05$. This metric tends to be noisy. Another metric proposed by DeepMind[7] is using estimated action value functions Q. To test this a random policy is used to collect a set of fixed states and during each evaluation, the average of the maximum predicated Q is calculated.

Prioritized Experience Replay

Because the experience replay memory doesn't change the architecture of the network, Prioritized Experience Replay (PRE) is used alongside the other networks to improve their performance.

The memory is implemented using the sum-tree data structure. We store the priority of each experience next to it. In the proportional variant of PER, which introduces stochasticity when sampling experiences, the experiences are segmented based on their priority and the number of samples we need. Then uniformly an experience is selected from each segment. Based on the implementations of [12], the exponent $\alpha = 0.6$ and the importance weight sampling exponent $\beta = 0.4$. Another hyperparameter adjustment was the learning rate which is a quarter of the baseline learning rate used in DQN $\eta = \eta_{\text{baseline}}/4 = 0.0000625$.

Algorithm 3: Double DQN with Proportional Prioritization

minibatch k, step-size η, replay period K and size N, exponents α and β, budget T
Initialize replay memory $H = \emptyset$, $\Delta = 0$, $p_1 = 1$;
Observe S_0 and choose $A_0 \sim \pi_\theta(S_0)$;
for $t = 1$ *to* T **do**
  Observe S_t, R_t, γ_t;
  Store transition $(S_{t-1}, A_{t-1}, R_t, \gamma_t, S_t)$ in H with maximal priority $p_t = \max_{i<t} p_i$;
  if $t \equiv 0 \mod K$ **then**
    for $j = 1$ *to* k **do**
      Sample transition $j \sim P(j) = \frac{p_j^\alpha}{\sum_i p_i^\alpha}$;
      Compute importance-sampling weight $w_j = (N \cdot P(j))^{-\beta} / \max_i w_i$;
      Compute TD-error $\delta_j = R_j + \gamma_j Q_{\text{target}}(S_j, \arg\max_a Q(S_j, a)) - Q(S_{j-1}, A_{j-1})$;
      Update transition priority $p_j \leftarrow |\delta_j|$;
      Accumulate weight-change $\Delta \leftarrow \Delta + w_j \cdot \delta_j \cdot \nabla_\theta Q(S_{j-1}, A_{j-1})$;
    end
    Update weights $\theta \leftarrow \theta + \eta \cdot \Delta$, reset $\Delta = 0$;
    From time to time, copy weights into target network $\theta_{\text{target}} \leftarrow \theta$;
  end
  Choose action $A_t \sim \pi_\theta(S_t)$;
end

Rainbow

The Rainbow agent uses a different set of parameters than the other deep Q-networks. Because we use prioritized experience replay the learning rate is $\eta = 0.0000625$. Also, the target network is updated every 8000 steps instead of 10,000. Rainbow uses Adam optimizer with Adam ϵ of 1.5 x 10^{-4}. Also because Rainbow uses Noisy Net, we don't need to use ϵ decay during the training of the network. When the network is being tested, we use the ϵ-greedy algorithm with $\epsilon = 0.001$. The Noisy Net σ_0 is set to 0.5, and for multi-step learning we use $n = 3$ for best performance.

3.2.5 Results

The performance of the DQN models, as described briefly in the previous section, is calculated by allowing the agent to play a series of episodes in an Atari game. Then, the average rewards accumulated by the agent are calculated and used as the baseline. In [7, 1], the performance of a random policy and human agent are gathered to compare with the DQN agent.

	B.Rider	Breakout	Enduro	Pong	Q*bert	Seaquest	S.Invaders
Random	354	1.2	0	-20.4	157	110	179
DQN	4092	168	470	20	1952	1705	581
Human	7456	31	368	-3	18900	2810	3690
DQN Best	5184	225	661	21	4500	1740	1075

Table 3.2: The table compares the average total reward for the DQN agent used by[7] using ϵ-greedy policy with $\epsilon = 0.05$ with a human and a random agent.

The performance of the DQN agent, as shown in Table 3.2, seemed promising. To better understand the performance of the agents, we normalize the score for each game:

$$\text{score}_{\text{normalized}} = \frac{\text{score}_{\text{agent}} - \text{score}_{\text{baseline}}}{\max\{\text{score}_{\text{baseline}}, \text{score}_{\text{human}}\} - \text{score}_{\text{random}}} \tag{3.1}$$

This way, we can easily compare the performance of each agent in different games. One thing to note is the use of no-op[2] start. At the beginning of each evaluation, the agent takes several no-op actions to create randomness for the starting point of the agent. One of the shortcomings of the no-op start is that due to the deterministic nature of most Atari games, the agent only needs to remember a sequence of actions to get a good score. To get more robust results, the method proposed by [8] is used. This method is called *human start*. For each game, we use different starting points sampled from a human expert's trajectory. Here is an example from [23] comparing the dueling network with prioritized double DQN:

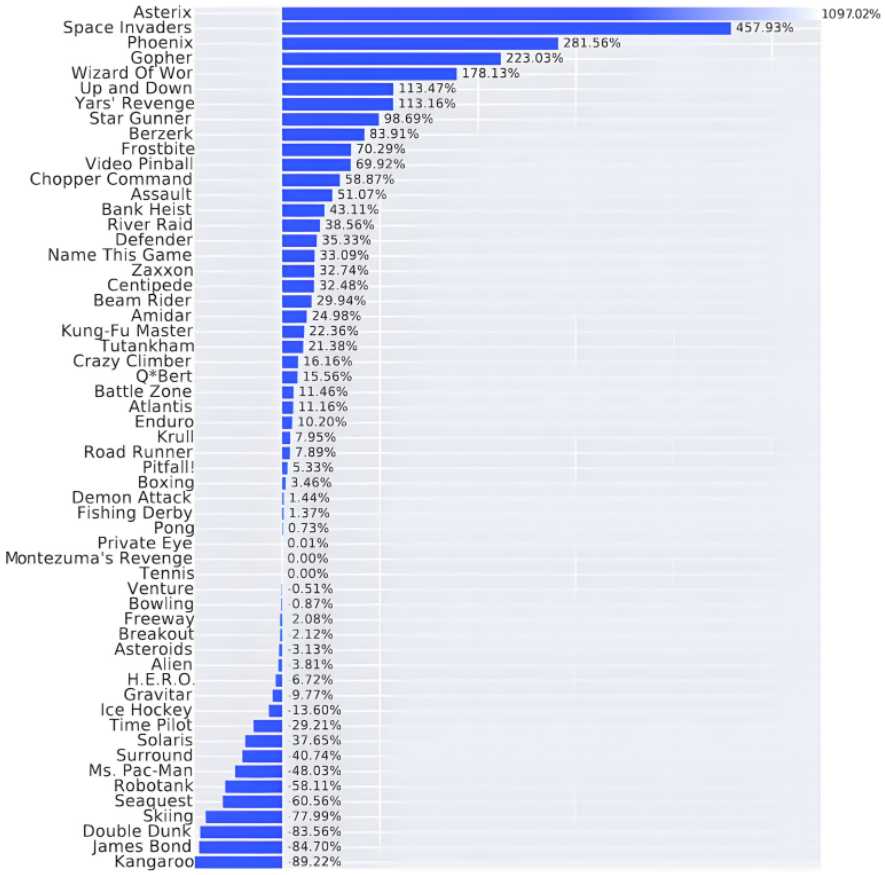


Figure 3.6: Using Equation 3.1 to compare the performance of Dueling DQN and Prioritized Double DQN agents.[23]

[2] no operation

The addition of prioritized experience replay and other improvements, like Noisy Networks and n-step return, have boosted the performance of deep Q-network agents. The Rainbow agent has shown massive improvements compared to other agents.

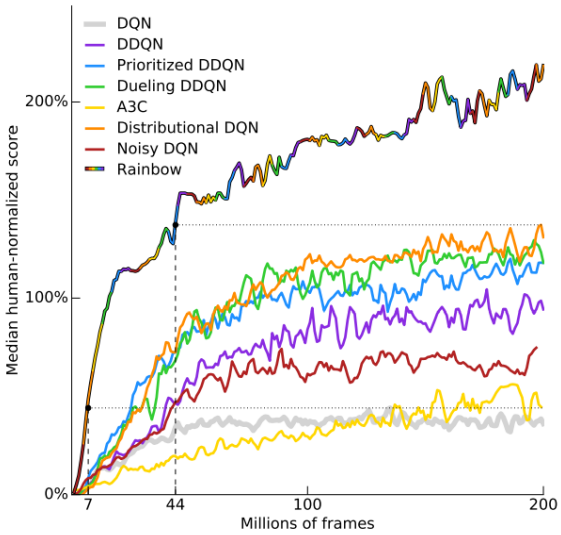


Figure 3.7: Comparison between different deep Q-network algorithms over 57 Atari games[6]

The research done by[6] also provided data to show how much each addition can improve the performance of the agent. Some concepts such as A3C and distributional reinforcement learning were not covered in this paper.

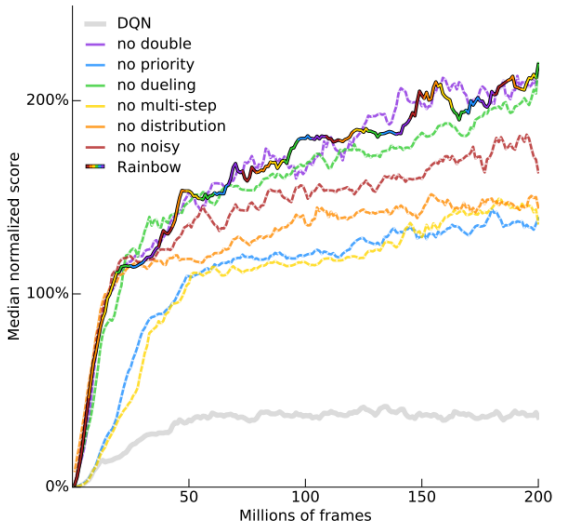


Figure 3.8: Showcasing the different DQN implementations and how much impact they have on the performance of Rainbow agent across 57 Atari games[6]

Chapter 4

Implementation Overview

In this chapter, we take a look at how we implemented the deep Q-network agent and what improvements we made to it. Other than not using distributional reinforcement learning (RL), our implementation closely follows Rainbow[6]. This was done due to a lack of time and knowledge about distributional RL.

4.1 Network Input

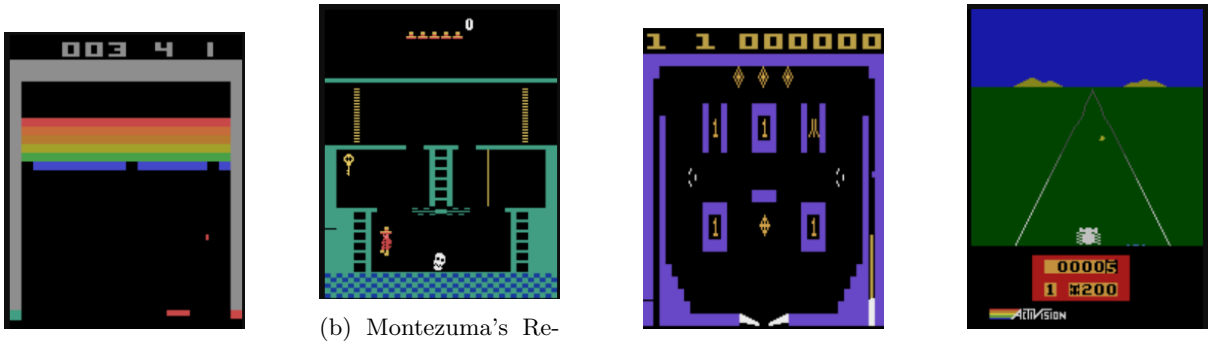
4.1.1 Environment

Instead of using the ALE environment, we used Gymnasium[21]. Gymnasium is an API for single-agent reinforcement learning environments. Gymnasium was created as a fork of OpenAI Gym environment. We used the Atari environment provided by them, which was created using ALE through the Stella emulator.

The observation space of the Atari environment is a 210 x 160 RGB image. After taking a step in the environment 5 elements are returned:

- **Observation**: The next observation that is seen after the agent takes an action.
- **Reward**: The reward as a result of taking the action.
- **Terminated**: Whether the agent reaches a terminal state. After this, the environment must be reset so a new episode can begin.
- **Truncated**: Whether a time limit is reached or the agent has gone out of bounds.
- **Info**: Contains auxiliary diagnostic information (helpful for debugging, learning, and logging). In the case of Atari games, it contains frame number, frame number in the episode and number of lives.

The rewards of the games are all clipped between $[-1, 1]$. This is done so that we can use the same learning rate and network parameters across all different Atari games.



(a) Breakout (b) Montezuma's Revange (c) Video-Pinball (d) Enduro

Figure 4.1: Examples of Atari2600 games. Images where captured from Gymnasium Atari environment

4.1.2 Preprocessing

We used the wrappers provided by gymnasium to preprocess our data. The TimeLimit wrapper was used to make sure that the length of each episode doesn't exceed 108 x 10^3 frames. If the game exceeds more than the number of frames provided, a truncated signal will return.

The other wrapper used is AtariPreprocessin. This wrapper takes care of most of the preprocessing that needs to be done. Firstly it makes the observation an 84 x 84 gray-scaled image. This wrapper also takes no-op actions at the start of each new episode. It also takes care of frame skipping mechanism.

Another useful help that the AtariPreprocessin wrapper provides is the *terminal on the loss of life condition*. In some games, our agent has more than one life, and if one life terminates, the agent continues to play the game. Some of the implementations of DQN used *terminal on the loss of life condition* during the training of the agent but not during the evaluation.

The last wrapper that we use is the FrameStack wrapper to stack the observations to gather. We chose every 4 stacks to be framed.

4.2 Network Architecture

The deep network was created using PyTorch[2] framework. The architecture of the network is a dueling double DQN[23]. We used gradient clipping to improve the stability of the network. Instead of using linear layers, we used the Noisy Linea layers for our fully connected layers.

The input of the network is 4, 84 x 84 images making the shape of the of the input (4, 84, 84). The output is a vector corresponding to Q values for every action that the agent can take based on Table 3.1.

4.3 Training and Evaluation

The hyperparameters used are the same as Rainbow. Learning rate $\alpha = 0.0000625$, discount factor $\gamma = 0.99$, Noisy nets $\sigma_0 = 0.5$, target network update rate $\tau = 8000$ steps, and multi-step return with $n = 3$. For the experience replay, we used the proportional variant of the prioritized experience replay with $\alpha = 0.6$ and $\beta = 0.4 \rightarrow 1$. The network was trained using a batch of 32 experiences from the replay memory every 4 steps. The implementation of the replay memory was done using a sum-tree data structure implemented in[12].

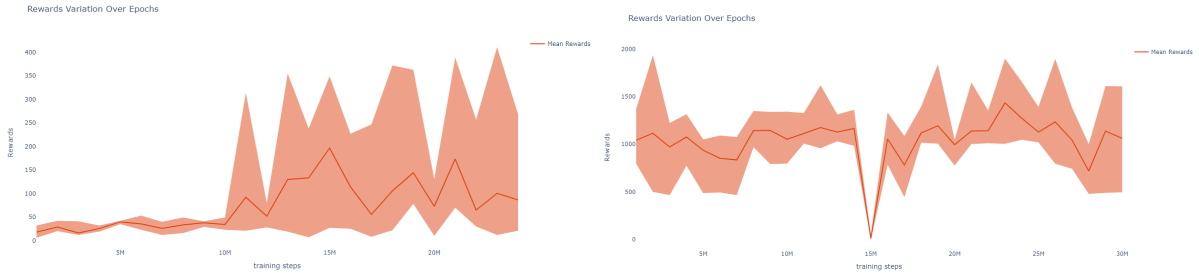
The network was trained for 50M steps using an RTX3060 Laptop GPU and an AMD Ryzen 9 5900HS CPU over the course of four days. The agent was evaluated every 1M step, the average reward for 10 episodes was gathered, and the Q-values were averaged over 500 previously sampled frames.

4.4 Results

Our test show similar results to [6] paper. Although our agent performed better than most DQN agents, it seems that using distributional DQN can significantly improve the performance of the agent. Table

Game	DQN	Distrib.DQN	Our Agent	Rainbow
Breakout	385.5	**612.5**	410.2	417.5
Enduro	729	**2,259.3**	1,940	2,125.9

Table 4.1: Raw score of Breakout and Enduro from agent snapshots that got the highest score during training, average over 200 episodes using a *no-op start*. The information about DQN, Distributional DQN and Rainbow were obtained from [6]



(a) Average rewards for game of Breakout

(b) Average rewards for game of Enduro

Figure 4.2: **Rewards over training steps** Figure 4.2a shows the results of training the agent over 25M steps on the game of Breakout. Figure 4.2b. The bold red line is the mean of the average reward, and the transparent part is the difference between the min and max values

Figure 4.2 shows the learning rates in the implemented agent, with the expected noise evident in the reward plots. Notably, during the testing phase, we observed that certain games, such as Breakout, require a specific action—typically the *fire* command—to initiate gameplay. The noticeable variance in the minimum and maximum reward, despite the agent's overall good performance, can be attributed to the agent's not learning the relation between the *fire* action and game initiation. We theorize that this is caused by the utilization of the *terminate on loss of life* condition during training. By giving the agent the command to start each episode with a *fire* action or adjusting the environment may enhance the agent's mean score. However, such adjustments are unlikely to significantly impact the maximum score achieved by the agent."

Chapter 5

Discussions

In this report, we covered most of the improvements that have been made to deep Q-Network agents, and the results show that these agents can excel in environments like Atari games. The most important thing about these environments is that they have a simple action space and follow the Markov property. As shown by the other papers, these agents don't have a reliable exploration strategy. The only agent that managed to score in Montezuma's Revenge game, which relies heavily on exploration, was the Rainbow agent, which made use of Noisy Networks and distributional RL. This means that the agents might work better in stochastic environments because the environment itself allows for more exploration.

The controlled and straightforward nature of Atari games provides an ideal environment for testing and refining these agents. The positive results obtained pave the way for applications in diverse fields such as robotics, autonomous vehicle control, finance, resource management, network routing, and control systems. The simplicity of the Atari games allows us to fine-tune these agents for deployment in real-world scenarios with higher stakes and limited data availability.

Even though the concept of DQN agents seems promising, they fall short in continuous and more complex environments. To solve more complex environments like Dota2 and Starcraft 2 games, variants of policy gradient methods such as Proximal Policy Optimization (PPO) were used. The complex nature of these gaming environments calls for the adaptation of different strategies, showcasing the need for a nuanced approach to address challenges in more complex scenarios.

During the preparation of this report, time constraints and a limited understanding of the subject prevented us from exploring policy gradient methods. Additionally, distributional methods in reinforcement learning remained unexplored due to similar constraints. Given adequate time and better testing space, we can refining our findings and expand the scope of this research.

An unexplored avenue in our study was the investigation of whether DQN agents can learn from other human players. Using human data to train, or test the agent is done in most reinforcement learning papers. However, teaching an agent to emulate human behavior introduces a distinct set of complexities.

5.1 Future Research

While this report provides valuable insights into the capabilities of DQN agents in playing Atari games, it is crucial to acknowledge the limitations stemming from time constraints and a constrained understanding of the subject matter. Notably, policy gradient methods and distributional approaches in reinforcement learning were unexplored, leaving promising avenues for future research. Despite these limitations, the findings presented here lay the groundwork for potential refinements and extensions with more comprehensive exploration and testing resources. Future research endeavors could include an in-depth examination of policy gradient methods, exploration of distributional reinforcement learning techniques, and expanding the evaluation of DQN agents to diverse and challenging environments.

Chapter 6

Conclusion

In conclusion, this report has explored the improvements and capabilities of deep Q-network (DQN) agents in playing Atari games. We have demonstrated the proficiency of these agents in simple environments with discrete action spaces. While the success of DQN agents in simpler environments is evident, the limitations in complex scenarios necessitate further exploration and the development of tailored strategies.

A fundamental aspect of this study is the acknowledgment of video games as an invaluable testing ground. These games, with their rich and complex environments, serve as a proving ground for developing and refining reinforcement learning agents. The controlled yet intricate nature of video game environments offers a unique opportunity to understand the nuances of agent behavior and learning strategies.

Our study, constrained by time and knowledge considerations, presents a foundation for future research. The unexplored territories of policy gradient methods and distributional reinforcement learning offer promising avenues for enhancing the capabilities of reinforcement learning agents in more intricate environments. Acknowledging the limitations, this work provides valuable insights that pave the way for subsequent investigations.

As we look to the future, the evolving landscape of reinforcement learning holds immense potential. Through continued research and exploration, we can refine existing models, develop new methodologies, and push the boundaries of what is achievable in artificial intelligence.

References

[1] Marc G Bellemare et al. "The Arcade Learning Environment: An Evaluation Platform for General Agents". In: *Journal of Artificial Intelligence Research* 47 (2013), pp. 253–279.

[2] Facebook. *PyTorch: An open source deep learning platform.* Version 1.10.0. https://pytorch.org/. 2022.

[3] John Feil and Marc Scattergood. *Beginning Game Level Design.* Thomson Course Technology, 2005.

[4] Meire Fortunato et al. "Noisy Networks for Exploration". In: *arXiv preprint arXiv:1706.10295* (2019). arXiv: 1706.10295. URL: http://arxiv.org/abs/1706.10295.

[5] Hado van Hasselt, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-learning". In: *arXiv preprint arXiv:1509.06461 [cs.LG]* (2016).

[6] Matteo Hessel and et al. "Rainbow: Combining Improvements in Deep Reinforcement Learning". In: (2017). arXiv: arXiv:1710.02298 [cs.LG].

[7] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *arXiv preprint arXiv:1312.5602* (2013).

[8] Arvind Nair et al. "Massively parallel methods for deep reinforcement learning". In: (2015).

[9] Jennifer Ooellette. *Move over AlphaGo: AlphaZero taught itself to play three different games.* https://arstechnica.com/science/2018/12/move-over-alphago-alphazero-taught-itself-to-play-three-different-games/. Accessed: Nov 30, 2023. 2018.

[10] OpenAI et al. "Dota 2 with Large Scale Deep Reinforcement Learning". In: (2019). arXiv: 1912.06680. URL: https://arxiv.org/abs/1912.06680.

[11] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming.* Wiley, 2014.

[12] Tom Schaul et al. "Prioritized Experience Replay". In: *arXiv preprint arXiv:1511.05952* (2016). arXiv: 1511.05952 [cs.LG].

[13] John Schulman et al. "Proximal policy optimization algorithms". In: *Proceedings of the 34th International Conference on Machine Learning (ICML).* 2017. URL: https://arxiv.org/abs/1707.06347.

[14] John Schulman et al. "Trust region policy optimization". In: *Proceedings of the 32nd International Conference on Machine Learning (ICML).* 2015. URL: https://arxiv.org/abs/1502.05477.

[15] Alessandro Sebastianelli et al. "A Deep Q-Learning based approach applied to the Snake game". In: May 2021. DOI: 10.1109/MED51440.2021.9480232.

[16] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529 (Jan. 2016), pp. 484–489.

[17] David Silver et al. "Mastering the game of Go without human knowledge". In: *Nature* 550.7676 (2017), pp. 354–359.

[18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2018.

[19] Daniel Takeshi. *Frame Skipping and Preprocessing for Deep Q Networks on Atari 2600 Games*. 2016. URL: https://danieltakeshi.github.io/2016/11/25/frame-skipping-and-preprocessing-for-deep-q-networks-on-atari-2600-games/.

[20] Gerald Tesauro. "Temporal Difference Learning and TD-Gammon". In: *Communications of the ACM* 38.3 (1995).

[21] Mark Towers et al. *Gymnasium*. URL: https://github.com/Farama-Foundation/Gymnasium.

[22] Oriol Vinyals et al. "AlphaStar: Mastering the Real-Time Strategy Game StarCraft II". In: *Nature* (2019). URL: https://www.nature.com/articles/s41586-019-1724-z.

[23] Ziyu Wang et al. "Dueling Network Architectures for Deep Reinforcement Learning". In: (2015). arXiv: arXiv:1511.06581 [cs.LG].