Freie Universität Bozen
Libera Università di Bolzano
Università Liedia de Bulsan

Fakultät für Ingenieurwesen
Facoltà di Ingegneria
Faculty of Engineering

Master in Computational Data Science

Master Thesis

Evaluation of Deep Learning techniques for Roll & Write board games

Candidate: Gianluigi Pelle'

Supervisor: Sergio Tessaris

March 2023

Acknowledgements

I would like to express my deepest appreciation to Prof. Sergio Tessaris for his invaluable support and supervision. In addition, I would like to extend my gratitude to my colleague Marco Di Panfilo for his assistance throughout my academic career. Last but not least, I would like to thank Matteo Perin and Federico Zilio for helping me test the game.

Abstract

The majority of game AI research has concentrated on video games and classical board games, ignoring modern board games with innovative mechanics and themes. Additionally, most of the previous studies have been conducted on powerful machines, with training periods of several weeks or months. Consequently, the aim of this study is to evaluate and compare the performance of state-of-the-art reinforcement learning algorithms on modern board games with a training time of less than one week. The game chosen for analysis is the Roll & Write board game Rake&Roll, which was first developed in OpenAI Gym in three variants: the complete version, the deterministic version and the simplified version. After that, DuelingDoubleDeepQNetwork with experience replay (PerD3QN) and Monte Carlo Policy Gradient (REINFORCE) algorithms were implemented in order to train the game AI agents. The REINFORCE agent was unable to learn anything and achieved a similar result as the random agent. Conversely, the PerD3QN agent achieved a significant score, even though it was not optimal. These results were consistent across all game variants introduced. In conclusion, the performance of the PerD3QN agent has demonstrated that, even on commercial machines, it is possible to train AI agents for modern board games within a limited period of time.

Contents

List of Figures

Chapter 1

Introduction

Game AI is a field of artificial intelligence that focuses on developing algorithms and techniques for creating intelligent agents that can play games. This field has many applications in the world of board games, as many popular board games require a high level of strategic thinking and decision-making. For example, in games like Chess or Go, the AI must be able to analyze the board state and make decisions based on the available options in order to make the best move. In addition, game AI can also be used to create computer opponents that can adapt to a player's strategy, making the game more challenging and interesting. By using game AI techniques, board game designers can create games that are more engaging and provide players with a more challenging and rewarding experience. Overall, game AI is an important field for the advancement of board games, and it has the potential to revolutionize the way we play and enjoy these games.

1.1 Research goals

The latest research on game AI has focused on the design and development of intelligent agents for complex video games, e.g. the works presented in (Mnih et al. 2013), (Vinyals et al. 2017), (OpenAI et al. 2019). Aside from the classical board games of Chess and Go, very little research has been conducted on modern board games, which have different game mechanics and offer multiple paths to victory. Furthermore, the majority of previous studies on Game AI have been conducted on powerful machines, with training periods of weeks, or even months. In this regard, we wanted to examine the potential of such methods using commercial hardware and limited training periods.

Therefore, we decided to evaluate and compare the performance of two state-of-the-art reinforcement learning algorithms, namely the DuelingDoubleDeepQNetwork with experience replay (PerD3QN) and Monte Carlo Policy Gradient (REINFORCE). The game on which we chose to analyse these two techniques is the Roll & Write board game Rake&Roll. Due to its reasonable complexity and the fact that it had not been published, it was an ideal match for our needs.

The first step was to develop a digital version of the game in Python, which accurately reproduced all game elements and the rules of the game. The next step was to use OpenAI Gym[1] in order to create an environment in which we could easily test the performance of agents playing the game. With Gym, multiple versions of an environment can be created by

[1] `https://www.gymlibrary.dev`

introducing a few modifications. As a result, we created and tested two variants of the game: a deterministic version and a simplified version. Lastly, we implemented two state-of-the-art reinforcement learning algorithms - PerD3QN and REINFORCE - with the objective of learning how to play the game effectively. In order to compare the two techniques, we will examine the performance of their respective game agents in the environment in comparison to a random agent as well as the maximum score achievable.

There are two hypotheses set in this work.

1. **commercial machines can be used to learn an optimal policy to play board games**
2. **hyperparameter tuning is only necessary when working with very complex environments**

1.2 Results

Although both REINFORCE and PerD3QN agents were unable to learn an optimal strategy to play the game, the PerD3QN agent did achieve a significant score. In contrast, the REINFORCE agent appeared to have learned nothing at all, achieving a similar score to the random agent. A similar set of results was achieved in all game variants introduced.

While the study was intended to evaluate the performance of the agents without tuning the neural network variables, a comparison was made in the training process using two values for the discount factor gamma. Nevertheless, no significant differences were found in learning among the two values.

1.3 Structure of the thesis

The study comes in 7 parts. Firstly, an introduction of the work is given in Chapter 1. Chapter 2 describes the game that was chosen for analysis and the reasons that led to that choice. In Chapter 3, we provide an introduction to reinforcement learning as well as a description of how neural networks work. A detailed explanation of the theory behind REINFORCE and PerD3QN algorithms is provided in Chapter 4. Chapter 5 explains how the original board game and two of its variants were developed in Python and then in OpenAI Gym. The results obtained by both algorithms, as well as the training settings, are presented in Chapter 6. Finally, the conclusions are given in Chapter 7.

Chapter 2

Rake&Roll

In this chapter, we will discuss the reasons for choosing Rake&Roll as our test game. First, we present the category of game we intend to examine. Then, we will describe in detail the distinctive features of Rake&Roll and how it works.

2.1 Roll & Write board games

Most of the related works in the context of game AI focus on video games, such as (Mnih et al. 2013), (Vinyals et al. 2017), (OpenAI et al. 2019). They present a very large observation space, which is generally represented by a collection of game frames represented as RGB images. Data of this type is not structured and is usually unable to provide all the information regarding the game state.

Rather, we chose to work in an environment with a reasonable level of complexity, which includes the following characteristics, as defined in (Russell & Norvig 2016, p. 41):

1. Fully Observable: the complete state of the environment is accessible by the agent at each point in time
2. Stochastic: the agent's actions do not entirely determine the next state of the environment
3. Single-agent: there is only one agent interacting with the environment
4. Static: there is no change in the environment over time
5. Discrete: the observation space and action space of the environment is discrete in nature
6. Episodic: the agent's current action will not affect a future action

As a result, we decided to analyze a board game. Specifically, we chose the category of Roll & Write board games. In these games, players roll dice and mark the results on pieces of paper or eraseable boards. Yahtzee[1] represents the archetype of such games, but many other variations have appeared in the years since.

In order to match the type of environment described above, we focused our search on board games belonging to the multiplayer solitaire genre. This type of board games does

[1] `https://web.archive.org/web/20230121010416/https://en.wikipedia.org/wiki/Yahtzee`

not involve much interaction between the players, making the game very similar whether it is played solo or with others. Moreover, only the solitaire mode was taken into account in our study.

Because all Roll & Write games involve the rolling of dice, there is a significant element of chance involved. Therefore, no game will be the same and the maximum score that can be achieved will depend on the results of the dice. As a result of this, stochastic policies are generally preferred in this kind of environment.

2.2 Rake&Roll

It was not easy to choose which Roll & Write board game to analyze in our study.

Boardgamegeek.com[2] reports that there are 835 published Roll & Write games as of February 2023. While most Roll & Write games meet the characteristics described in section 2.1, not all of them belong to the multiplayer solitaire category. In addition, we decided to analyze an unpublished board game to avoid copyright issues.

Considering these factors, we selected the board game Rake&Roll, designed by David Abelson. It has been awarded second place in the Roll & Write Global Jam 2018[3]. Figure 2.1 illustrates the board of the game.

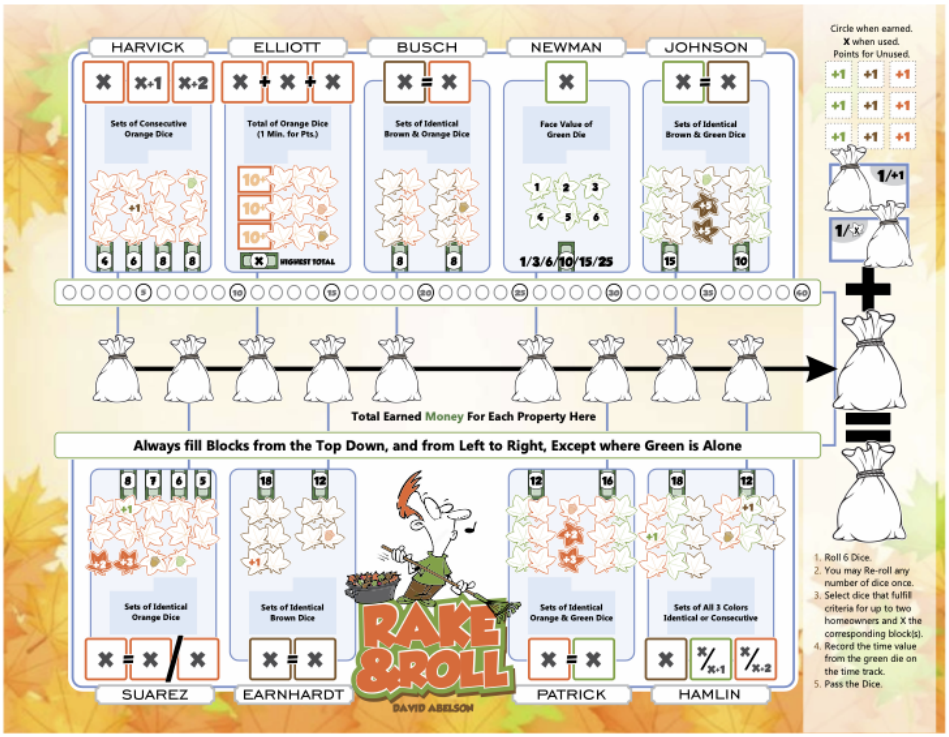


Figure 2.1: Rake&Roll board game

[2] https://web.archive.org/web/20230222214152/https://boardgamegeek.com/boardgamefamily/41222/mechanism-roll-and-write

[3] https://web.archive.org/web/20221123084600/https://sites.google.com/view/rollandwriteglobaljam/home/results-and-downloads

Appendix A provides a comprehensive explanation of the rules of the game. Here is an excerpt from the rulebook that presents an overview of Rake&Roll.

> In this game, you and your opponents are teenagers who have each laid claim to a block in your neighborhood. Each of your neighbors has negotiated unique payment terms with you to rake their leaves. Your goal is to earn the most money before the winter arrives.

Every turn, each player rolls six dice, three oranges, two browns, and one green. After that, any number of dice may be rerolled once. The goal is to rake leaves and satisfy the neighbors on your block by selecting up to two combinations of dice. The neighbors must be satisfied with a specific set of dice of a particular color and value, e.g. Harvick requires three orange dice with consecutive values. As soon as a player chooses a combination of dice from those available, he will select a neighbor's property and cross the leaves corresponding to the dice selected. While neighbor properties usually consist of rows or columns of leaves, which must be crossed from left to right and top to bottom, Newman properties have a specific condition for each leaf and can be crossed in any order. By crossing out certain leaves, a bonus is earned that can be used at any time. Then, before passing the dice to the next player, the time track advances based on the value of the green die.

The game ends when the time track reaches or surpasses 40. Each player will play an equal number of turns. Players total their money from all sections based on their scoring conditions. The player who has earned the most money is the winner!

Figure 2.2 illustrates an example of a complete turn.

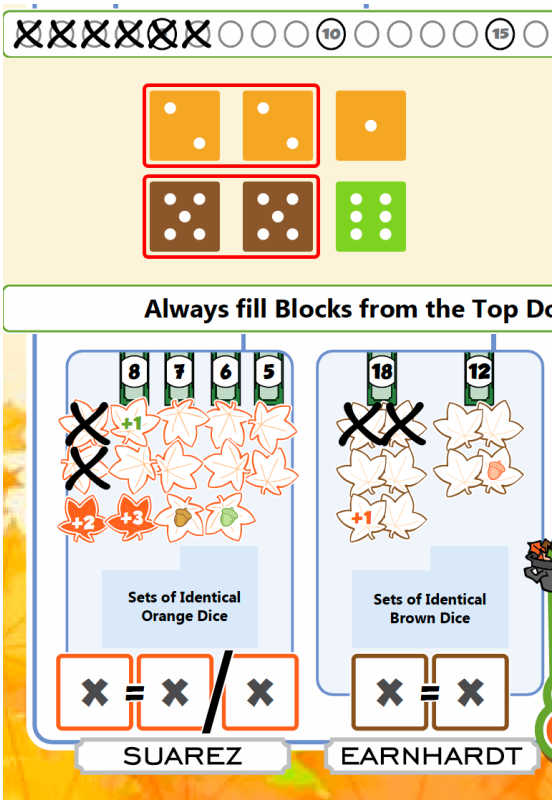


Figure 2.2: Rake&Roll turn example

Chapter 3

Background

The purpose of this chapter is to provide some background information about the algorithms and technologies that we used. Firstly, we will introduce the category of algorithms known as reinforcement learning algorithms. Finally, we will discuss the logic of neural networks, which were used to develop more advanced algorithms.

3.1 Reinforcement Learning

Reinforcement learning (RL) is a subfield of machine learning concerned with developing algorithms and models that enable agents to learn optimal behavior through interaction with their environment. In RL, an agent acts to maximize a reward signal that it receives as feedback for its actions in an environment. As it explores its environment and receives feedback on the outcomes of its actions, the agent learns to make better decisions over time. Robotics, game playing, and autonomous driving are all examples of applications where reinforcement learning has been successfully applied. The paradigm provides an effective means for agents to learn from their experiences and adapt to new and changing environments.

3.1.1 Core elements

In order to understand how Reinforcement Learning works, it is necessary to define its components, which are as follows:

1. agent: the entity that interacts with the environment to learn and make decisions
2. environment: a physical or virtual world in which an agent learns and decides what actions to take
3. action: the actions that an agent can perform
4. state: a representation of the current situation of the agent in the environment
5. reward: every action taken by the agent is rewarded by the environment. This reward is usually a scalar value
6. policy: a function that maps states to actions. It defines what action the agent should take in a given state. It can be either deterministic or stochastic

3.1.2 Agent-environment interaction loop

The agent performs some actions in the environment and observes how the environment's state changes. For each action-observation exchange (a timestep), the agent receives a reward. Upon reaching a goal or making progress towards it, the agent is rewarded positively, otherwise a negative reward is given. As a result, the agent will be trained to maximize its reward over a long period of time. As soon as the agent achieves its goal or fails to do so, the environment enters a terminal state. When this occurs, the environment is reset to a new initial state. A continuous environment, however, does not have an end and the agent is constantly seeking to increase its reward. Figure 3.1 illustrates the interaction between the agent and the environment.

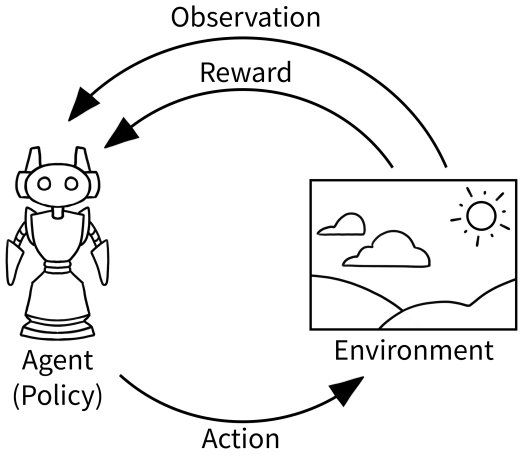


Figure 3.1: Interacting with the Environment, Gym Documentation

3.1.3 Q-Learning

Q-learning is a well-known reinforcement learning algorithm that is used to help an agent make decisions in an environment in order to maximize its rewards. The agent learns which actions to take in each state by trying out different actions and observing the rewards it receives from the environment. The algorithm works by building a table of expected rewards for each state-action pair, which is called the Q-table. The Q-table is updated every time the agent takes an action and receives a reward, with the goal of increasing the expected reward for each state-action pair. This process of updating the Q-table is based on the Bellman equation, which is a mathematical equation used in optimization problems. The result is a Q-table that the agent can use to make the best decision possible in any given state, based on the expected rewards of each action.

3.2 Neural networks

Advanced RL algorithms rely on neural networks to extract relevant features from complex environments. Most of the time, a great deal of data is irrelevant when determining which action to take at a given timestep. By using neural networks, only the relevant information is taken into account. Furthermore, neural networks are used to approximate the expected

return of specific state-action pairs. Value-based methods rely on these values in order to determine the optimal course of action.

In machine learning, neural networks are models that are based on the structure and function of the human brain. Essentially, they consist of a series of interconnected nodes, known as neurons, which process and transform input data into output. Each neuron applies a mathematical function to its input and outputs the result to the next layer of neurons. The connections between neurons, which are represented by a matrix of weights, are learned from data via a process known as training, which minimizes the error between the predicted output and the actual output. There are many applications for neural networks, including image recognition, speech recognition, natural language processing, and prediction.

Chapter 4

Game AI agent development

The purpose of this chapter is to describe the two state-of-the-art algorithms that were chosen to analyse in our study, namely the Monte Carlo Policy Gradient (REINFORCE) and the Double Dueling Deep Q Network with prioritized experience replay (PerD3QN). Firstly, we will discuss some related works already available in the field of game AI for board games. Then, we will examine in detail the structure and logic of the two algorithms selected.

4.1 Evolution of game AI in board games

For many years, research in the field of artificial intelligence has focused primarily on developing AI for board games, particularly chess, backgammon and Go. The complexity and high level of skill required to play board games have made them ideal benchmarks for AI research.

The study of chess was the main focus of early research in game AI. The first computer program to play chess was developed by Claude Shannon[1] in the 1950s. This program, however, was not capable of making strategic decisions and was not able to play at a high level. During the 1960s and 1970s, chess AI research gained momentum with the development of programs such as Mac Hack VI[2] and chess 4.0[3]. Both of these programs used brute force search methods to analyze the game tree and evaluate possible moves. In 1996, IBM's Deep Blue[4] chess program famously defeated the world champion Garry Kasparov in a six-game match. Through a combination of brute force search and selective pruning of the game tree, Deep Blue was able to analyze more than 200 million possible moves per second. Consequently, this victory marked an important milestone in the development of game AI and demonstrated the potential of computers to surpass human abilities in this field.

Following the study of chess, the development of TD-Gammon presented in (Tesauro 1995) demonstrated the potential for reinforcement learning to be applied to board games with elements of chance. TD-Gammon's success showed that machine learning algorithms

[1] https://web.archive.org/web/20221113175722/https://www.chessprogramming.org/Claude_Shannon

[2] https://web.archive.org/web/20230220021624/https://www.chessprogramming.org/Mac_Hack

[3] https://web.archive.org/web/20230216005346/https://www.chessprogramming.org/Chess_(Program)

[4] https://web.archive.org/web/20230217170546/https://en.wikipedia.org/wiki/Deep_Blue_(chess_computer)

could learn to play at a world-class level through self-play and reinforcement learning without the need for brute force search methods.

In the case of Go, the complexity of the game and the wide range of possible board configurations made it a challenging task for AI researchers. However, in 2016, Google's AlphaGo[5] program defeated the world champion Lee Sedol in a five-game match. The AlphaGo algorithm evaluated board positions and made strategic decisions using a combination of Monte Carlo tree search and deep neural networks. As a result of this breakthrough, game AI has made significant advances and machine learning techniques have demonstrated their potential for complex games as well.

4.2 Review of the State of the Art

In order to determine which algorithms to develop for our research, we reviewed the literature searching for other related works in the field of game AI in board games.

Initially, we discovered the reinforcement learning Python package called SIMPLE (Self-play in Multiplayer Environments), presented in (Foster 2021). The package is capable of training AI agents to play any board game. All it requires is a file containing game logic. It learns everything from the ground up through self-play. In addition, it also supports multiplayer games. In order to train the game agents, it uses the Proximal Policy Optimisation (PPO) engine, developed by (Schulman et al. 2017).

Despite the fact that the git repository consists only of examples of multiplayer board games, they provided a useful basis for understanding how to model the observation and action spaces appropriately. As we had no prior experience with this, it was particularly important to see how they modeled the action space, since there were several options available.

Afterwards, we reviewed the literature more specifically for information relevant to the development of an AI agent for a Roll & Write board game. During our search, we came across the article "Learning to play Yahtzee with Advantage Actor-Critic (A2C)", (Häfner 2021). The article focused on the well-known Roll & Write game Yahtzee, which was a perfect match for the characteristics of the chosen environment. The game AI agent was developed using the Actor Critic algorithm (A2C), (Mnih et al. 2016), with the machine learning framework Google JAX[6]. As with SIMPLE, which was developed using the PPO algorithm, A2C belongs to the category of policy gradient methods. The algorithm represents the policy explicitly by its own function approximator and uses a value estimator to update it in accordance with the policy gradient. Using policy gradient methods, either stochastic or deterministic policies may be produced.

The algorithms just described require advanced optimization techniques which are beyond the scope of this thesis. Thus, we chose to develop the REINFORCE algorithm, which belongs to the same family of policy gradient algorithms. Then, in order to compare the results obtained with another kind of algorithm, we chose PerD3QN, which belongs to the class of value-based methods.

[5] https://web.archive.org/web/20230226222009/https://www.deepmind.com/research/highlighted-research/alphago

[6] https://github.com/google/jax

4.3 REINFORCE algorithm

REINFORCE (Monte Carlo policy gradient), (Sutton & Barto 2018, p. 326), belongs to the class of Policy Gradient methods. In this category of algorithms, the agent's policy is modeled and optimized directly. REINFORCE works by creating a policy, which is then constantly updated until the desired performance is achieved. A policy is a model that calculates the probability of taking each of the available actions given the current state. Based on these probabilities, the agent decides which action to perform in the environment. The agent is trained through a series of episodes. A policy gradient update is performed at the end of each episode based on the total reward derived from the current policy. One of the advantages of the REINFORCE algorithm is its ability to generate stochastic policies.

4.3.1 Pseudocode overview

The pseudocode of the REINFORCE algorithm can be seen in Figure 4.1. Following is an overview of the algorithm's logic. First, it creates a random policy, usually represented by a neural network. The policy is a model that takes as input the current state of the environment and returns the probability distribution across all the action space. Second, it uses the policy to play an entire episode of the game. During each timestep of the episode, however, the action-probabilities along with the chosen action and the immediate reward obtained are stored in memory. Based on this information, the discounted reward $G = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ for each timestep is calculated using backpropagation. The parameter γ refers to the discount factor of future rewards, which quantifies their significance. Finally, the expected return generated by following the current policy is calculated and the weights of the associated neural network are updated accordingly in order to increase performance. The entire process is then repeated for a chosen number of training episodes.

As a result of this algorithm, the weights matrix theta is produced, which represents the weights of the neural network that models the policy of the agent.

REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ (G_t)
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t,\boldsymbol{\theta})$

Figure 4.1: REINFORCE: Monte Carlo Policy Gradient, Sutton & Barto summary chap 13

4.4 PerD3QN algorithm

PerD3QN is a state-of-the-art algorithm that combines several techniques into a singular algorithm. It belongs to the class of value-based methods. Algorithms in this category learn

to estimate the value of different states or actions in an environment. The goal of value-based methods is to find a policy, or a mapping from states to actions, that maximizes the expected cumulative reward obtained by the agent over time. Value-based methods typically use a function approximator, such as a neural network, to estimate the value of states or actions. The most popular value-based method is Q-learning, which updates an estimate of the expected reward based on the immediate reward and the estimated value of the next state.

4.4.1 Deep Q-Network

The DQN (Deep Q-Network) algorithm was introduced in (Mnih et al. 2013). The combination of both reinforcement learning and deep neural networks succeeded in solving a wide range of Atari games. Deep Q-Network is a more advanced version of Q-Learning. In Q-Learning, each state-action pair value is stored in a table, the Q-table. When the observation space and action space grow dramatically, it becomes impossible to store all values in a table. DQN proposes a solution that replaces the Q-table with a deep neural network that approximates the Q-values of each state-action pair. Using a neural network also has the advantage of working well in continuous environments, since similar states can be better generalized. Figure 4.2 illustrates the evolution of the Q-table into a neural network.

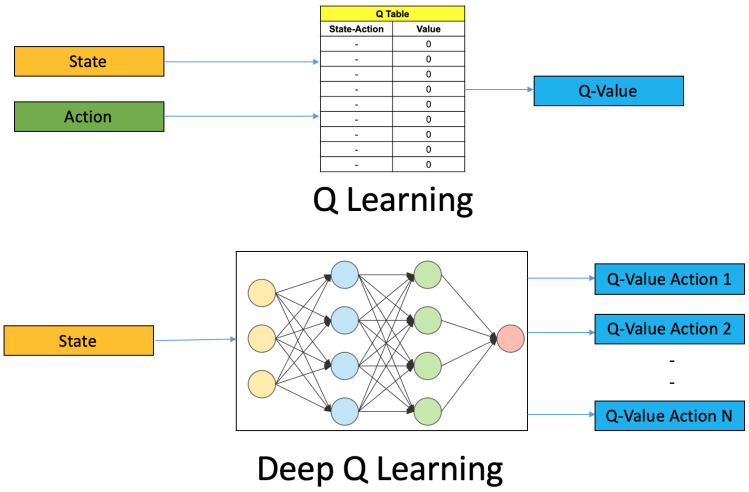


Figure 4.2: Deep Q-Networks, Introduction to Deep Q-Learning

The pseudocode of the DQN algorithm can be seen in Figure 4.3. Following is an overview of the algorithm's logic. As a first step, a replay memory is created to store the interactions between agents and environments, also known as experiences. Next, the action-value function Q is modeled using a neural network.

Then, training begins. In each training episode, the agent selects a random action with probability epsilon, otherwise, it selects the action with the highest value. Upon completing the chosen action, the transition - (S_t, a_t, r_t, S_{t+1}) - is stored in the replay memory. Finally, a minibatch of transitions is sampled for training using gradient descent.

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
  end for
end for

Figure 4.3: Deep Q-learning with experience replay pseudocode, (Jafari et al. 2019)

4.4.2 Double DQN

The Double DQN technique was developed in (van Hasselt et al. 2015). DoubleDQN derives from Double Q-learning, a method developed to reduce overestimations that may occur under certain conditions. This occurs more often in large-scale problems, due to learning's intrinsic estimation errors. DoubleDQN introduces a second Q-value estimator so that the former is used for selecting actions and the latter for evaluating them. The online and target neural networks introduced by (Kavukcuoglu 2015) were the best candidates for implementing this novel technique. Each timestep, the agent selects an action from the online network and calculates its estimated value-based on the target network. The key formula of the algorithm is:

$$Y_t = R_{t+1} + \gamma Q(S_{t+1}, \underset{a}{\operatorname{argmax}}\, Q(S_{t+1}, a; \theta_t); \theta_t^{'}). \tag{4.1}$$

where θ_t refers to the weights of the online network, whereas $\theta_t^{'}$ refers to the weights of the target network. R_{t+1} is the immediate reward, γ is the discount factor and S_{t+1} is the next state.

As opposed to the online network, which is updated continuously when an agent interacts with its environment, the target network is only soft updated after a certain number of interactions. The τ parameter, which ranges from 0 to 1, determines the degree of the update. A value of 1 would totally overwrite the weights between the two networks, whereas a value of 0.5 would move the weights of the target network in the direction of the online network without assuming the exact same values.

4.4.3 Dueling network

Dueling network architectures have proved to be beneficial for deep reinforcement learning in (Wang et al. 2015). In the dueling neural network architecture, state value approximation is explicitly separated from action advantage approximation. After some feature extraction

layers, there are two independent dense layers that share a common input. The first approximates the state value, and the second approximates the action advantage for each action available in that state. At the end of the process, the two estimations are combined so that a single output Q value is produced. Figure 4.4 illustrate the architecture of such dueling networks highlighting in red the key components.

In an intuitive sense, dueling architectures are able to determine which states are valuable (or not) without needing to understand the impact of each action on each state individually. The use of this method is particularly beneficial in states where actions do not have any significant impact on the environment.

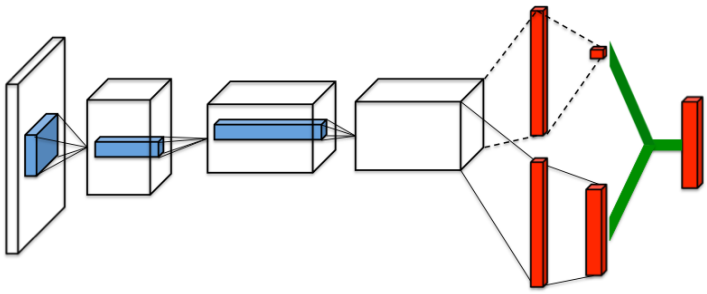


Figure 4.4: Dueling Network Architectures, Image from (Wang et al. 2015)

4.4.4 Prioritized experience replay

As part of the Atari DQN work, a technique known as Experience Replay was introduced to increase the stability of network updates. As data collection progresses, transitions are added to a circular buffer known as the replay buffer. Then, in order to compute the loss and gradient during training, rather than using just the latest transition, a mini-batch of transitions is sampled from the replay buffer. A batch of uncorrelated transitions has the dual advantage of allowing the same data to be reused multiple times while improving learning stability.

Previous works sampled experience transitions uniformly from replay memory. While this approach is effective, transitions are simply replayed at the same frequency as when they were first experienced, regardless of their significance. (Schaul et al. 2015), presented a more sophisticated technique known as prioritized experience replay. This technique is designed to increase the replay probability of experiences that should lead to a higher expected learning progress, which is represented by their absolute TD-error. It is a measure to evaluate the difference between the predicted and actual reward received at a particular time step. The formula for the TD-error is:

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t). \tag{4.2}$$

where R_{t+1} is the immediate reward, γ is the discount factor and $V(S_t)$ and $V(S_{t+1})$ represent the values of the current state and the future state.

As there is no standard formula for $V(S_t)$, which represents the value of the state at timestep t, it is implementation independent. In the case of the PerD3QN algorithm, the value of each state is approximated with a one-unit dense layer right before the output layer of the neural network.

Chapter 5

Experiment setup

In this chapter, we describe the development of the digital edition of Rake&Roll, as well as the implementation of the OpenAI Gym environment. The entire code for this project can be found in Zenodo[1]. Afterwards, some variants of the game are discussed as well as how they were developed.

5.1 Experiments description

As a first step, a version of the game with all the rules was developed, including all of the available bonuses. Henceforth, the neighbors available in the game will be referred to as the sections of the game. In order to limit the number of actions available, the game flow was divided into four phases: Reroll, DiceChoice, SectionChoice, and InnerSectionChoice. The result is that instead of having independent actions that encompass all possible options, such as rerolling, choosing the dice, and selecting the section, there are now only a subset of legal actions available for each of these phases. Consequently, the number of actions available in total is reduced, but not all actions are legal in all phases.

Afterwards, the game was developed in several variations with simplified rules.

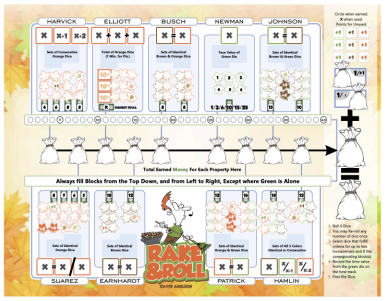
In order to ensure that no errors were present in the logic of the game, multiple tests were developed in order to verify that each version of the game behaved correctly. Due to the amount of time that it takes to run the algorithms, this was a critical aspect to consider. This is because a bug could result in a significant amount of time being wasted.

Rake&Roll-Complete was the first environment created, and it represents the original game faithfully. Figure 5.1 shows the comparison between a picture of the original game and a screenshot of the Gym game environment. There is only one limitation on the Complete environment: bonuses can only be collected, not used. As a result of its complexity, the environment has a large observation area, and due to the presence of dice, it is a stochastic environment.

In Rake&Roll-Deterministic, the game becomes deterministic by introducing a fixed random seed. As a result of this feature, a smaller subset of the observatory space will be explored, therefore reducing the number of states that will be accessible.

Rake&Roll-Simple is a very simplified version of Rake&Roll-Complete. The four phases of the game have been combined into one, and the action space has been reduced. The actions are very straightforward. All you need to do is to select the section you wish to fulfill

[1] `https://zenodo.org/badge/latestdoi/611813688`



(a) Rake&Roll picture

(b) Rake&Roll-Complete screenshot

Figure 5.1: Rake&Roll vs Rake&Roll-Complete

(a) Rake&Roll-Complete screenshot

(b) Rake&Roll-Simple screenshot

Figure 5.2: Rake&Roll-Complete vs Rake&Roll-Simple screenshot

and the dice will be automatically selected and utilized. Therefore, the observation and action spaces are significantly reduced. As with Rake&Roll-Complete, Rake&Roll-Simple is a stochastic environment as it does not make use of a fixed seed for randomization.

A comparison of Rake&Roll-Complete and Rake&Roll-Simple is shown in Figure 5.2.

5.2 Technologies

For this project, we used the Conda package and environment management system. This application is compatible with all operating systems, including Windows, macOS, and Linux, and it manages all dependencies between installed packages. To setup our customized environment, we created a configuration file listing all of the required frameworks and packages.

All code was written in Python 3.10. We used the Numpy[2] library to manipulate multidimensional array objects and perform operations on them. In Python, it represents the fundamental package for scientific computing.

[2] `https://numpy.org`

All machine learning models were developed and trained using TensorFlow[3], an open source library. Additionally, we chose to use tensors in order to execute all operations on the Graphical Process Unit (GPU). Finally, the game was implemented in OpenAI Gym[4], a platform for the development and testing of learning agents. In particular, it was used the version 0.21 of Gym.

5.3 Python implementation

The digital implementation of Rake&Roll includes several Enums and Classes. The most significant are RollAndRakeState, ContinuosSection, and IrregularSection. RollAndRakeState manages all global game information, including the turn counter, the phase flow, and the dice available during the game. IrregularSection and ContinuousSection are subclasses of the same abstract class Section, which defines the logic and behavior of a single game section. ContinuousSection defines a section in which crosses are to be inserted sequentially, from left to right and from top to bottom. In contrast, IrregularSection contains a list of boxes that may be crossed in any order.

5.3.1 Game flow explanation

The logic of the game was implemented as follows:

On turn 0, the game begins. There are 40 time units available and six dice on the display, three oranges, two browns, and one green. The game is in the Reroll phase, which allows for one reroll and a maximum of two dice combinations to be used. The player can now choose between two actions: rerolling a combination of dice, or choosing a combination of dice. In the event the player decides to reroll some dice, the game enters the DiceChoice phase. This is when he must determine a combination of dice to take and proceed to the SectionChoice phase. Alternatively, the player may proceed to the SectionChoice phase immediately if he is satisfied with the dice available during the Reroll phase.

During the SectionChoice phase, the player must choose which section of the game he wishes to spend his dice on. If the player selects an irregular section, the game advances to the InnerSectionChoice phase, where the player must specify which boxes he wishes to cross. In either case, if there is still a dice combination available, the chosen dice are discarded from the display and the game returns to the DiceChoice phase. Otherwise, the available time units of the game will decrease based on the value of the green die, and the game will proceed to the next turn. The dice will be rerolled, and the combinations and rerolls available will be reset to two and one, respectively. As soon as the number of time units available decreases to a value that is lower or equal to zero, the game will end.

A diagram illustrating the flow of the different phases of the turn can be found in figure 5.3.

5.4 OpenAI Gym implementation

The game was implemented in OpenAI Gym in order to develop game AI agents.

OpenAI Gym is a toolkit that provides a collection of environments for developing and testing reinforcement learning algorithms. It is designed to be an open-source platform

[3] `https://www.tensorflow.org`
[4] `https://www.gymlibrary.dev`

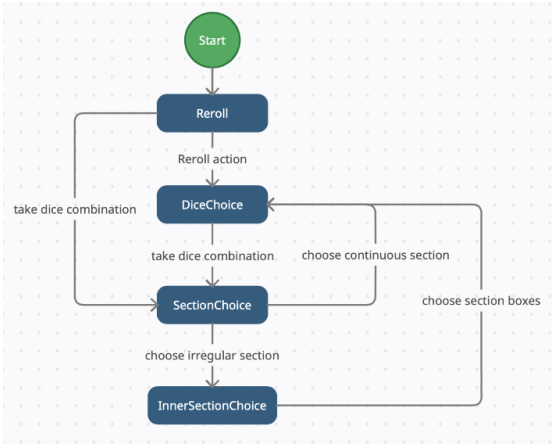


Figure 5.3: Flowchart of turn phases

that allows researchers and developers to experiment with various reinforcement learning techniques and compare the results across different environments. Additionally, it allows the creation of custom designed environments with ease.

5.4.1 Rake&Roll-Complete environment

In the Complete environment, a close representation of the original game is implemented following the descriptions provided in the previous section on the game flow. All game rules are present, except for bonuses, which may not be gained or used. A screenshot of the Complete game environment can be seen in figure 5.4.

The section of the game outlined in box 1 in Figure 5.4 contains all the general game information. This includes the current game phase, the number of rerolls available and the dice combinations remaining. Box 2 shows the dice that are currently available, while box 3 illustrates all game sections and their associated bonuses. Lastly, box 4 outlines all the legal actions that are available during the current game phase.

observation space

In Gym, each state of the game is represented by an observation. Two states with the same observation representation are treated as if they were the same. Therefore, it is essential to select the appropriate values to include in the observation of an environment. Observations should be sufficiently detailed to enable the learning algorithm to distinguish between states, but not too detailed so that it can generalize between similar states.

In order to make the observation, each item of information about the game is coded into an array of values between 0 and 1. These arrays are then concatenated together as a whole.

The observation scheme for the environment Rake&Roll-Complete is the following:

1. available dice [42 values]: available dice values are converted to a length 7 one-hot encoding (since dice that have already been used are assigned a value of 0). As the dice are distinguishable based on their position in the list, which is always fixed, the color of the dice is not encoded into the observation space.

2. game sections [108 values]: each section is represented by an array with a length equal

Figure 5.4: Rake&Roll-Complete game screenshot

to the number of boxes it contains. Empty boxes are marked with a value of 0, while crossed-out boxes are marked with a value of 1

3. game phase [4 values]: with four possible phases for the game, the current phase value is one-hot encoded in a manner similar to the dice available to the player

4. rerolls available [1 value]: the rerolls available value is normalized with a value between 0 and 1

5. dice combinations available [1 value]: the dice combination available value is normalized with a value between 0 and 1

6. remaining time [1 value]: the remaining time value is normalized with a value between 0 and 1

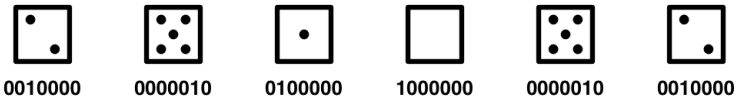


Figure 5.5: Available dice encoding scheme

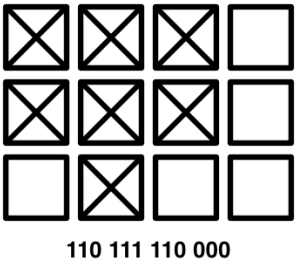


Figure 5.6: Game sections encoding scheme

7. Elliott scoring [1 value]: As opposed to other sections, Elliott section behaves in a completely different manner. In all sections of the game, a cross is placed on top of each leaf, however in the Elliott section, the dice values are written on top of each leaf. Further, the Elliott section score refers to the highest sum of values among the rows of leaves, whereas all other sections score points for each row. Hence, we have introduced a variable in the observation space to indicate the current score of the Elliot section to facilitate the learning process. This variable is normalized with a value between 0 and 1

8. legal actions [142 values]: the observation space ends with an array containing a number of elements equal to the number of possible actions in the environment. A value of 1 indicates that the corresponding action is legal, and a value of 0 indicates that the action is illegal

An example of a complete observation is illustrated in figure 5.7.

action space

The number of actions available is 142, even though not all of them are legal for all possible environmental conditions. Depending on the game phase, only a specific subset of actions are possible. The action scheme for the environment Rake&Roll-Complete is the following:

1. reroll: there is a total of 63 different reroll actions, one for each possible combination of dice to reroll. For instance, the action R245 refers to rerolling the second, fourth, and fifth die

2. take dice combination: there is a total of 63 different take dice combination actions, one for each possible combination of dice to take. For instance, the action T13 refers to taking the first and third die

3. choose category: there is a total of 9 different choose category actions, one for each available section in the game

(a) Rake&Roll-Complete screenshot

Observation space scheme

available dice
[0001000 0000100 0000100 0010000 0000010 0100000]

game sections
[000000000000 000000000 111110000000 011000 110000000000000
000000000000 0000000000 000000000000000 111000000000000]

game phase
[0100]

rerolls available
[0]

dice combinations available
[0]

remaining time
[0.6]

elliot scoring
[0.6]

legal actions
reroll actions
[0000000000000000000000000000000000000000000000000000000000000000]
take dice combination actions
[0000000000000000000000000000000000000000000000000000000000000000]
choose category actions
[000000001]
choose inner category actions
[000000]
pass action
[1]

(b) Rake&Roll-Complete observation scheme

Figure 5.7: Example of a Rake&Roll-Complete observation encoding

4. choose inner category: there is a total of 6 different choose inner category actions, one for each individual box available in the Newman section (these actions become legal only after the choice of the Newman section)

5. pass: taking this action ends the current turn and advances the game to the next. This is the only action that is always legal

The scheme of the action space is illustrated in figure 5.8.

Action space structure

[0 - 62] reroll actions
[63 - 125] take dice combination actions
[126 - 134] choose category actions
[135 - 140] choose inner category actions
[141] pass action

Figure 5.8: Rake&Roll-Complete action space scheme

5.4.2 Rake&Roll-Deterministic environment

Rake&Roll-Deterministic is a minor modification of Rake&Roll-Complete. In both environments, the code is identical with the exception that the latter contains a specific random seed. The purpose of this feature was to simplify the environment and make the game deterministic.

Due to the stochastic nature of dice rolls, there were many different states visited just occasionally in the original game. Since the different states of the environment must be visited multiple times in order to be approximated and learned, this presented a problem for training our game agent. A fixed random seed ensures that the dice will always obtain the same value every time they are rolled within a specific time step. The exact same game can therefore be played if the same game action sequence is selected.

observation space

There is no difference between the action space in Rake&Roll-Deterministic and Rake&Roll-Complete. An example of a complete observation is illustrated in figure 5.7b.

action space

There is no difference between the action space in Rake&Roll-Deterministic and Rake&Roll-Complete. An example of the action space scheme is illustrated in figure 5.8.

5.4.3 Rake&Roll-Simple environment

Rake&Roll-Simple is a simplified version of the original game. To reduce the complexity of the game, phases were eliminated. Consequently, both the observation space and the action space were reduced in dimension.

As opposed to the original flow of the game, there is now only one phase during which you can choose directly what section you wish to spend your dice on instead of choosing a

combination of dice. Each turn, you can either choose a legal section or pass, and you play until the time limit expires. In order to accomplish this, the selection of the section should determine the appropriate combination of dice to be taken without any ambiguity. A review of all sections revealed only one combination of dice that might lead to ambiguity, the Elliott section, which was then removed. As a result of this new game phase type, the maximum number of dice combinations you may take on a single turn has been reduced from two to one. Consequently, fewer boxes will be crossed in each section of the game, decreasing the overall score. To compensate for this problem, the time limit was increased from 40 to 60 units.

In order to infer the appropriate combination of dice to take depending on the selected section, some code was added to the logic of the game.

observation space

The observation space of Rake&Roll-Simple reflects the changes introduced in this version of the game. Starting with Rake&Roll-Complete, all variables related to phases, rerolls, and dice combinations are removed, resulting in a smaller observation space. An example of a complete observation is illustrated in figure 5.9.

Observation space structure

available dice
[0001000 0000100 0000100 0010000 0000010 0100000]

game sections
[110111110000 111111110000 101011 110001110011000 11111011100000
1111000000 111001100011100 111110111000000]

remaining time
[0.9]

legal actions
choose category actions
[000000000]
pass action
[1]

Figure 5.9: Rake&Roll-Simple observation example

action space

The action space for Rake&Roll-Simple is very compact. A total of 9 actions are available, one for each section of the game and one for passing. An example of the action space scheme is illustrated in figure 5.10.

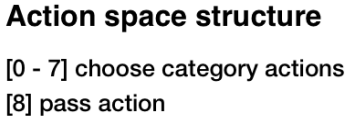


Figure 5.10: Rake&Roll-Simple action space scheme

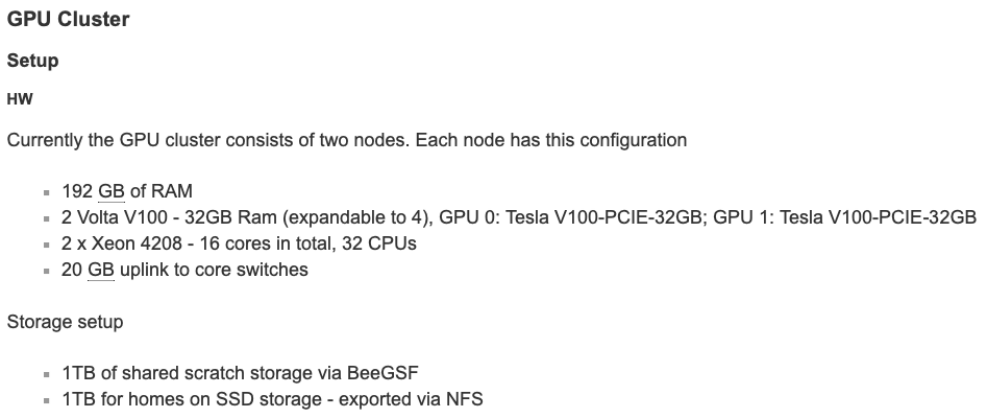
Chapter 6

Evaluation

Throughout this chapter, we will describe how REINFORCE and the PerD3QN algorithms were developed and explain the reasons behind some of the choices that were made in the development of these algorithms. A comparison of all results obtained by the two algorithms is made at the end of the chapter.

6.1 Computation setting

In this study, the objective is to test the performance of state-of-the-art reinforcement learning algorithms with limited resources. REINFORCE and D3QN with experience replay algorithms were selected for testing. The computation was performed by the unibz GPU cluster. All specifications of the cluster can be found in figure 6.1.



GPU Cluster

Setup

HW

Currently the GPU cluster consists of two nodes. Each node has this configuration

- 192 GB of RAM
- 2 Volta V100 - 32GB Ram (expandable to 4), GPU 0: Tesla V100-PCIE-32GB; GPU 1: Tesla V100-PCIE-32GB
- 2 x Xeon 4208 - 16 cores in total, 32 CPUs
- 20 GB uplink to core switches

Storage setup

- 1TB of shared scratch storage via BeeGSF
- 1TB for homes on SSD storage - exported via NFS

Figure 6.1: Cluster configuration

Another significant limitation was the time constraint, which was set at one week maximum for the training of each agent. Throughout this chapter, the implementation of the two algorithms previously discussed will be described in detail. Afterward, the performance of the two agents will be compared with that of a random agent. At the end, the results obtained will be discussed.

6.2 REINFORCE algorithm

REINFORCE was the first algorithm to be developed. The code was written following the pseudocode presented in (Sutton and Barton, 2020, p. 326) as well as the Python implementation of Abhishek Suran available on github[1].

As opposed to the standard implementation, a few modifications were made in order to improve the handling of illegal actions and speed up the learning process. This idea was taken from the reinforcement learning Python package SIMPLE (Self-play In MultiPlayer Environments), developed by David Foster.

Due to the inability of neural networks to fix particular output values for specific inputs, illegal actions would require a lot of training experiences to approximate a negative value. Two modifications were made by SIMPLE in order to address this issue. As a first step, the observation space is expanded so that a value is assigned to each action indicating whether or not it is legal. As a second step, a new custom layer is added just prior to the output layer of the neural network. This layer acts like a mask: it extrapolates the legal action indicators from the input layer, leaving legal actions unchanged, while penalizing illegal actions severely. Figure 6.2 shows the logic of this layer.

```
61
62  def policy_head(y, legal_actions):
63
64      for _ in range(POLICY_DEPTH):
65          y = dense(y, FEATURE_SIZE)
66      policy = dense(y, ACTIONS, batch_norm = False, activation = None, name='pi')
67
68      mask = Lambda(lambda x: (1 - x) * -1e8)(legal_actions)
69
70      policy = Add()([policy, mask])
71      return policy
72
```

Figure 6.2: Mask layer

6.2.1 Actor model architecture

The actor model was implemented with a feedforward neural network. The model consists of two subnetworks, which are then combined to produce the final output. Therefore, the input of the network, which is an observation of the current state of the environment, is divided into two tensors: the observation variables and the legal actions indicators.

The observation tensor is elaborated through a sequence of two dense layers, with a size of 128 units each. The goal of these layers is to extract the most significant data and derive a set of features to describe the same information in a more compact manner. This is also known as the feature extraction process. The new set of data is then passed through a dense layer whose size is equal to the number of actions available in the environment. The goal of this layer is to assign an expected reward for each action.

The legal actions tensor, on the other hand, is elaborated with a custom layer that converts legal actions to 0 and illegal actions to -1e8. Then, the outputs of these two subnet-

[1] https://github.com/abhisheksuran/Reinforcement_Learning/blob/8947ec88af/Reinforce_(PG)_ReUploaded.ipynb

works are added together so that the expected value of illegal actions is nullified. Finally, there is a softmax output layer that is used to normalize the expected values of each action into a probability distribution. An overview of the neural network architecture is presented in Figure 6.3, while the Python implementation is depicted in Figure 6.4.

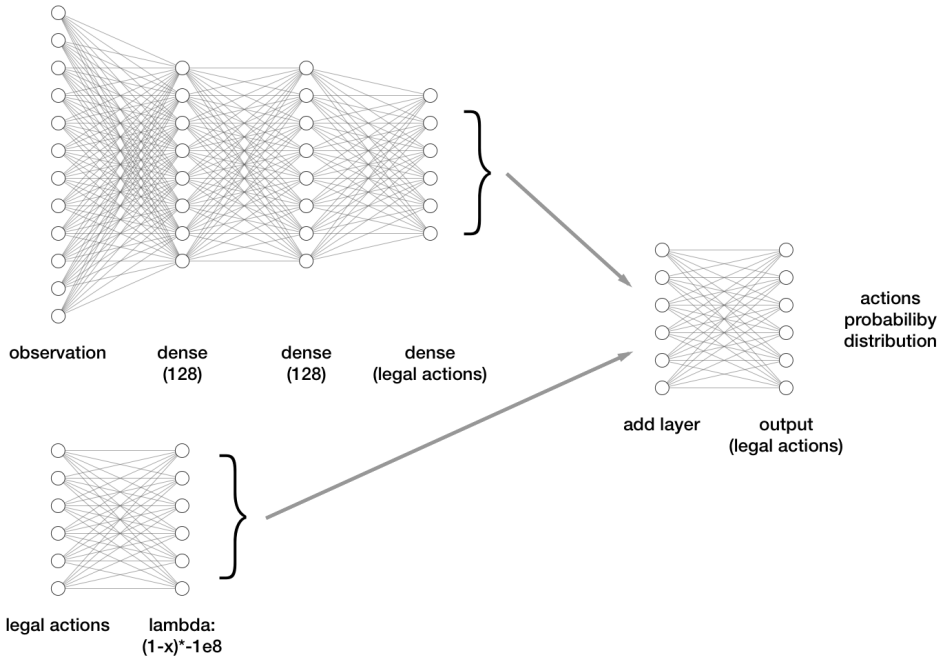


Figure 6.3: Actor model architecture

In neural networks, topology is a very critical factor to consider. In deep learning, there are a lot of hyperparameters to tune in order to achieve the desired outcome. These include the number of units in each layer and the number of hidden layers. In order to tune hyperparameters, several configurations must be compared, which takes a lot of time.

Due to this, we analyzed the neural network configurations implemented in SIMPLE for the games Connect4 and Butterfly, and chose one that could be scaled appropriately. Connect4 has an observation space of 126 variables and a model of 1 hidden layer with 128 units. Butterfly, on the other hand, has a total of 4655 variables in its observation space, and its model is comprised of five hidden layers with 128 units each.

As our game environment contains 150 variables, which is a bit larger than the observation space of Connect4, we decided to adopt a model with two hidden layers containing 128 units each. In this way, the neural network can also learn complex nonlinear functions.

6.2.2 Training setting

The REINFORCE algorithm was used to train all the game environments, namely Rake&Roll-Complete, Rake&Roll-Deterministic and Rake&Roll-Simple. As previously discussed, the code was developed using Tensorflow and is run directly on the GPU. Apart from the input and output layers, which were adjusted according to the observation and action space, the neural network structure remained the same for all game environments. As a policy gradient method, the REINFORCE algorithm does not require any explicit exploration strategy.

```
12
13 class ActorModel(tf.keras.Model):
14     def __init__(self, action_space):
15         super().__init__()
16         self.action_space = action_space
17
18         self.d1 = tf.keras.layers.Dense(128, activation="relu")
19         self.d2 = tf.keras.layers.Dense(128, activation="relu")
20         self.d3 = tf.keras.layers.Dense(action_space, activation=None)
21         self.add = tf.keras.layers.Add()
22         self.out = tf.keras.layers.Dense(action_space, activation="softmax")
23
24     def call(self, obs, legal_actions):
25         x = self.d1(obs)
26         x = self.d2(x)
27         x = self.d3(x)
28
29         # mapping illegal actions to negative values
30         y = 1 - legal_actions
31         y = y * -1e8
32
33         x = self.add([x, y])
34         x = self.out(x)
35
36         return x
37
```

Figure 6.4: Actor model implementation

Exploration occurs in an intrinsic way, since the generated policy is a stochastic policy, and each possible action has a non-zero probability of being selected.

Gamma was the only parameter examined with different values. This variable is of crucial importance, since it quantifies the importance we give to future rewards. In general, low gamma values lead to the algorithm preferring immediate rewards, whereas high gamma values lead to the algorithm preferring long-term rewards. Initially, the gamma value was set to 0.9, but even after a prolonged training period, the agent kept receiving a final score of 0. After reducing the gamma value, the game agent began to learn how to score some points. Consequently, 0.5 and 0.2 were selected for parameter tuning in all of the experiments.

A fixed number of 200'000 episodes was chosen for training the game agent, so that the algorithm could complete its execution in less than a week.

6.3 PerD3QN algorithm

Several different techniques are involved in the PerD3QN algorithm, therefore a standard implementation is not available. Thus, this algorithm was developed in multiple steps, beginning with the implementation of the DoubleDQN algorithm based on Methods and Apparatus for Reinforcement Learning (Mnih et al., 2017). As a second step, the Python classes Memory and SumTree developed by Jaromír Janisch[2] were used in order to implement the Prioritized Experience Replay as described by Schaul et al. in 2016. Finally, the architecture of the neural network representing the value estimator was adapted to take advantage of

[2] https://github.com/jaromiru/AI-blog/blob/361e8c79dc/Seaquest-DDQN-PER.py

the concept proposed in Dueling Network Architectures for Deep Reinforcement Learning (Wang et al., 2016).

To deal with illegal actions and speed up the training process, the same technique presented in section 6.2 and shown in figure 6.2 was employed.

6.3.1 Value model architecture

The value model was implemented with a feedforward neural network. The model consists of two subnetworks, which are then combined to produce the final output. Therefore, the input of the network, which is an observation of the current state of the environment, is divided into two tensors: the observation variables and the legal actions indicators.

The observation tensor is elaborated through a sequence of two dense layers, with a size of 128 units each. The goal of these layers is to extract the most significant data and derive a set of features to describe the same information in a more compact manner. This is also known as the feature extraction process. The new set of data is then passed through two different dense layers. The former is a one-unit dense layer whose objective is to estimate the value of the state represented by the input observation. The latter is a dense layer that calculates the relative advantages of each possible action. This is accomplished by first estimating the overall value for each action, and then subtracting the average of all values to determine the advantages of each action. The state value and the action advantages layers are then added together in order to calculate the estimated value for each action.

The legal actions tensor, on the other hand, is elaborated with a custom layer that converts legal actions to 0 and illegal actions to -1e8. Then, the outputs of these two subnetworks are added together so that the expected value of illegal actions is nullified. Finally, there is a softmax output layer that is used to normalize the expected values of each action into a probability distribution. An overview of the neural network architecture is presented in Figure 6.5, while the Python implementation is depicted in Figure 6.6.

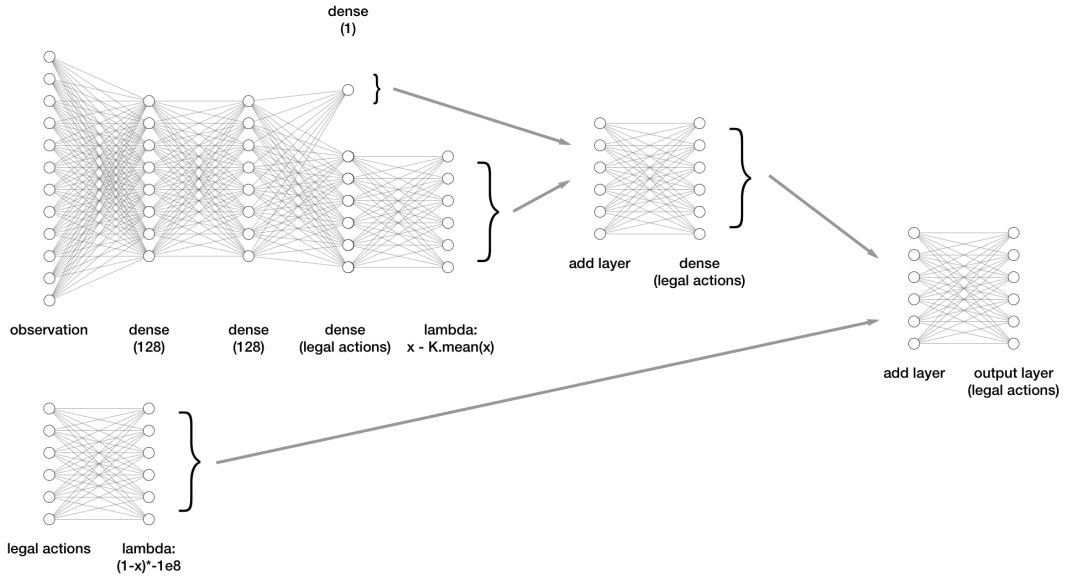


Figure 6.5: Value model architecture

```
class ValueModel(tf.keras.Model):
    def __init__(self, action_space):
        super().__init__()
        self.action_space = action_space

        self.d1 = tf.keras.layers.Dense(128, activation="relu", kernel_initializer='he_uniform')
        self.d2 = tf.keras.layers.Dense(128, activation="relu", kernel_initializer='he_uniform')
        self.d3 = tf.keras.layers.Dense(1, activation="relu")
        self.d4 = tf.keras.layers.Dense(action_space, activation="relu")
        self.add1 = tf.keras.layers.Add()
        self.add2 = tf.keras.layers.Add()

    def call(self, obs, legal_actions):
        x = self.d1(obs)
        x = self.d2(x)

        state_value = self.d3(x)

        action_advantage = self.d4(x)
        action_advantage = Lambda(lambda a: a - K.mean(a, keepdims=True), output_shape=(len(legal_actions),))(action_advantage)

        sum_value = self.add1([state_value, action_advantage])

        # mapping illegal actions to negative values
        y = 1 - legal_actions
        y = y * -1e8

        x = self.add2([sum_value, y])

        return x
```

Figure 6.6: Value model implementation

6.3.2 Training setting

The PerD3QN algorithm was used to train all the game environments, namely Rake&Roll-Complete, Rake&Roll-Deterministic and Rake&Roll-Simple. As with the previous algorithm, the code was developed using Tensorflow and is run directly on the GPU. Apart from the input and output layers, which were adjusted according to the observation and action space, the neural network structure remained the same for all game environments.

In the PerD3QN algorithm, the epsilon-greedy exploration strategy was chosen to balance exploration and exploitation. Training begins with an epsilon value of 0.5, which indicates that the agent takes a random action half of the time, and the best known action the other half. After every training step, the epsilon value decreases with a decay rate of 0.9998, but it never falls below 0.1. As epsilon decreases, the agent becomes more likely to take the most effective action that he knows rather than attempt to discover a new strategy by making random decisions.

Configuring the experience replay mechanism was another critical aspect of the training setup process. Each time a new experience is observed, the agent samples a batch of 32 experiences from the memory buffer to enhance its estimation accuracy. The agent begins doing so only after at least 500 experiences have been stored in the memory buffer, which has a maximum capacity of 5000 experiences.

Gamma was the only parameter examined with different values. This variable is of crucial importance, since it quantifies the importance we give to future rewards. In general, low gamma values lead to the algorithm preferring immediate rewards, whereas high gamma values lead to the algorithm preferring long-term rewards. Initially, the gamma value was set to 0.9, but even after a prolonged training period, the agent kept receiving a final score of 0. After reducing the gamma value, the game agent began to learn how to score some points. Consequently, 0.5 and 0.2 were selected for parameter tuning in all of the experiments.

The final parameter to be chosen was τ, which controls the soft network update. This approach prevents the weights of the target network from being simply overwritten with those of the online network. In place of this, they are slightly modified so as to move in the

direction of the online network weights, rather than taking their value as it is. Thus, the τ value was set at 0.1 in order to increase learning stability.

A fixed number of 10 thousand episodes was chosen for training the game agent, so that the algorithm could complete its execution in less than a week. Due to the experience replay phase, fewer training episodes are required compared to REINFORCE. The agent is trained after each step using a batch of 32 previous experiences. REINFORCE makes a number of updates equivalent to the number of steps that occurred in the episode, while PerD3QN makes a number of updates equal to 32 times the number of steps that occurred in the episode.

6.4 Results

This section presents the results obtained from the REINFORCE and PerD3QN game agents in all game environments, namely Complete, Deterministic and Simple. A time limit of one week was imposed on the execution of both algorithms in order to make their results comparable. In order to illustrate the performance of the two game agents, some episode outputs were recorded during training. Specifically, a batch of five score samples was saved every 25000 episodes for REINFORCE, and a batch of ten score samples every 100 episodes for PerD3QN. As a result, the data was presented using a boxplot with a whisker parameter of value 1.5, which is most appropriate for this type of information.

6.4.1 Random agents

A testbed of two newly developed agents was introduced as a means of evaluating the performance of the two algorithms. The first is a pure random agent. The second is a no-pass random agent. It performs random actions, avoiding the pass action whenever possible. There is no benefit to passing. It does nothing but continue the game as it is. The pass action is usually unnecessary, but sometimes you cannot make any moves and must use it to continue.

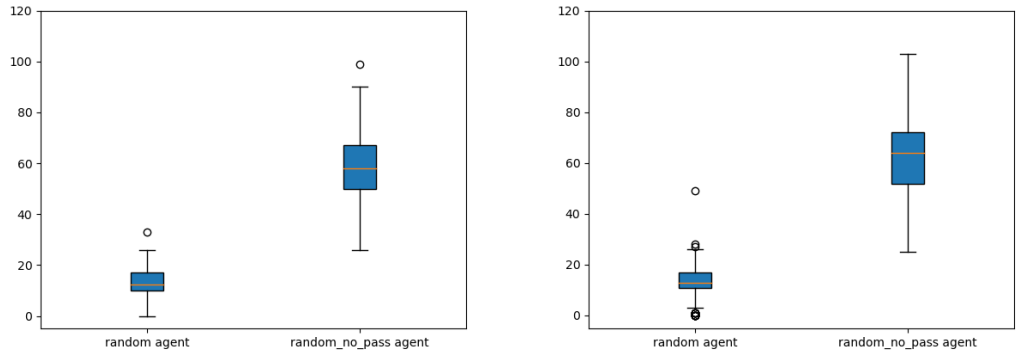
Figure 6.7 shows the scores obtained by these two agents on Rake&Roll-Complete and Rake&Roll-Deterministic.

The random-no-pass agent achieves a significantly higher score than the pure random agent. The reason for this disparity lies in the use of the pass action, which does nothing more than continue the game. Random-no-pass agents avoid passing whenever possible, thereby fulfilling more sections and scoring more points.

Both agents behave similarly in the two environments Rake&Roll-Deterministic and Rake&Roll-Complete. In contrast, the scores obtained by the two random agents are similar on Rake&Roll-Simple, as shown in Figure 6.8.

6.4.2 Parameter tuning of gamma parameter

This section examines the influence of the gamma parameter during the training phase. Gamma is a critical parameter since it serves to quantify the importance we place on future rewards. Thus, we decided to compare the results obtained with gamma values of 0.2 and 0.5. The former causes the learning agent to seek immediate rewards, whereas the latter gives equal weight to both short-term and long-term rewards.



(a) Rake&Roll-Complete random agents comparison

(b) Rake&Roll-Deterministic random agents comparison

Figure 6.7: Random agents comparison on Complete and Deterministic environments

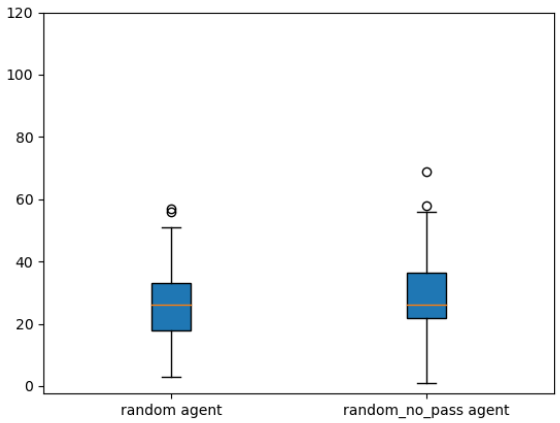
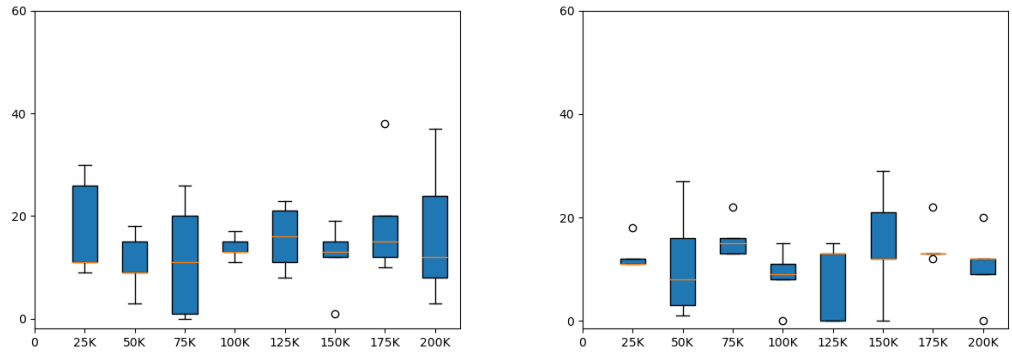


Figure 6.8: Rake&Roll-Simple random agents

Regardless of the environment, the training results obtained with the REINFORCE algorithm do not differ significantly. In both Complete, Deterministic and Simple environments, the training agent exhibits unstable learning. In Figure 6.9, you can see the scores obtained in Rake&Roll-Complete using the REINFORCE algorithm with different values of gamma. Appendix A illustrates the comparison of gamma values for the REINFORCE algorithm across all game environments, including Deterministic and Simple environments.

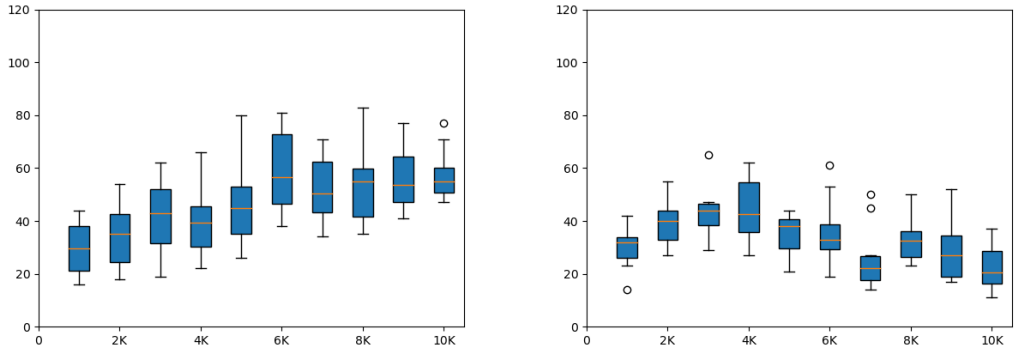
In contrast, the PerD3QN algorithm exhibits some significant differences in the training phase of the Complete environment. When using a gamma value of 0.2, training is very stable and the agent learns to achieve a score of approximately 60 points. Instead, with a gamma value of 0.5, learning becomes very unstable, and after four thousand episodes the agent's score begins to decrease. Figure 6.10 illustrates the differences between the training phase of the PerD3QN algorithm with both values of gamma.

Other than this, there is no significant difference between the training phases in the other game environments. Appendix A illustrates the comparison of gamma values for the



(a) REINFORCE gamma 0.2 (b) REINFORCE gamma 0.5

Figure 6.9: Rake&Roll-Complete REINFORCE training



(a) PerD3QN gamma 0.2 (b) PerD3QN gamma 0.5

Figure 6.10: Rake&Roll-Complete PerD3QN training

PerD3QN algorithm across all game environments, including Deterministic and Simple environments.

6.4.3 Results comparison

For the purpose of comparing the REINFORCE and PerD3QN agents, a thousand episodes of each game environment were played using the models that resulted from the training process. To visualize the distributions of the scores obtained by each agent, we chose to use boxplots. Figure 6.11 illustrates the performance of the agents in the Complete environment.

As shown in Figure 6.11, REINFORCE performs very similarly to the random agent, offering no significant advantages. Meanwhile, the PerD3Qn agent achieved a much higher score, despite being far from reaching the maximum score.

Results obtained in both of the remaining game environments - Deterministic and Simple - are very consistent with those just stated. Determinism introduced in the Deterministic environment does not appear to have any effect on the learned agents, such as the simplifi-

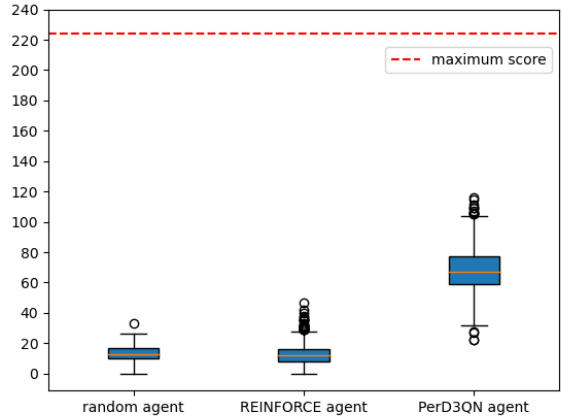


Figure 6.11: Rake&Roll-Complete agents comparison

cation of the game logic in the Simple environment.

The results obtained by the agents in the Deterministic environment are shown in Figure 6.12, while those obtained in the Simple environment are shown in Figure 6.13.

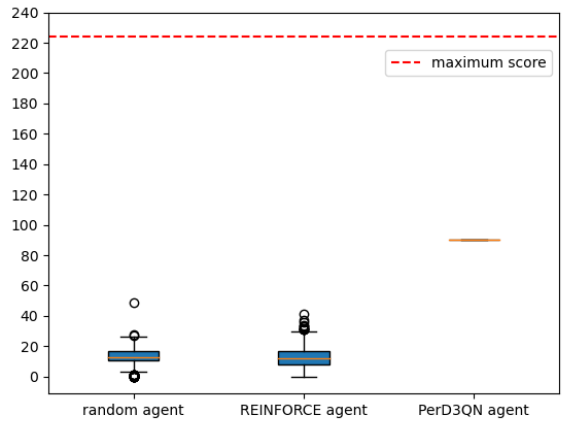


Figure 6.12: Rake&Roll-Deterministic agents comparison

6.5 Discussion

The results presented in the previous section have several limitations. Firstly, there are a number of possible ways in which Rake&Roll can be implemented digitally, depending on how actions are formulated. The same applies to the creation of the OpenAI Gym environment, as the structure of the observation space can be designed in a variety of ways.

It was also crucial to address the issue of illegal actions. As explained in Chapter 4, we adopted the idea introduced in the framework SIMPLE, but several alternatives are also available.

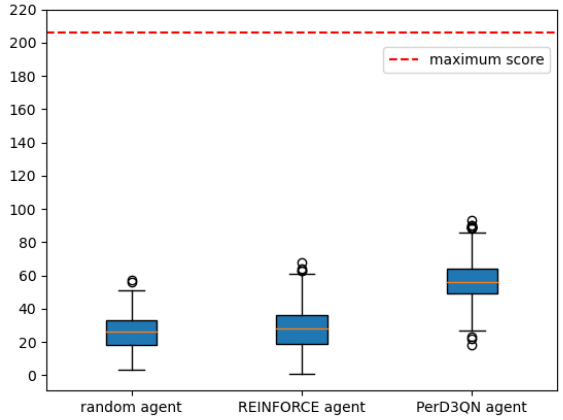


Figure 6.13: Rake&Roll-Simple agents comparison

In conclusion, the biggest limitation in our study was the absence of any hyperparameter tuning for all variables related to the neural network models. Due to the lack of resources and the length of time it takes to train each game agent, we decided to skip this step which would have doubled the study's duration.

Chapter 7

Conclusions

The purpose of our study was to evaluate the performance of state-of-the-art techniques in board games. In particular, we sought to demonstrate the performance of Game AI agents playing board games trained on commercial hardware and with a limited execution time. We selected a game with an appropriate level of complexity for analysis, and developed a Python version of its digital version. Afterwards, the game was implemented in OpenAI gym along with two variants. In order to train the Game AI agents, REINFORCE and PerD3QN learning algorithms were selected.

In conclusion, we find that the REINFORCE agent is unable to learn how to play the game successfully, whereas the PerD3QN agent manages to learn how to play the game in a successful manner, even if it is not optimal. Despite the limitations of the results presented in Chapter 6, the performance of the PerD3QN agent demonstrates that even on commercial machines such computations can be performed and with a significant degree of accuracy.

In future studies, a different board game may be evaluated using the same techniques to verify that the difference between the policy gradient method and the value-based method remains consistent. Another approach to continue our study would be to develop more advanced algorithms and compare their performance with that of the REINFORCE and PerD3QN agents.

Appendix A

Appendix

RAKE &ROLL

Components
6 6-Sided Dice
- 3 Orange
- 2 Brown
- 1 Green

1 Board/Scoresheet per Player

A Dice Rolling, Leaf Raking Game for 1-3 Players

In this game, you and your opponents are teenagers who have each laid claim to a block in your neighborhood. Each of your neighbors has negotiated unique payment terms with you to rake their leaves. Your goal is to earn the most money before the winter arrives.

Game Play

The player who has most recently raked leaves is the first player.

- Beginning with the first player, on your turn you will roll all 6 dice.
- You may re-roll any number of dice one time.
- You will select up to 2 combinations of dice to be used to rake leaves and satisfy the various neighbors on your block.
- You will choose a neighbor's property and draw an X on the leaves that correspond to the chosen dice on the property you have selected. (see below)
- At the end of your turn, and before passing the dice to your opponent, record the value of the green die on the time track by drawing an X in the corresponding amount of spaces.

Bump Bonuses

A +1 can be used during your turn, before assigning a die, to increase the die's value by 1. Dice cannot increase higher than 6. When you cover up a +1 leaf you earn a +1. Draw a square around the +1 of the corresponding color on the upper right corner of the board. When you use a +1, draw an X in the box to indicate that it is no longer available.

Acorn Bonuses

When you cover up an Acorn you must immediately place an X on the next available leaf of the same color, at whatever property you choose. The next available leaf is the next empty leaf in the current row or column at that property that matches the Acorn's color. Empty leaves in previously raked columns or rows are not available. Effectively, Acorns can help you achieve a requirement more easily by reducing the number of leaves required.

Game End

When one player's time track reaches or surpasses 40, the end game is triggered. Each player will play an equal number of turns. After all turns have been played, tally your score. The player who has earned the most Money is the winner! Write the score for each property in the corresponding bag in the center of the board. Add all these scores together to form a base score.

You also earn 1 for each unused +1 in your +1 Bank, and You earn 1 for each un-scored X that you have drawn at a property during the game. Add these 3 numbers together to get your final score. This is how much money you earned. The player with the highest score is the Winner! In case of a tie, the player with the least un-scored X's is the winner.

Figure A.1: Rake&Roll rules page 1.

PROPERTY REQUIREMENTS AND SCORING:

HARVICK HOME

3 Consecutive Orange Dice. Place an X in all 3 Leaves in the next available column. In the scoring phase you will earn the amount of money shown for each column you have completed.

ELLIOTT HOME

2 or 3 Orange Dice with a total value of 10 or greater. Write the total in the BOX and place an X in each leaf that corresponds to a die (ie 2 leaves for 2 dice). You only need to assign dice to this property a minimum of 1 time to be able to score it. In the scoring phase you will earn an amount of money equal to the highest written value.

BUSCH HOME

1 Orange Die and 1 Brown Die with identical values. Place an X in each leaf in the next available row. When you have completed the third and sixth rows you will earn Money. In the scoring phase you will earn an amount of money corresponding to the number of columns you completed.

NEWMAN HOME

1 Green Die that matches the value of an available leaf. Place an X on the matching value leaf. In the scoring phase you will earn an amount of money corresponding to the number of covered leaves (ranging from 1-25 for 1-6 Leaves).

JOHNSON HOME

1 Green Die and 1 Brown die with identical values. Place an X in each leaf in the next available row. If you assign a second brown die with a matching value, you may place an X in the center leaf of the row to earn the bonus. If you miss a bonus in any of the first 3 attempts, you may attempt to earn the bonuses again in the last 3 attempts. When you have completed the third and sixth rows you will earn Money. In the scoring phase you will earn Money for each column you complete (up to 2) plus any bonus earned (indicated by the colored leaf with the white number in it).

SUAREZ HOME

2 or 3 Orange Dice with identical values. If a third die is not assigned to the current column, the corresponding bonus is not earned. Place an X in the corresponding 2 or 3 Leaves in the next available column. In the scoring phase you will earn the amount of money shown for each column you have completed. **NOTE:** No money is earned for completing the first column.

EARNHARDT HOME

2 Brown dice with identical values. Place an X in each leaf in the next available row. When you have completed the third and fifth rows you will earn money. In the scoring phase you will earn an amount of money corresponding to the number of columns you complete (up to 2).

PATRICK HOME

1 Green Die and 1 Orange die with identical values. Place an X in each leaf in the next available row. If you assign a second orange die with a matching value, you may place an X in the center leaf of the row to earn the bonus. If you miss a bonus in any of the first 3 attempts, you may attempt to earn the bonuses again in the last 3 attempts. When you have completed the third and sixth rows you will earn Money. In the scoring phase you will earn an amount of money corresponding to the number of columns you complete (up to 2) plus any bonus earned (indicated by the colored leaf with the white number in it).

HAMLIN HOME

1 die of each color. The assigned dice may be EITHER identical or consecutive in value. Place and X in all 3 leaves in the next available row. When you have completed the third and fifth rows you will earn money. In the scoring phase you will earn an amount of money corresponding to the number of columns you complete (up to 2).

Figure A.2: Rake&Roll rules page 2.

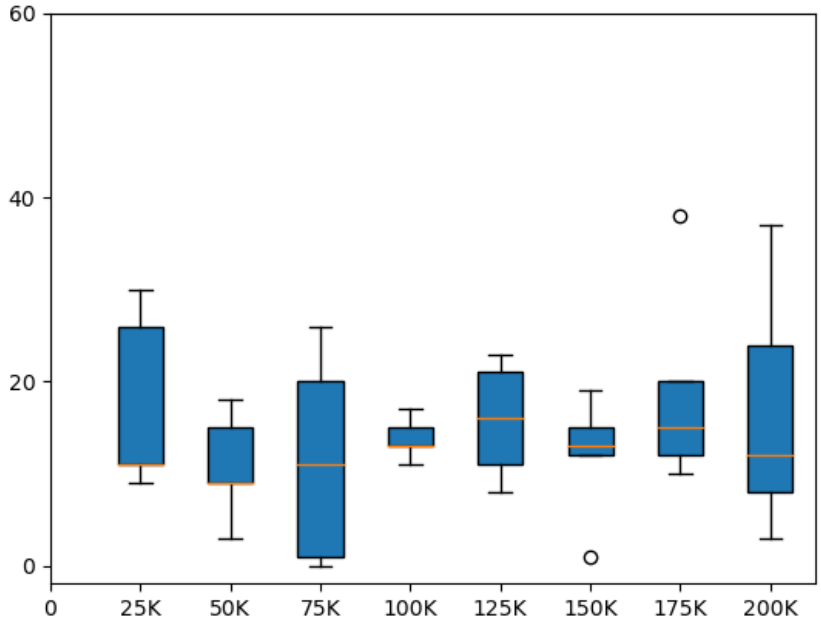


Figure A.3: REINFORCE gamma 0.2 on Rake&Roll-Complete.

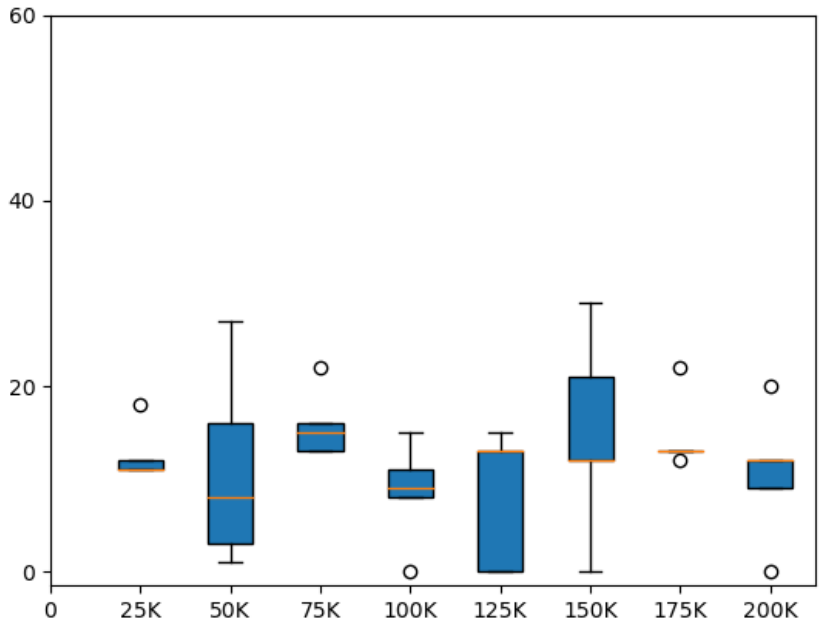


Figure A.4: REINFORCE gamma 0.5 on Rake&Roll-Complete.

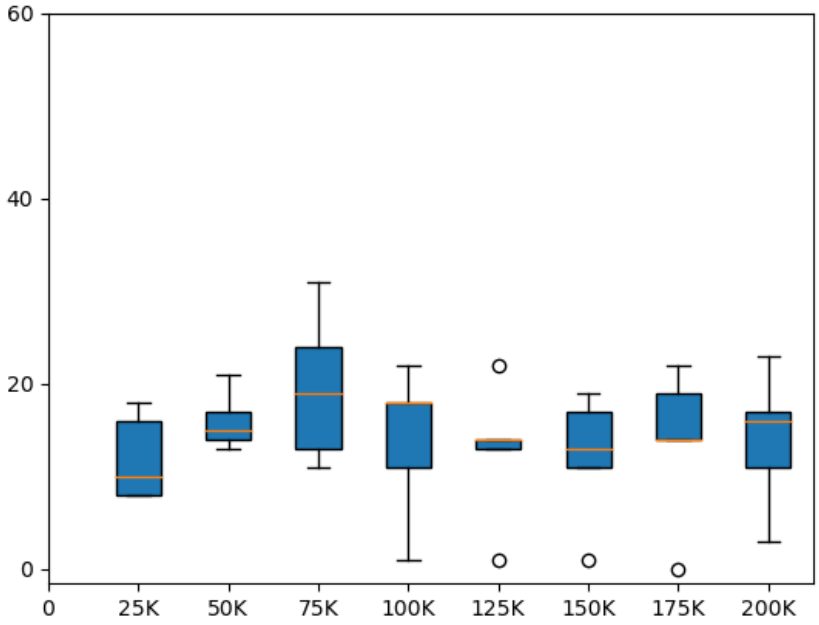


Figure A.5: REINFORCE gamma 0.2 on Rake&Roll-Deterministic.

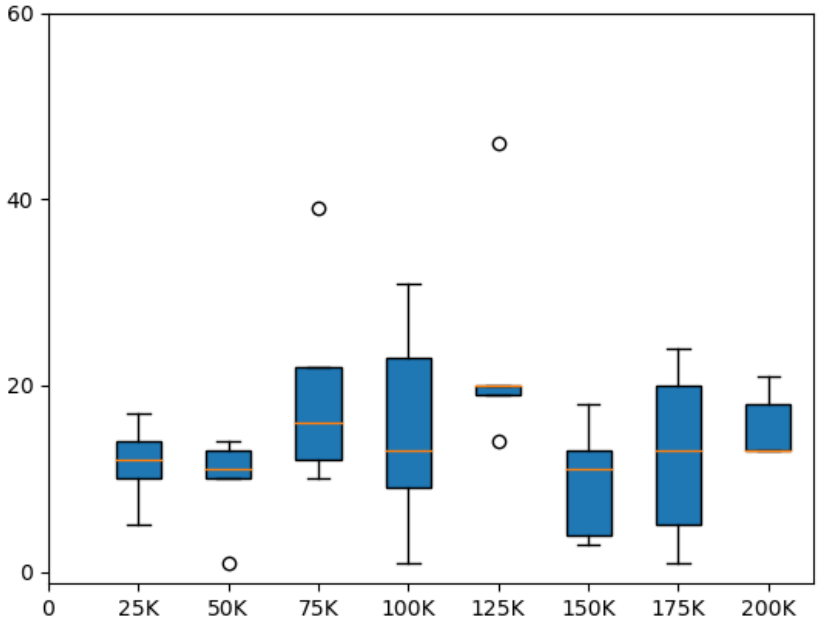


Figure A.6: REINFORCE gamma 0.5 on Rake&Roll-Deterministic.

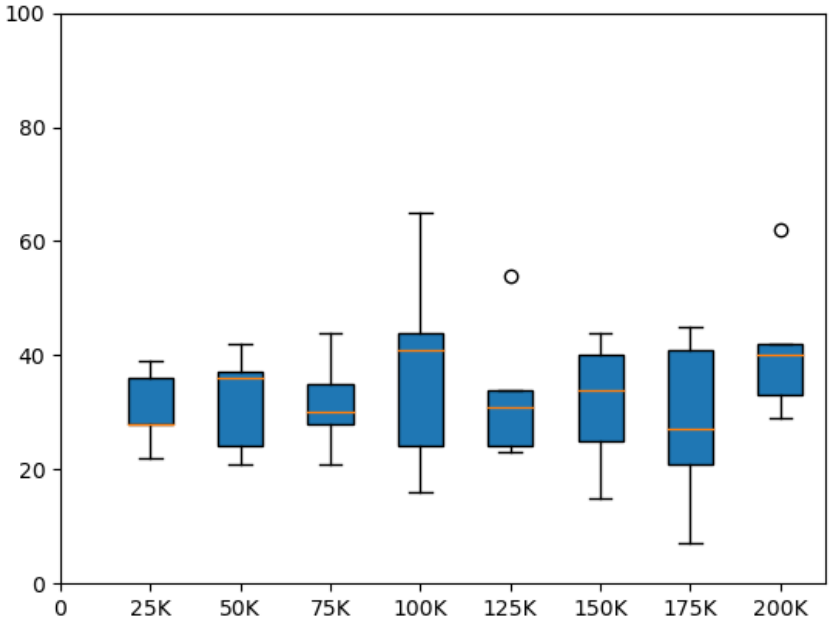


Figure A.7: REINFORCE gamma 0.2 on Rake&Roll-Simple.

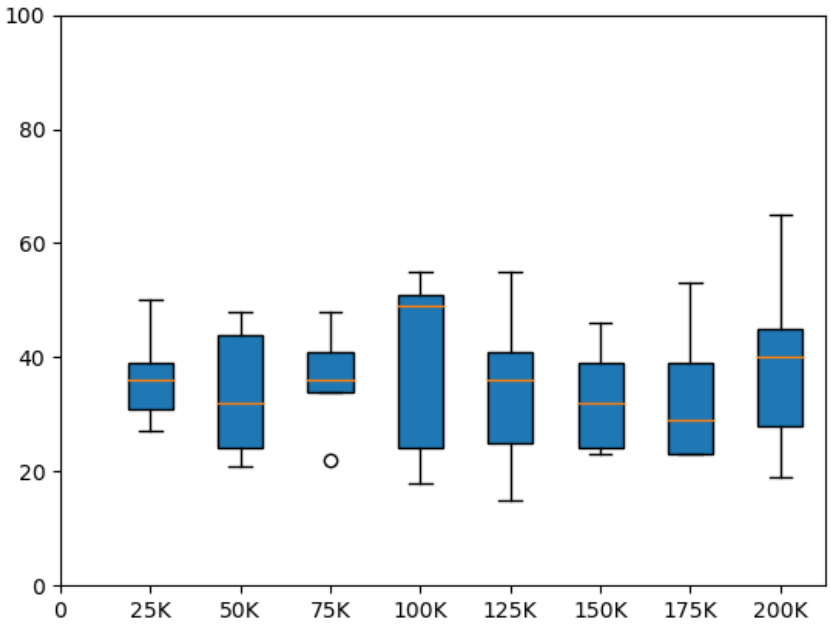


Figure A.8: REINFORCE gamma 0.5 on Rake&Roll-Simple.

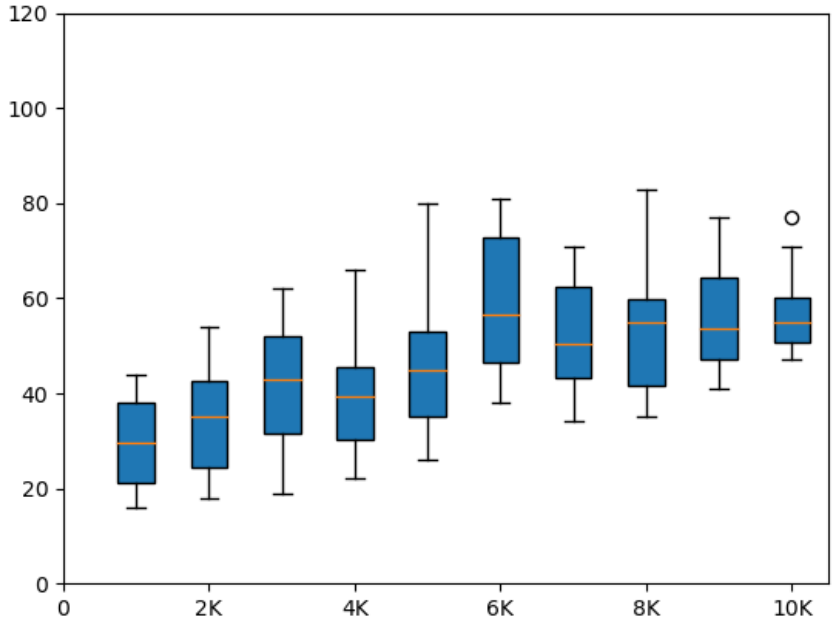


Figure A.9: PerD3QN gamma 0.2 on Rake&Roll-Complete.

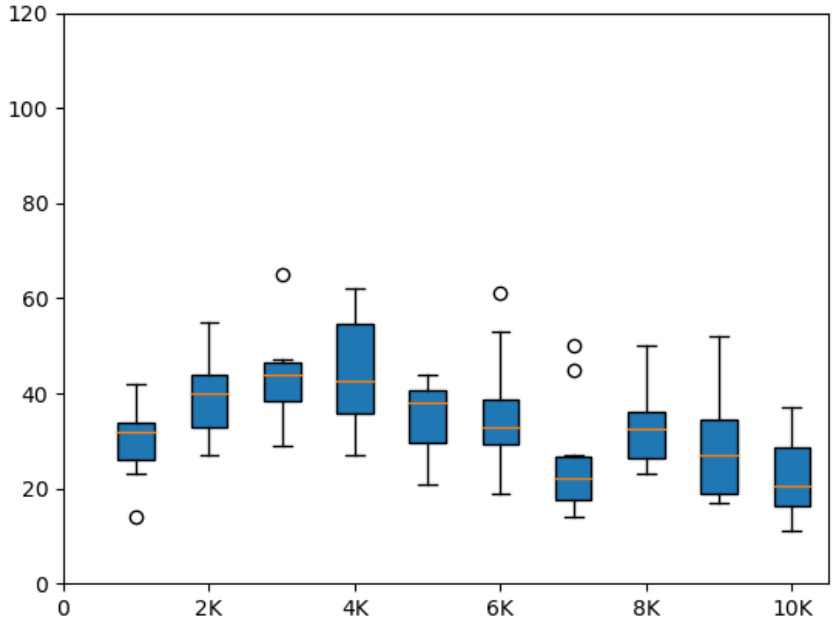


Figure A.10: PerD3QN gamma 0.5 on Rake&Roll-Complete.

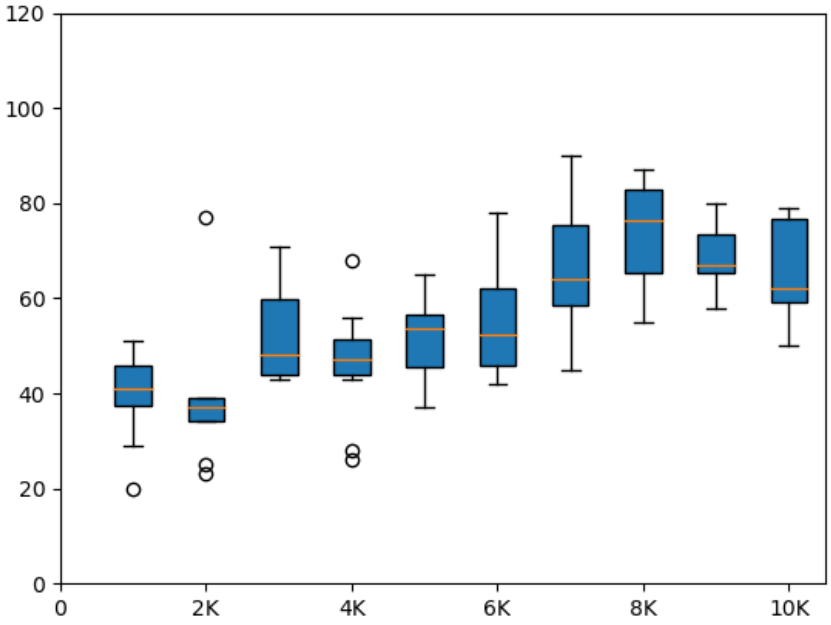


Figure A.11: PerD3QN gamma 0.2 on Rake&Roll-Deterministic.

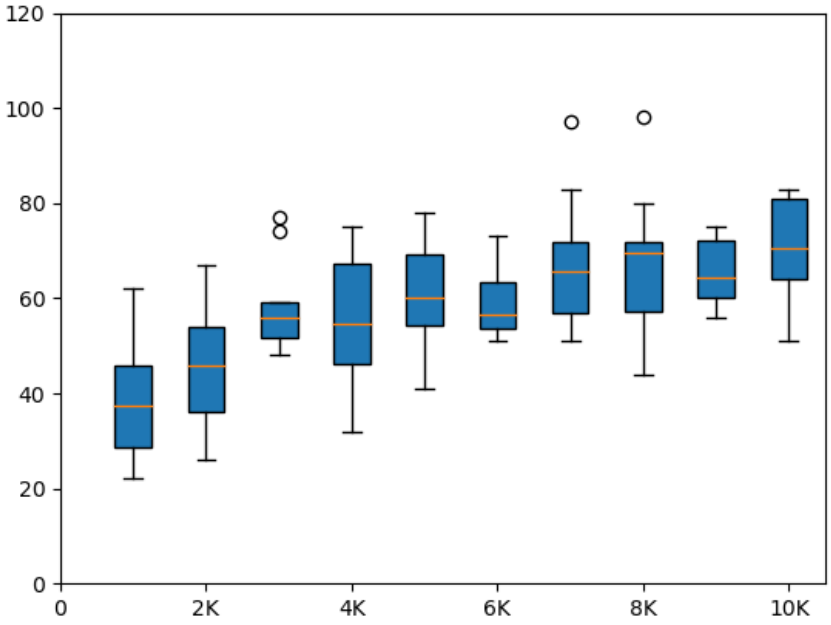


Figure A.12: PerD3QN gamma 0.5 on Rake&Roll-Deterministic.

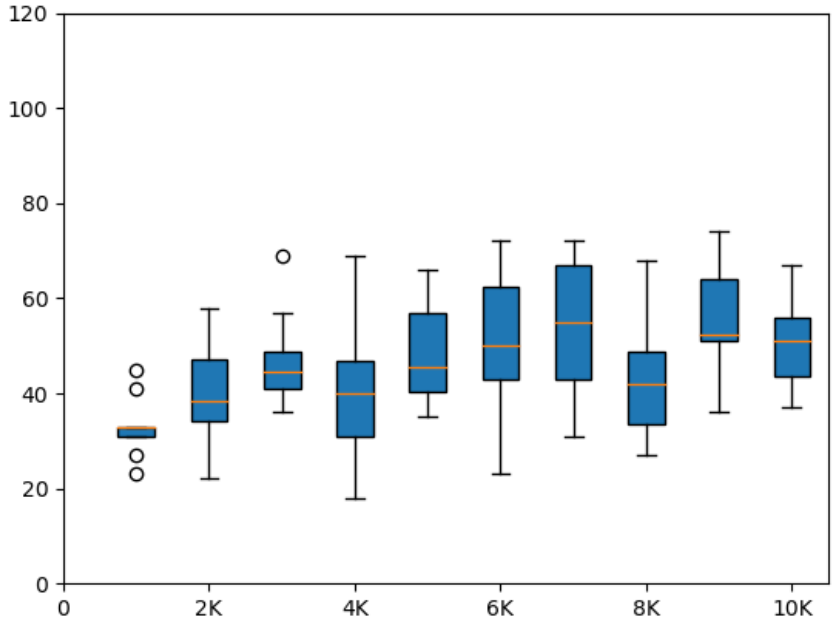


Figure A.13: PerD3QN gamma 0.2 on Rake&Roll-Simple.

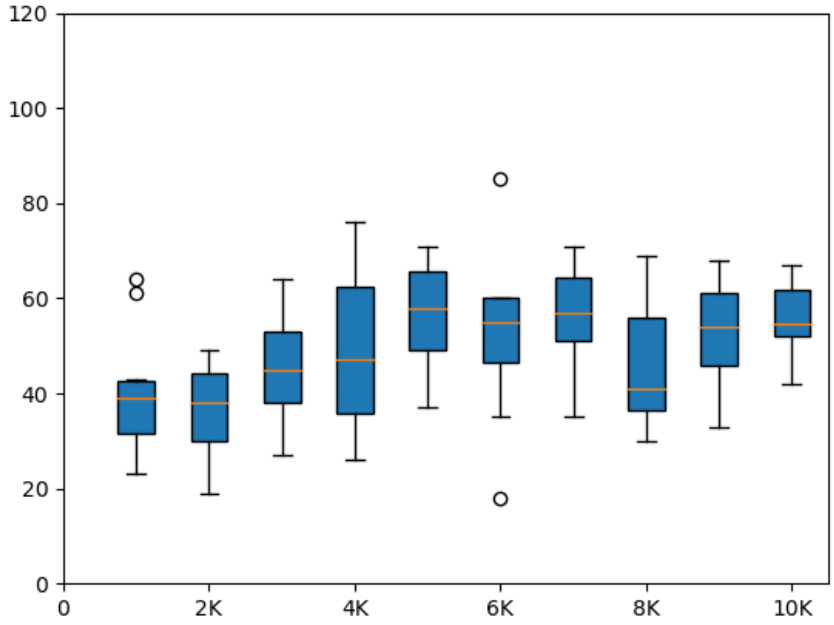


Figure A.14: PerD3QN gamma 0.5 on Rake&Roll-Simple.

Bibliography

Foster, D. (2021), 'How to build your own ai to play any board game'. https://medium.com/applied-data-science/how-to-train-ai-agents-to-play-multiplayer-games-using-self-play-deep-reinforcement-learning-247d0b440717, last visited 21/02/2023.

Häfner, D. (2021), 'Learning to play yahtzee with advantage actor-critic (a2c)'. https://dionhaefner.github.io/2021/04/yahtzotron-learning-to-play-yahtzee-with-advantage-actor-critic/, last visited 21/02/2023.

Jafari, R., Javidi, M. & Kuchaki Rafsanjani, M. (2019), 'Using deep reinforcement learning approach for solving the multiple sequence alignment problem', *SN Applied Sciences* **1**.

Kavukcuoglu, V. M. (2015), 'Methods and apparatus for reinforcement learning'.
URL: *https://patents.google.com/patent/US20150100530A1/en*

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D. & Kavukcuoglu, K. (2016), 'Asynchronous methods for deep reinforcement learning'.
URL: *https://arxiv.org/abs/1602.01783*

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. & Riedmiller, M. (2013), 'Playing atari with deep reinforcement learning'.
URL: *https://arxiv.org/abs/1312.5602*

OpenAI, :, Berner, C., Brockman, G., Chan, B., Cheung, V., Dębiak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., Józefowicz, R., Gray, S., Olsson, C., Pachocki, J., Petrov, M., Pinto, H. P. d. O., Raiman, J., Salimans, T., Schlatter, J., Schneider, J., Sidor, S., Sutskever, I., Tang, J., Wolski, F. & Zhang, S. (2019), 'Dota 2 with large scale deep reinforcement learning'.
URL: *https://arxiv.org/abs/1912.06680*

Russell, S. J. & Norvig, P. (2016), *Artificial Intelligence: A Modern Approach*, third edn, Pearson Education, England.
URL: *https://www.pearson.com/us/higher-education/program/PGM156683.html*

Schaul, T., Quan, J., Antonoglou, I. & Silver, D. (2015), 'Prioritized experience replay'.
URL: *https://arxiv.org/abs/1511.05952*

Schulman, J., Wolski, F., Dhariwal, P., Radford, A. & Klimov, O. (2017), 'Proximal policy optimization algorithms.', *CoRR* **abs/1707.06347**.
URL: *http://dblp.uni-trier.de/db/journals/corr/corr1707.html#SchulmanWDRK17*

Sutton, R. & Barto, A. (2018), *Reinforcement Learning, second edition: An Introduction*, Adaptive Computation and Machine Learning series, MIT Press.
URL: *https://books.google.it/books?id=sWV0DwAAQBAJ*

Tesauro, G. (1995), 'Temporal difference learning and td-gammon', *Commun. ACM* **38**(3), 58–68.
URL: *https://doi.org/10.1145/203330.203343*

van Hasselt, H., Guez, A. & Silver, D. (2015), 'Deep reinforcement learning with double q-learning'.
URL: *https://arxiv.org/abs/1509.06461*

Vinyals, O., Ewalds, T., Bartunov, S., Georgiev, P., Vezhnevets, A. S., Yeo, M., Makhzani, A., Küttler, H., Agapiou, J., Schrittwieser, J., Quan, J., Gaffney, S., Petersen, S., Simonyan, K., Schaul, T., van Hasselt, H., Silver, D., Lillicrap, T., Calderone, K., Keet, P., Brunasso, A., Lawrence, D., Ekermo, A., Repp, J. & Tsing, R. (2017), 'Starcraft ii: A new challenge for reinforcement learning'.
URL: *https://arxiv.org/abs/1708.04782*

Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M. & de Freitas, N. (2015), 'Dueling network architectures for deep reinforcement learning'.
URL: *https://arxiv.org/abs/1511.06581*