HAW HAMBURG

BACHELORTHESIS
Klejda Alushi

Exploration of Reinforcement Learning Methods when Training a Dice Game Playing Agent

FACULTY OF COMPUTER SCIENCE AND ENGINEERING
Department of Information and Electrical Engineering

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

HAMBURG UNIVERSITY
OF APPLIED SCIENCES

Hochschule für Angewandte
Wissenschaften Hamburg

Klejda Alushi

Exploration of Reinforcement Learning Methods when Training a Dice Game Playing Agent

Bachelor Thesis based on the examination and study regulations
for the Bachelor of Engineering degree programme
Bachelor of Science Information Engineering
at the Department of Information and Electrical Engineering
of the Faculty of Engineering and Computer Science
of the University of Applied Sciences Hamburg
Supervising examiner: Prof. Dr. Marc Hensel
Second examiner: Prof. Dr. Ulrike Herster

Day of delivery: 27.10.2022

Klejda Alushi

Title of Thesis

Exploration of Reinforcement Learning Methods when Training a Dice Game Playing Agent

Keywords

Artificial Intelligence, Deep Learning, Reinforcement Learning, Dice Game, Advantage Actor Critic, Policy Gradient

Abstract

This thesis will use different reinforcement learning algorithms to train a neural network to play a dice game. It will analyse how these algorithms are influenced by stochastic processes and by the dimensions of the neural network.

Klejda Alushi

Thema der Arbeit

Erforschung von Reinforcement-Learning-Methoden beim Training eines Würfelspiel-Agenten

Stichworte

Künstliche Intelligenz, Tiefes Lernen, Reinforcement Learning, Würfelspiel, Advantage Actor Critic, Policy Gradient

Kurzzusammenfassung

Diese Bachelorarbeit wird verschiedene Reinforcement-Learning-Algorithmen verwenden, um ein neuronales Netzwerk darauf zu trainieren, ein Wuerfelspiel zu spielen. Es wird analysiert, wie diese Algorithmen durch stochastische Prozesse und durch die Dimensionen des neuronalen Netzes beeinflusst werden.

Contents

List of Figures

List of Tables

Abbreviations

A2C Advantage Actor Critic.

AI Artificial intelligence.

ANN Artificial Neural Network.

MDP Markov Decision Process.

1 Introduction

This chapter will discuss the motivation behind this thesis and the objective that it will work towards, and it will outline the contents of the remaining chapters.

1.1 Motivation

The technological innovations of AI and machine learning have been on the rise in the last few decades. What started a storytelling device, which told of intelligent machines and automatons[1], has developed into an academic discipline that has expanded into major fields such as astronomy, media, healthcare, finance, and so on [25]. During its evolution, AI has carved an important role for itself in the gaming industry, since games present a reproducible, virtual environment with controllable conditions but with many complex challenges and scenarios, in which AI can be trained [20]. Board games, such as chess and go, have also been a focus point in AI, as it is easy to determine the winner of the game, and their state possibilities are fully observable, albeit large [5].

This thesis will focus its efforts on the game Yahtzee being trained by reinforcement learning algorithms. Different deep learning approaches will be compared, and it will be analysed how they are affected by random variables, such as dice rolling. The objective is to use reinforcement learning to train an agent to not only learn the rules of the game, but to also learn from its previous actions and improve its future gameplay.

1.2 Outline

The thesis will start with Chapter 2, which will provide an overview of artificial intelligence and machine learning, with a focus on reinforcement learning fundamentals and

[1] a mechanism that is relatively self-operating [12]

algorithms, and will explain the rules of the game Yahtzee. Chapter 3 will discuss the results of a stakeholder analysis, the possible use cases in the game, and the functional and non-functional requirements. Chapter 4 will detail the software programs and languages used, and the planning of the implementation including the gameplay design and the deep learning algorithms that will be used. Chapter 5 will lay out how the game and the deep learning algorithms were implemented, and will comment on any changes made to the plans from the previous chapter. Chapter 6 will serve as a closing statement to this paper by discussing the results of the implementation, the challenges that arose, and the ways that this work could be improved or built upon, in the future.

2 Background

This chapter will provide an overview of artificial intelligence and the development of some of its subgroups such as machine learning and representation learning. The foundations of artificial neural networks will also be discussed, with an emphasis being placed on reinforcement learning fundamentals and algorithms. The chapter will end by providing some history about the game Yahtzee and will outline its rules.

2.1 Artificial Intelligence and Machine Learning

Artificial Intelligence (AI) is a subsection of computer science, which centers around the creation of machines designed to mimic human intelligence [4], which has been at the forefront of technology in the recent decades, with big leaps being made in the fields of speech and image recognition, recommendation systems, robotics, and so on. In order to better define intelligence, in 1950 Alan Turing created the Turing test, which a computer passes if, after a written question and answer session with an examiner, the examiner does not know whether the answers are coming from a human or a machine [19]. There is also, an extended version of the Turing test, which tests the machine's perceptual abilities and its ability to manipulate physical objects; through these tests the six focus areas of artificial intelligence were defined [19], those being:

- Natural Language Processing
- Knowledge Representation
- Automated Reasoning
- Machine Learning
- Computer Vision
- Robotics

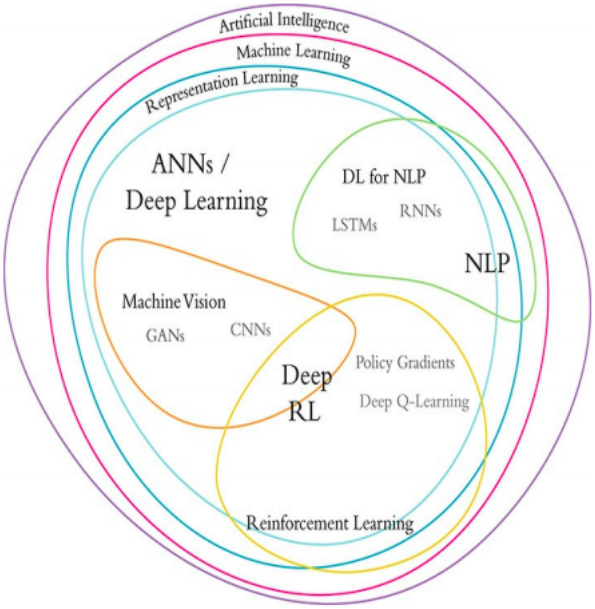

Figure 2.1: Venn diagram of artificial intelligence [10]

Machine learning Machine learning is a subset of AI which focuses on creating software that can teach itself to recognize patterns, thus removing the need of human guidance [4]. Machine learning traditionally focuses a large part of its efforts on feature engineering which consists of processing the raw data, whether that be from text, images, or audio excerpts, into data that could more easily be used by traditional statistical techniques [10]. While feature engineering is important and it allows the use of machine learning algorithms that are not yet equipped to process raw data, it also requires subject-specific knowledge and draws time away from optimizing the machine learning algorithms.

Representation learning Representation learning, on the other hand, stops relying on feature engineering and works to create machine learning models that can automatically transform the raw data into features, and can adapt quickly to changes in that data [10]. As can be seen in Figure 2.1, deep learning, which will be discussed in more detail below, has dominated and led the field of representation learning

2.2 Artificial neural networks

Artificial neural networks (ANN) are a subset of machine learning which were initially created to simulate the natural behaviors of neurons in the brain. Biological neural networks or neural circuits as they might otherwise be called, are collections of thousands of neurons, which communicate with each other by sending and receiving chemical or electrical signals [2]. A neuron, as shown in Figure 2.2, is built up of the cell body (soma), multiple dendrites which receive the signals, and a single axon which sends the signals out [2]. The neurons are not directly in contact with each other, but instead have a small gap between them called the synapse which is what enables the transmission of the signals [2].

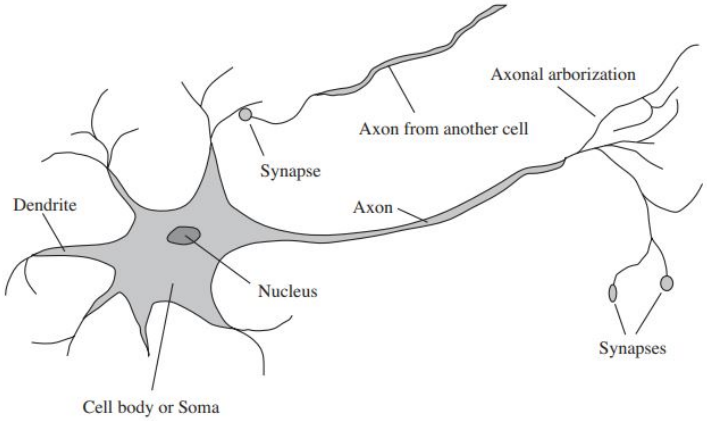

Figure 2.2: Diagram of a biological neuron [19]

Artificial neural networks can be split into single-layer networks, where the input neurons send information directly to the output neurons and multi-layer networks, where the input and output neurons are separated by a number of hidden layers [2]. These multi-layer networks, or deep neural networks, are organized in several layers [10]: the input layer, the output layer, and a certain number of hidden layers. The input layer gathers data from an external source, which may be the pixels in an image the software needs to recognize, or the state in which a game piece is in. Whereas the output layer is responsible for producing the ultimate result. The number and size of the hidden layers are left completely up to the programmer to determine.

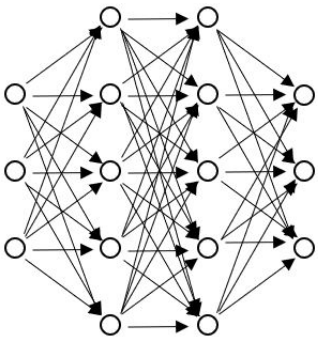

Figure 2.3: Simple structure of a deep neural network with two hidden layers

2.2.1 Types of deep learning

The paragraphs below will detail the three main types of training techniques used in neural networks, which are: supervised learning, unsupervised learning, and reinforcement learning.

Supervised learning Supervised learning is used in neural networks when the desired output is known and is communicated to the network. The training data that the network is provided with, has a label accompanying it which contains the correct output; this enables the network to compare its current output with the correct one and adjust its parameters accordingly. Supervised learning involves two main statistical techniques, regression and classification [14].

- Regression is used when the outcome that should be predicted is a continuous value [14]. An example of this would be predicting the weather or predicting house prices, based on certain input variables.
- Classification is used when the neural network should predict the category in which the input belongs in [14]. The output, in this case, will be a categorical value, such as deciding whether an email is spam or not or recognizing a numerical digit written in an image.

Unsupervised learning Unsupervised learning differs from supervised learning, in that the training data has no label, and there is no fixed outcome that should be predicted.

Instead, the network itself is responsible for finding a pattern or a hidden structure among the training data, and group them accordingly [6]. This type of learning comes in useful for example, in recommending similar products on an e-commerce website, or even in anomaly/error detection.

Reinforcement learning Reinforcement learning has to do with observing how the network, represented by an agent, learns to navigate an environment in order to receive the highest reward [3]. There are no data labels, and the training data looks very different from the two types of learning mentioned above. The training data contains information about the environment in which the agent is in, about the possible actions the agent can take, and the different states in which an agent can be in.

2.3 Reinforcement learning

The goal of reinforcement learning is to train an agent to improve its behaviour in a defined environment. Unlike supervised learning, there are no labelled data sets but instead, the environment provides information about its state and gives feedback after an action is taken. Ultimately, the agent is very autonomous, and is responsible for deciding whether to accept certain feedback, and what information to pay attention to. During the training process, the agent uses trial and error to experiment with different actions and to map out the best path that achieves the set goal. This autonomy proves to be beneficial to reinforcement learning, since it can be used in a wide range of environments where the best solution or the best strategy to solve a problem is not yet known.

To best achieve the goal, it is important to find a strategy that the agent will use to determine what is the best action to take in a certain state; the best action being the one that maximises the possible reward [3]. At a predetermined time in the game, the environment will give the agent positive feedback, which is referred to as a reward, or negative feedback, which is referred to as a punishment.

2.3.1 Agent and Environment

Figure 2.4 shows the interaction and feedback loop between the agent and the environment. At time t the environment is in state s_t. After the agent takes an action a_t, the

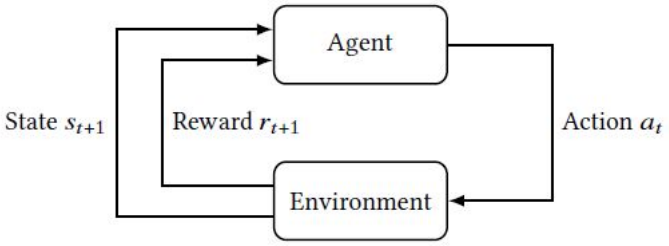

Figure 2.4: Reinforcement learning diagram [16]

environment is then in the state s_{t+1}, which it communicates to the agent, along with the reward s_{t+1}.

The environment can be very broad in nature, and it can be quite difficult to find the best reinforcement learning method that fits with a specific environment. Therefore, in order to better understand and work with the environment, it can be classified into the following categories [19]:

- Fully observable vs. Partially observable
- Deterministic vs. Stochastic
- Single-agent vs. Multi-agent
- Sequential vs. Episodic
- Static vs. Dynamic
- Discrete vs. Continuous
- Known vs. Unknown

Fully observable vs. Partially observable A fully observable environment is one in which the agent can see the state of the whole environment at any point in time, and if this is not possible, then the environment is partially observable. For example, in a chess game the agent can see the state of the board at any point in time, so the environment is fully observable but in a card game with an opponent, the environment is partially observable since the cards of the opponent are not known to the agent.

Deterministic vs. Stochastic A deterministic environment is one where the next state can be determined just from the current state and the action taken [19], this means that all the input variables are known and produce the same output each time. On the other hand, stochastic environments sometimes have random or unpredictable variables, which produce different outputs each time and are not replicable.

Single-agent vs. Multi-agent If an agent is defined as a decision-making entity, then a multi-agent environment is one where there are multiple agents. These agents could be opponents, which would create a competitive multi-agent environment, or they could be working towards the same goal, which creates a cooperative multi-agent environment [19].

Sequential vs. Episodic In sequential environments, the actions taken by the agent depend on the previous actions taken, and in turn can affect any possible future actions. Whereas, in episodic environments actions do not have any effect on each other, and the agent does not need to take into consideration any previous events or future possibilities.

Static vs. Dynamic An environment is considered dynamic if it changes while the agent is considering a decision, otherwise it is considered static. In a dynamic environment, it can be deemed that the environment is constantly asking the agent for a decision, and deliberation can be seen as deciding to not do anything.

Discrete vs. Continuous A discrete environment is one in which the states and the actions are quantitative, whereas a continuous environment may have an infinite number of actions or states. For example, chess, while it may have a very large number of states and actions, has a discrete environment, whereas the environment of a self-driving car is continuous since the position of the entities around it such as other cars or pedestrians change continuously over time.

Known vs. Unknown In a known environment, the agent knows the outcome or the probability of an outcome of a certain action, whereas in an unknown environment the agent does not know what each action does [19].

2.3.2 Markov decision process

Reinforcement learning utilizes sequential decision making, meaning that actions are made one by one in a sequential form, and after each action the agent recalibrates and observes the changes made to the environment [17]. Mathematically, sequential decision making can be represented by a Markov decision process (MDP). The Markov decision process is defined by five components [17]:

- ***S*** represents all the possible states, also known as the state space.
- ***A*** represents all the possible actions, also known as the action space.
- $\boldsymbol{R_a(s,\ s')}$ represents the reward given when going from state s to state s' with action a.
- $\boldsymbol{P_a(s,\ s')}$ stands for probability distribution. It represents the probability that action a at state s at time t will lead to state s' at time $t+1$.
- $\boldsymbol{\gamma}$ (gamma) represents the discount factor. The discount factor is a value between 0 and 1, and is responsible for guiding the agent to favor either immediate or delayed rewards. The lower the discount factor, the more importance is placed on the possible immediate rewards.

2.3.3 Policy and Value functions

In order to achieve the optimal behaviour of the agent, finding the optimal policy function is crucial. A policy, denoted by π, maps each state with the probability of selecting a certain action [21]. If, at time t, the agent is following policy π, then $\pi(a|s)$ shows the probability of choosing action a from action space A when in state s from state space S.

The value function, denoted as $v_\pi(s)$ is also an important part of reinforcement learning, since it starts to take into consideration the possible reward that an agent can receive when in state s and following policy π [21]. The Q-value function, noted as $q_\pi(s, a)$ is also quite important, as it calculates the expected return when taking action a from state s while following policy π [21].

The optimal policy is defined as the policy that is better than or equal to all the other policies. Which mathematically can be expressed as, $\pi \geq \pi'$ if $v_\pi(s) \geq v_{\pi'}(s)$ for $s \in S$

[21]. Unfortunately, in a real environment the optimal policy rarely gets reached, since, for quite a lot of reinforcement learning problems, even with the high-powered computers of today, the memory and computing power needed is too high.

2.4 Reinforcement learning algorithms

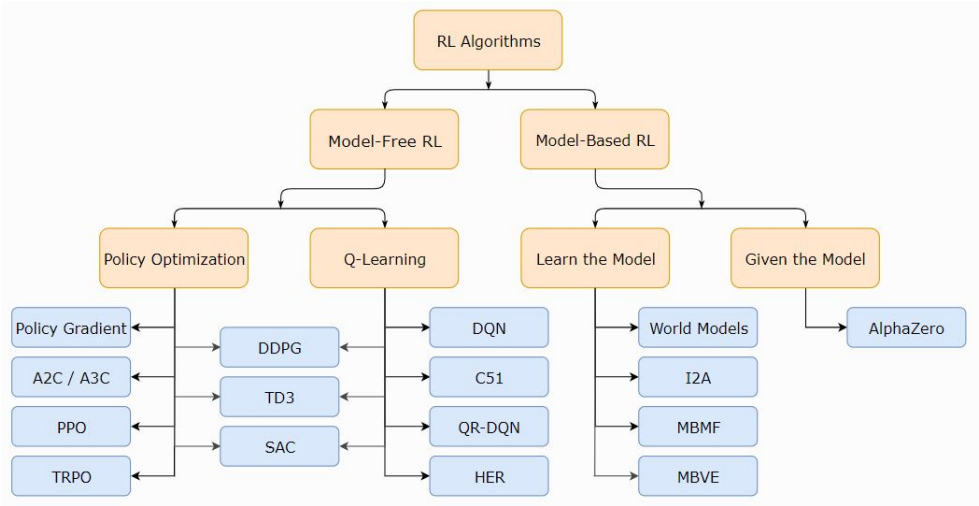

Figure 2.5: Types of reinforcement learning algorithms [1]

There are many algorithms that implement reinforcement learning, and as shown in Figure 2.5, they can be grouped into model-based and model-free learning. A model-based algorithm places an emphasis on planning the strategy beforehand, and tries to acquire a model of the environment, and an understanding of how states interact with each other [7]. Model-free algorithms, on the other hand, emphasize learning; during the course of the training, information about the value of actions and states is stored, which is then used to help predict future actions.

Exploration vs Exploitation The balance between exploration and exploitation is one of the main obstacles in reinforcement learning. When using exploitation as the main learning method, the agent chooses to use a greedy policy, with which the action with the highest probability of a win is always chosen. While this method may sometimes yield a high return, it does create some tunnel vision in the agent, since they will not be encouraged to take risks on less probable options which may produce a higher return. Agents can also use the exploration method, with which they take more risks, and try to

explore many different options in their environment. While this method allows them to have a better view of their environment, it fails to take advantage of the agent's previous experience. One option that combines the above-mentioned methods is the ε-greedy approach, where ε represents the probability of choosing the exploration option. So, there is a ε chance that the agent picks the exploration option, and a $1 - \varepsilon$ chance that the agent picks the exploitative option [13].

2.4.1 Q-learning algorithm

To find the most optimal policy, finding the maximum expected reward for the current state and action is needed. Unfortunately, this method requires very high computational power, which makes it quite troublesome to implement. To find a fix for this issue, the Q-learning method was developed. As mentioned in the previous section, the value function defined by $V_\pi(s)$, lets the agent know how beneficial a certain state is, if from then onward policy π is followed [21]. The Q-value function, denoted as $Q_\pi(s, a)$, builds upon that by utilising state-action pairs, to represent the expected rewards that the agent can receive if action a is taken from state s while following policy π, as shown in Equation (2.1) [23]. Equation (2.2) shows how the Q-value is updated after every time step [23].

$$Q_\pi(s_t, a_t) = E_\pi[G_t | S_t = s, A_t = a] \tag{2.1}$$

$$Q(s_t, a_t) \Leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma \cdot \underset{a}{max}(Q(s_{t+1}, a_t)) - Q(s_t, a_t)] \tag{2.2}$$

r_t = reward received when moving from state s_t to s_{t+1} with action a_t

γ = discount factor of future rewards

$\alpha\gamma\underset{a}{max}(Q(s_{t+1}, a_t))$ = the max reward that can be received from state s_t, weighted by the learning rate and discount rate

Implementation steps

1. The Q-value function $Q(s_t, a_t)$ is initialised with random variables.

2. The agent selects action a from state s using the $Q(s_t, a_t)$ policy.
3. After action a is carried out, the action chosen, the reward received, and the current state of the environment are stored.
4. The new $Q(s_t, a_t)$ is calculated according to the Equation (2.2).
5. If a terminal state is not reached, then steps 2 to 5 are repeated.

2.4.2 Policy Gradient algorithm

Policy based methods learn a policy that can select actions, without first referring to the value function. The policy gradient, which will be looked at in this section, is an algorithm which seeks to find the optimal parameter θ for a stochastic parametric policy $\pi(\theta)$ [8]. The θ parameter, is the parameter for which $\pi(\theta)$ receives the maximum reward. To optimize this parameter, Equation (2.3) is used [15]

$$\theta_{t+1} = \theta_t + \alpha \cdot \nabla J(\theta_t) \tag{2.3}$$

θ_t = current parameter

θ_{t+1} = parameter in the next time step

α = learning rate parameter

$\nabla J(\theta_t)$ = policy gradient, which optimizes the parameter θ

$J(\theta_t)$ is a performance metric that the agent would like to maximise, it can be represented as the value function of the initial state, seen in Equation (2.4).

$$J(\theta_t) = v_{\pi_0}(s_0) \tag{2.4}$$

From this point, the gradient of this performance metric needs to be calculated. This calculation is documented in Sutton und Barto [21] and it leads to Equation (2.5), with G_t representing the total rewards accumulated from step t until the end of the episode.

$$\nabla J(\theta_t) = G_t \cdot \nabla_\theta \ln(\pi(s_t, a_t, \theta) \tag{2.5}$$

Implementation steps

1. The parameter θ is initialized randomly.
2. The agent completes an episode with the policy $\pi(\theta)$, in which the action chosen, the reward received, and the current state of the environment are stored.
3. The stored data is then used to calculate the policy gradient value according to $\nabla\theta(T_n; \pi(\theta_n))$.
4. θ is then updated according to Equation (2.3).
5. If a terminal state is not reached, then steps 2 to 4 are repeated.

2.4.3 Actor Critic algorithm

While the policy gradient algorithm enables the agent to finetune the policy without needing to know the value of each action, it does lead to the agent having a limited view of the environment [8], since it has not learned any specific state-action behaviour. The Actor Critic method combines the policy optimization from the Policy Gradient method, with value optimization from the Q learning method [9]. The policy optimization is represented by the actor agent, who chooses an action based on the $\pi(\theta)$ policy. On the other hand, the value optimization is represented by the critic agent, who evaluates the chosen action and provides feedback as to how or if it should be changed [9].

In the Actor Critic algorithms, the cumulative reward G_t is replaced with another value, depending on the type of the algorithm. The Advantage Actor Critic algorithm, replaces it with the advantage function, denoted in Equation (2.6), with $V(s_t, \phi)$ being the value function which estimates the value of state s, resulting in Equation (2.7) [8].

$$A(s_t, a_t) = G_t - V(s_t, \phi) \tag{2.6}$$

$$\theta_{t+1} = \theta_t + \alpha \cdot A(s_t, a_t) \cdot \nabla_\theta \ln(\pi(s_t, a_t, \theta) \tag{2.7}$$

Implementation steps

1. The parameters θ and the weights of the critic ϕ are initialized randomly.
2. The agent completes an episode with the policy $\pi(\theta)$, in which the action chosen, the reward received, and the current state of the environment are stored.
3. The critic calculates the advantage according to Equation (2.6).
4. The policy gradient is calculated.
5. The θ parameter is updated according to Equation (2.7)
6. The weights ϕ of the critic are minimized with the $A(s_t, a_t)$ loss function.

2.5 Yahtzee

The dice game Yahtzee was patented in the year 1956 by Edwin Lowe, the founder of the E. S. Lowe Company which had popularised the game Bingo [22]. Yahtzee has similarities to certain board games that predated it, those being the games Kniffel from Germany, Generala, and Poker Dice and Cheerio from England [27]. According to Hasbro, the company who owns the rights to the Milton Bradley company which bought the E. S. Lowe company in 1973, the current version of Yahtzee was created by an anonymous, wealthy Canadian couple who initially called it the Yacht game, since they used to play it on their yacht with their friends [22].

2.5.1 Game rules

The main goal in Yahtzee is to attain more points than your opponent by rolling/rerolling a set of dice in order to get the combination with the highest number of points. There are thirteen rounds in the game, since there are thirteen different point categories available, each with their own set of requirements. The players at first roll 5 dice simultaneously, then they have two chances to reroll some or all of the dice. After the three dice throwing rounds, the players must pick the point category that should be used for that round. Each point category must be used only once, and if the dice do not match the requirements needed for that category, then the player gets 0 points for that round.

Table 2.1: Point categories in Yahtzee [27]

Category	Dice Requirements	Points
Aces	Any dice	Total of Ones only
Twos	Any dice	Total of Twos only
Threes	Any dice	Total of Threes only
Fours	Any dice	Total of Fours only
Fives	Any dice	Total of Fives only
Sixes	Any dice	Total of Sixes only
Three of a kind	Three of the same dice	Sum of all dice
Four of a kind	Four of the same dice	Sum of all dice
Full house	Three of one number and two of another	25
Small straight	Four sequential dice	30
Large straight	Five sequential dice	40
Yahtzee	All five of the same dice	50
Chance	Any dice	Sum of all dice

2.5.2 Point System

In Yahtzee, the categories are split into the Upper section and the Lower section; the Upper section consisting of the top six categories[1], and the Lower section consisting of the remaining 7 categories[2]. The points for each category are listed in Table 2.1, but there are also other chances for the player to receive bonus points.

Upper bonus If the total sum of points in the Upper section is higher than or equal to 63, then the player receives an extra 35 points. The number 63 is chosen based on the sum of the Upper section categories, if three of each number is rolled for each corresponding category.

Yahtzee bonus As the name of the game suggests, rolling a Yahtzee is quite important in this game, not only because it is the category with the highest amount of points but also because of the existence of the Yahtzee bonus. The Yahtzee bonus occurs only when the Yahtzee category has been filled, and when a player rolls another Yahtzee, which consists of five of the same dice. If the category has been filled with fifty points, the

[1] Aces, Twos, Threes, Fours, Fives, Sixes
[2] Three of a kind, Four of a kind, Full house, Small straight, Large straight, Yahtzee, Chance

player receives an additional 100 points, whereas if the category has been filled with zero points, the player does not receive any additional points. The player can then move on to pick a category based on the Joker rules.

The Joker rules state, that if the player has rolled a Yahtzee, and has already filled in the Yahtzee point category, then they have to pick the respective category in the Upper section. For example, if the player rolls 5 Threes, then they would have to pick the Threes category. If the respective Upper section category is already filled in, then the player can pick a category of their choice from the Lower section. Whereas, if the respective Upper section category and all categories from the Lower section are filled in, then the player has to enter a 0 in any open Upper section category.

3 Requirements

This section will cover the stakeholder analysis, use case diagrams, and the requirement analysis, including functional/non-functional requirements.

3.1 Stakeholder Analysis

Figure 3.1, shows an analysis of the possible stakeholders based on their influence and interest over this project. There are four main categories available:

Key player = high influence, high interest

- The supervising professor has a hand in the development of the project from the very beginning. He invests more time into getting familiar with the project and the field that it is in, and he takes on a more advisory/editorial role both with the software solution and the written paper. The supervising professor is interested in the following of the scientific methods, which involves the literature being researched correctly and the testing of any claims or hypotheses made, but also in the development of more projects in the deep learning field. With this stakeholder, holding a constant line of communication is key. He needs to be kept up to date with any progress made, and needs to be informed of any blockers or problems that have arisen.
- One of the stakeholders with the most interest in the success of this project is the thesis author. The interest stems not only from her interest in this field, but since this is also the final hurdle before she receives her bachelor's degree certification. Whilst she does not have complete influence over the project, since the evaluation is done by some of the other stakeholders mentioned, she does have full responsibility over the writing of this research paper, and of creating the system itself.

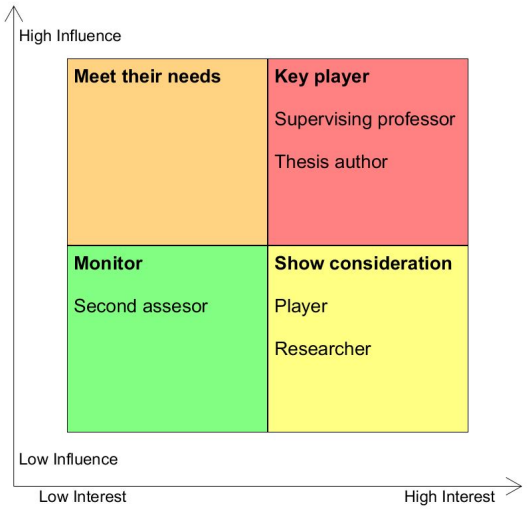

Figure 3.1: Stakeholder analysis, factoring their influence and interest

Meet their needs = high influence, low interest

- At this point in time, there are no stakeholders that could be identified in this category.

Show consideration = low influence, high interest

- Researchers and students who wish to cite the work done in this project, or use it as a foundation in their own work are important stakeholders, with a high interest in its success. The main way to appease them would be by having thorough documentation and well-tested, quality software. Since these stakeholders will most likely come into contact with this project after its completion, there is no concrete line of communication. Instead, information will be communicated to them through the thesis itself, which will detail how the project was created, and any issues that may have emerged along the way.
- Players, who would like to play against the AI agent, also must be shown consideration during the implementation process. In order for the game experience to be problem free for them, the game rules and point calculations must be followed without fault. There should be occasional communication with these stakeholders, since they have a very solution focused view of the project, and might have helpful insights.

Monitor = low influence, low interest

- The thesis assessor assists in the evaluation and the grading of the final project. Since her role starts only once the project has been finished, she, in contrast to the supervising professor, does not have any invested interest in the outcome of the project. Her main interests lie in the engineering methods used, and in learning more about AI and deep learning. The needs of this stakeholder should be monitored, but there is no need for excessive communication.

3.2 Use cases

This section contains an analysis of the most important actions that can be taken during the game, including the main actors involved, the conditions that have to be met, and the steps needed to complete the action. The use case diagram in Figure 3.2 shows an overview of all the possible use cases, which are looked at in more detail below.

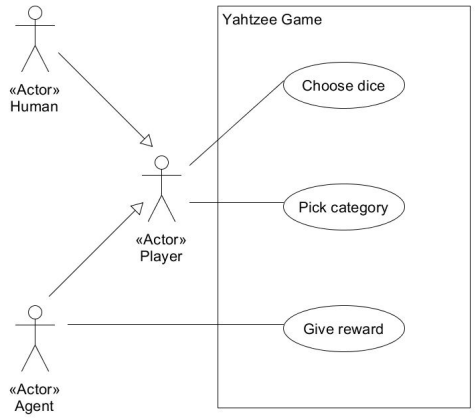

Figure 3.2: Use case diagram

3.2.1 Actors

There are three types of external entities that can interact with the system, which are represented in Figure 3.2 as actors. The actors 'Human' and 'Agent' are specializations of the 'Player' actor, which is involved in the gameplay use cases. The Agent inherits all

the roles of the Player, but is part of other use cases which relate to the training of the neural network.

3.2.2 Use cases

Choose dice Player chooses which dice to keep and which dice to reroll. Player can do this three times during the dice throwing round of the game.

Table 3.1: Use case: Choose dice

Name	Choose dice
Primary Actor	Player
Preconditions	Dice rolling rounds are not over
Postconditions	Game moves to the next round If all dice are kept, the dice rolling rounds are over
Action sequence	Player decides which dice should be kept Other dice are rerolled

Pick category After the dice throwing round, player must pick which point category to use. Player can only do this thirteen times, once for each category.

Table 3.2: Use case: Pick category

Name	Pick category
Primary Actor	Player
Preconditions	Dice rolling rounds are over
Postconditions	Game moves to the next round Points are calculated based on category requirements
Action sequence	Check if category is available Player picks point category

Give reward The agent gets a reward, once a category has been filled or when the game is over.

Table 3.3: Use case: Give reward

Name	Give reward
Primary Actor	Agent
Preconditions	Category has been filled Game is over
Postconditions	Agent receives reward
Action sequence	The score is then measured against reward threshold

3.3 Requirements

This section will discuss the functional and non-functional requirements that will be used as a guide during the implementation process. Functional requirements define a certain functionality of the program, or to state it more simply, they define what a program can do. If any of the functional requirements are not followed, the program will not be successful. Non-functional requirements, on the other hand, are a set of specifications used to determine the operational ability of the program, so they define how a system should perform. Many of the requirements will be based around the rules of the game Yahtzee as described in Section 2.5.1, since they represent the main constraints that the program will face.

3.3.1 Functional Requirements

F1: The player must roll all the dice at the beginning of each round.

F2: The player can reroll the dice a maximum of two times.

F3: The game must end when all the categories have been filled.

3.3.2 Non-functional requirements

NF1: Agent should be trained using reinforcement learning.

NF2 Categories must not be chosen more than once.

NF3: The player can confirm all the dice, according to the game rules mentioned in Section 2.5.1, and move to the category choosing stage.

4 Concept

This chapter will discuss the preparation that was done in order to make the implementation process be carried out correctly. It will delve into the software programs that were used, how the game Yahtzee will be implemented programmatically, and how deep learning concepts will be incorporated.

4.1 Programming Language

The coding of the neural network will be done using the Python language. The reason why Python was chosen is threefold. Firstly, it is a widely used language in the field of machine learning/artificial intelligence, and thus there would be more documentation around its use in various reinforcement learning methods, and it would be significantly easier to find solutions for any future issues that might arise. Secondly, since Python has an extensive set of data collection and data manipulation tools, it would be a good fit for this project, in which the main computational obstacle is the handling of large amounts of data [11]. Lastly, Python is the language in which I have the most proficiency in, having worked with it for a couple of years, so the learning curve for the new terminology would not be as steep.

4.2 Implementation Strategy

4.2.1 Gameplay Design

The basic design of the gameplay is shown in the activity diagram in Figure 4.1, with the actions and their respective action values being shown in Appendix A.1. There are thirteen rounds in the game, representing each of the categories; in the beginning of each round, a check will be carried out to see if there are any empty categories left. In each

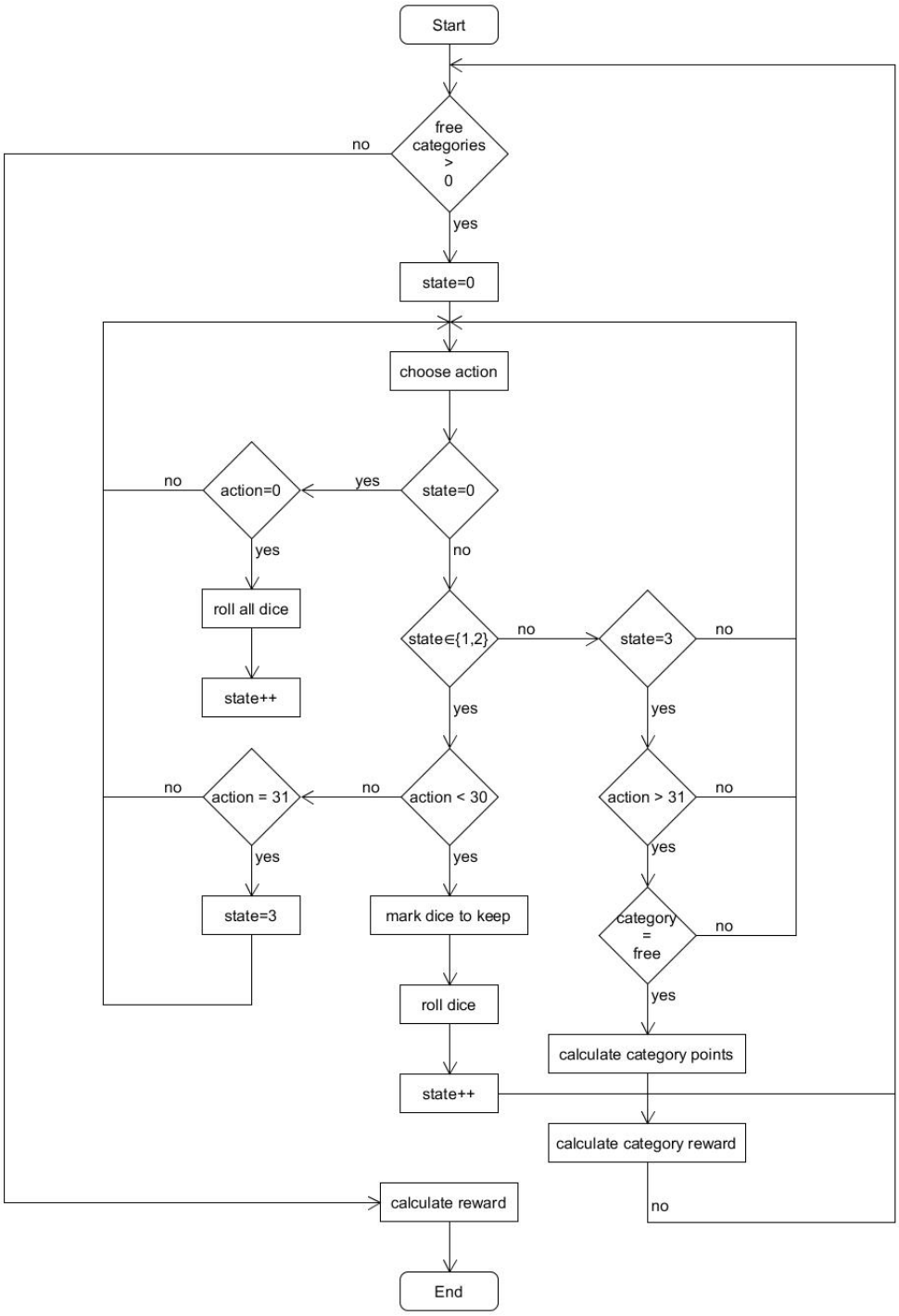

Figure 4.1: Diagram showing a walkthrough of the game

round the agent must roll all the dice at least once, and after each roll they shall decide whether they would like to keep the dice and move on to the category picking stage, or reroll some or all the dice.

Action Space The action space variable defines all the possible actions that can be taken by the player. Throughout the game, the agent will be able to choose between two sets of actions, a dice action or a category action. The dice action will be used to determine which dice should be rerolled and which dice should be kept, whereas a category action means that the agent must choose the point category that should be used. Each action will be represented by a number between 0 and 44, with the dice actions taking values between 0 and 31, and category actions taking values between 32 and 44. The dice action value will be converted into its binary value, with '0' representing the dice to be kept, and '1' representing the dice to be rerolled. For example, the action 25 is equivalent to the binary value of 11001, which means that the first, second and fifth dice will be kept, and the third and fourth dice will be rerolled. The category action values represent one out of the thirteen categories, in the order listed in Table 2.1.

During the action handling process, the question of what will happen when the agent picks an invalid category comes up, and there are multiple ways in which this issue could be handled. Firstly, each turn the agent could be given a new list of valid actions to choose from. Although, this option would be slightly troublesome to implement, since many reinforcement learning methods use a static action space. Secondly, whenever an invalid action is chosen, another random valid action could be chosen in its place. This option might disrupt the learning process of the agent, since it will be hard to predict the outcome of each action. Lastly, whenever an invalid action is chosen, the agent will just be prompted to choose another action.

Game states During a regular[1] gameplay of Yahtzee, players can decide at any point before the second dice reroll, that they would like to pick a point category, instead of continuing to roll the dice. In this version of the game, it was decided that the agent would not have the option to choose between a category or a dice action, but instead would have to either confirm all the dice or run out of rerolls. To make the traversal through these different scenarios easier, the game is split into 4 different game states. The possible game states are shown in Figure 4.2, and the agent can move between them

[1] In this case, regular refers to a game between two human opponents

by picking which dice to keep and which to reroll. This information is given through a Boolean array where '1' means that the dice should be kept, and '0' means that the dice should be rolled again. Thus, with the action [0,0,0,0,0] the agent decides to roll all the dice, and in contrast with the action [1,1,1,1,1] all the dice are kept as they are. Taking a closer look at the states and the state transitions:

- State 0 is used only as a signifier that the game has yet to start. While in this state, the agent cannot take any action since the only action available is rolling all of the dice. After this happens, the agent moves to State 1.
- State 1 is the first chance the agent has, to pick which dice to keep and which to reroll. If the agent chooses to keep all the dice with [1,1,1,1,1], it will move to State 3, and will not be allowed a chance to reroll. However, if the agent choses any option but the aforementioned, it will move to State 2.
- State 2 is the final opportunity the agent gets to change the dice. After this dice reroll, the agent moves to State 3.
- State 3 marks the end of the dice actions and the beginning of the category actions. Only when the agent is in this state, can it choose between any of the remaining point categories.

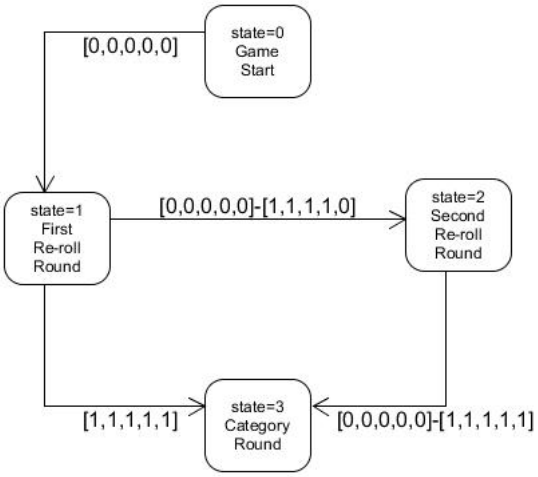

Figure 4.2: Diagram showing the possible states during the dice rolling stage

Observation space The observation space encompasses all the information that the agent needs to know about the environment, so that it can make better choices when choosing an action. The information will be passed to the agent after each action has been taken. The information that will be passed along contains the values of each of the dice, a Boolean[2] value for each category where '0' represents a free category and '1' represents a category that has already been filled, and the current dice state of the game, as shown in Figure 4.3. Using the rule of product [24], the total number of possible states can be calculated which equals to $254,803,968$.

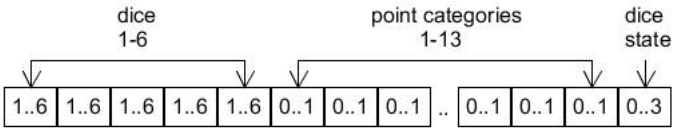

Figure 4.3: Environment observation space

4.2.2 Reward Structure

The reward structure defines when the agent will receive a reward, and on what conditions this reward is based upon. These two parameters need to be chosen carefully, so that the agent can be encouraged to accumulate a higher amount of points in the entire game, and can be discouraged from choosing the category which offers the most amount of points per round, but which could yield a higher amount if saved for later.

Yahtzee is a finite game, so the 'when' could easily be set as the end of the game, and only once all the categories have been filled, will the reward be distributed. Although, since an entire game does last quite long, it might be helpful for the agent to have some intermittent rewards given after each category is chosen.

Setting the reward conditions proves to be slightly trickier, since in this case there is no opponent against which to measure the score. Using an optimal strategy [27] an average score would be around 254 points. If all the categories were filled with their maximum scores, the player would receive a total of 340 points (excluding bonuses), so it can be assumed that an average score is close to 75% of the maximum score. This is the benchmark against which the agent will be measured, and the final reward will be given once the score is higher than 254 points. While, having a score of 254 points

[2]Data with only two possible values, for example 0/1 or True/False

would not be enough for the agent to compete against more advanced players, it does make the agent comparable to an average human, and this threshold could be raised if it is exceeded during the training process. For the intermittent rewards, a high benchmark (75% - 80%) also needs to be used so as to incentivize the agent to save the categories for more fitting dice sets.

4.3 Reinforcement Learning

This game will be trained using reinforcement learning, for which there are many algorithms possible, as detailed in Chapter 2. The algorithms that were considered for the implementation of Yahtzee were the Policy Gradient algorithm, the Q-learning algorithm, which were chosen since they represent two of the main approaches taken in reinforcement learning, the policy-focused method and the value-focused method, and the Actor Critic algorithm, which was designed to be a combination of these two approaches.

The two algorithms that will be implemented are the Policy Gradient algorithm and the Advantage Actor Critic algorithm. The Q-learning algorithm was not chosen, because for it to be effective it has to sample a large amount of states, a large amount of times. As calculated in Section 4.2.1, there are a lot of possible states that the environment could be in, thus for the agent to have a good overview of the results of certain state action pairs, it would take a very long amount of time, and very high computing power. And even if higher computing power was available, the deep Q-learning approach might not be the best approach to a game with a stochastic environment like Yahtzee, since random outcome of the dice rolls makes it difficult to get sufficient data for each possibility.

In the Advantage Actor Critic algorithm, the advantage will be calculated as the difference between the discounted cumulative reward G_t and the value function as shown in Equation (4.1).

$$A(s_t, a_t) = G_t - V(s_t, \phi) \tag{4.1}$$

For both algorithms, the number of hidden layers, and the size of each of those layers will be experimented with during the implementation process. The network will be tested with one, two, and three hidden layers, and the size of the layers will vary throughout.

The effect of these different parameters will be analysed not only in terms of the accumulated rewards but also in the number of steps it takes to complete one training episode which consists of an entire game. Since the game itself contains a very large number of states, it is important to not add too much complexity in the neural networks, which will only add to the computation power needed.

5 Implementation

This chapter will discuss how the implementation process was done, how well the plan defined in Chapter 4 was carried out, and what parts of the plan were changed.

5.1 Tools

The reinforcement learning was carried out using the OpenAI Gym[1] library and PyTorch[2]. OpenAi Gym helps in creating the general outline of the environment, and has a lot of resources relating to the creation of the action and environment spaces. PyTorch is a machine learning framework that eases the use of complex reinforcement learning algorithms, and since it is a Python library, it has a lot of documentation and examples aimed towards beginners [18]. The implementation was done in PyCharm, which is a Python IDE[3] which gives a good overview of the project files, and makes it easier to navigate through them. It also has a debugging mode, in which the program can be suspended at certain points, and the developer can execute commands one at a time while seeing how the variables and the environment change.

5.2 Game Design

To create the game environment, the OpenAI Gym library was used, which already has a makeshift template of the necessary variables and functions needed. The action space remained the same as planned, with 45 different actions representing the dice rolls and the category picks. The observation space, was represented with a variety of Spaces[4]

[1] https://www.gymlibrary.dev/
[2] https://pytorch.org/
[3] Integrated development environment
[4] Data structures defined by the OpenAI Gym library, which help to identify certain values in the learning environment.

[18]. At the end, the structure that was used to store information about the categories and the dice, was the Discrete structure, which can hold the value of any integer between a user-defined minimum and maximum value. At first, the dice values were held in a MultiDiscrete space, which is an array of Discrete values and the categories were held in a MultiBinary space, with the Binary structure being similar to the Discrete structure but having only two possible values. This however caused problems when getting the shape of the observation space, since it meant that multiple arrays were nested within each other, and had to be traversed through in order to find their size.

The categories were initialised in a separate file, which also checked whether the dice met the requirements of the categories, and subsequently calculated the points for the dice, as shown in Figure A.1. To make the lookup of the dice points per category easier during the game, the dice points were stored in a data frame, which had 13 columns representing the categories, and 252 rows representing a unique combination of the dice. The maximum points of each category were also stored, so as to make the calculation of the reward thresholds easier.

5.2.1 Action Handling

In the game of Yahtzee, it is possible to have invalid actions, such as when the agent picks a category that is already chosen, and this procedure needs to be handled programmatically. At first, it was decided to make the action space dynamic[5]; after each time step the possible actions will be evaluated, and only the valid actions would be returned to the agent. This method was chosen, since it would theoretically allow the training to be done faster, but unfortunately, it proved to be quite difficult to combine with most reinforcement learning tools, which use static action spaces. Static action spaces are used, since neural networks in reinforcement learning, base the size of the output layer on the size of the action space, since the neural network will return a probability distribution of which actions should be chosen. This means that when the neural network is initialized, the size of the action space is also declared, and the agent can sample any action from that action space. However, if the environment were to return the valid actions, the size of the action space would be constantly changing. The next action handling method that was chosen, consisted of letting the agent choose from all the actions possible, but if it chose an invalid action then it would be prompted for a new action until a valid one was

[5]Constantly changing, not static

chosen. This method, which ended up being used for the remainder of the training process, proved to be easy to combine with reinforcement learning methods, but extended the time of the training in two main ways:

- Firstly, after each action was chosen, the current state of the environment and the availability of the categories had to be checked, which added unnecessary time to each turn.
- Secondly, since during the training the agent tries to learn which actions have a higher return, it would sometimes have a preference for certain actions that either were invalid or represented a category that had already been filled. This would extend the training time by an extremely high amount, since the agent would loop between 10 to 15 actions that were invalid and ignore all other actions.

5.2.2 Rewards

In Chapter 4, the category rewards would be given out only when the agent scored more than 80% of the maximum category points. This method proved to be ineffective in the early stages of training, thus the threshold was subsequently lowered to 50%, and if the agent scored less than that amount a negative reward or punishment was given, as shown in Figure 5.1. The end of the game reward was also changed and instead two different rewards were given based on how high the accumulated amount of points were. If the agent scored between $50\% - 80\%$ of the maximum amount of points, a reward of $+2$ was given, and if the score was higher than 80% a reward of $+5$ points was given.

5.2.3 Game model

The game Yahtzee has a very large number of paths and actions that can be taken, from deciding which dice to reroll, to weighing out the current and future benefits of each category. In order to reduce this inconvenience, it was planned to first train the model on a minimized version of the game, which would have three dice instead of five, and seven point-categories instead of thirteen. Unfortunately, some categories could only be used with all of the dice, and changing the model would have meant changing the requirements and point calculation of each category. The seven point-category model worked well in the early implementation stages, and it allowed for faster traversal through the different stages in the game, which meant that the entire game could be tested quite quickly and

any errors that may have been made in the implementation could be easily found. It also made it easier to test different approaches for solving a certain problem, for example changing the data structure used for displaying whether the category was filled is much easier to do for seven categories rather than 13. After there were no more errors during the creation of the model or the neural networks, the six other point-categories were added.

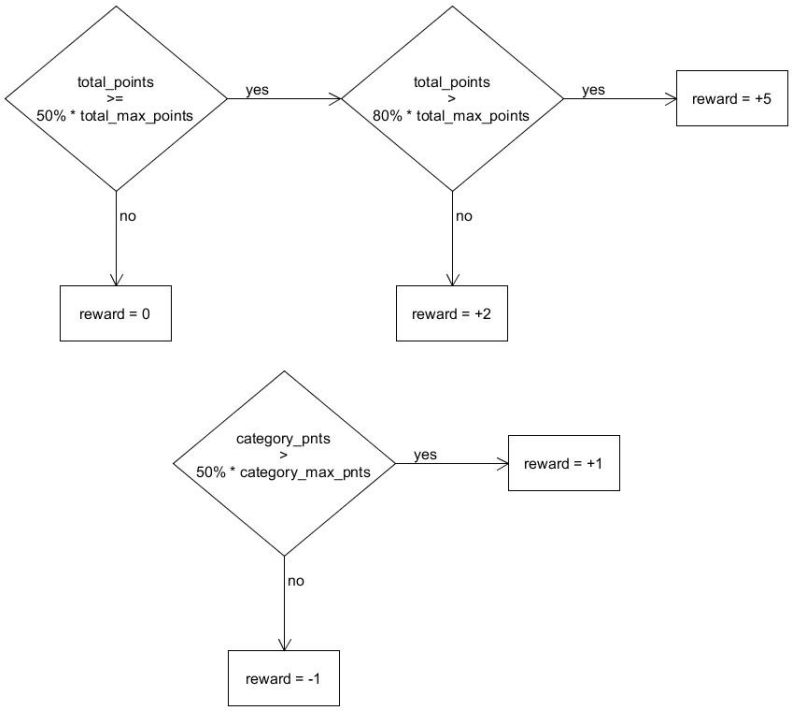

Figure 5.1: Reward system

5.3 Training

5.3.1 Advantage Actor Critic algorithm

This algorithm involves the co-operation of two neural networks, called Actor and Critic, as shown in Figure 5.2. These two networks are nearly identical in their set up, with the only difference being the softmax[6] distribution of the final value in the Actor class. This

[6]Function which is used to convert real values into a probability distribution

is needed, because during the training, the actor learns which actions are more beneficial in certain states and gives those actions a higher probability to be chosen. Whereas the final value of the neural network in the Critic class, is used to define the difference between the actual and the expected outcome.

```
class actor(nn.Module):

    def __init__(self, state_size, action_size):
        super(actor, self).__init__()
        self.state_size = state_size
        self.action_size = action_size
        self.layer1 = nn.Linear(self.state_size, 128)
        self.layer2 = nn.Linear(128, self.action_size)

    def forward(self, state):
        output = F.relu(self.layer1(state))
        output = self.layer2(output)
        output = F.softmax(output, dim=-1)
        return output
```

```
class critic(nn.Module):

    def __init__(self, state_size, action_size):
        super(critic, self).__init__()
        self.state_size = state_size
        self.action_size = action_size
        self.layer1 = nn.Linear(self.state_size, 128)
        self.layer2 = nn.Linear(128, 1)

    def forward(self, state):
        output = F.relu(self.layer1(state))
        value = self.layer2(output)
        return value
```

Figure 5.2: A2C algorithm implementation

As mentioned in Chapter 4, the Actor Critic algorithm will be implemented with the discounted cumulative reward advantage. The algorithm was first tried with one hidden layer with a size of 256, and then with two hidden layers, the first one having a size of 128 and the second having a size of 256. Since the computation power needed is quite high, the first training sessions were run for 100 episodes, an episode consisting of one full game. The two main results that are going to be analysed are the trend of the average rewards per game, and the average amount of steps it took to complete one game, a step consisting of the process of picking an action and checking the validity of the action. The training results can be seen in Figure 5.3.

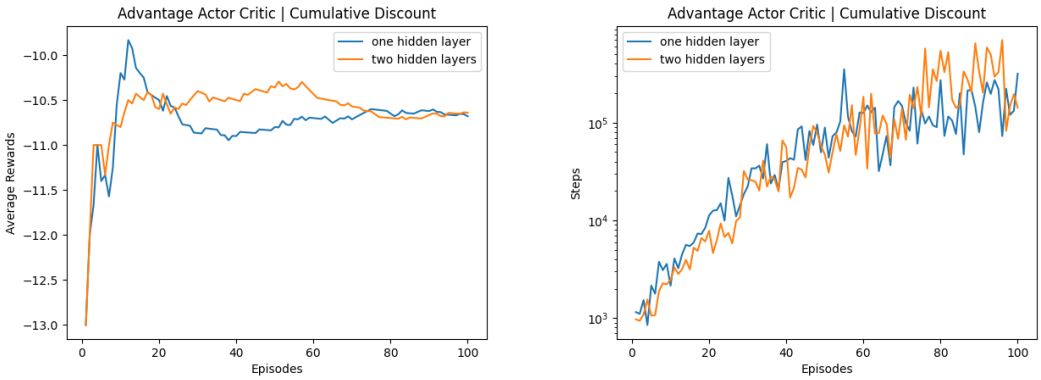

Figure 5.3: Comparison of the A2C algorithm with one/two hidden layers

In an ideal case one game round should take a minimum of 39 steps and a maximum of 52 steps to complete, however in the training sessions the number of steps needed rose exponentially. The reasons for this can be attributed to the action handling process and the static action space, as detailed in Section 5.2. The network with two hidden layers performed especially poorly in this aspect, with some rounds taking more than a million steps to complete. This proved to be a large obstacle in the training process, since the agent needs a large amount of samples of state and action combinations to optimise its action sampling. During the conception phase, it was planned to test out the algorithm with three hidden layers, but based on the factors mentioned above, it was not implemented since it would add a computational strain.

Based on the reward system as shown in Figure 5.1, the maximum cumulative reward would be +18, if the agent manages to receive a +1 reward for each category and a +5 reward for having a high total score, whereas the minimum score or punishment, would be −13 which occurs if the agent gets a −1 penalty for each category. Within 100 episodes, the average reward in all training sessions converged to ≈ -10.5. Both algorithms have a high amount of penalties in the first $20-30$ episodes, but the algorithm with one hidden layer displays a more constant increasing trend in the average reward accumulated.

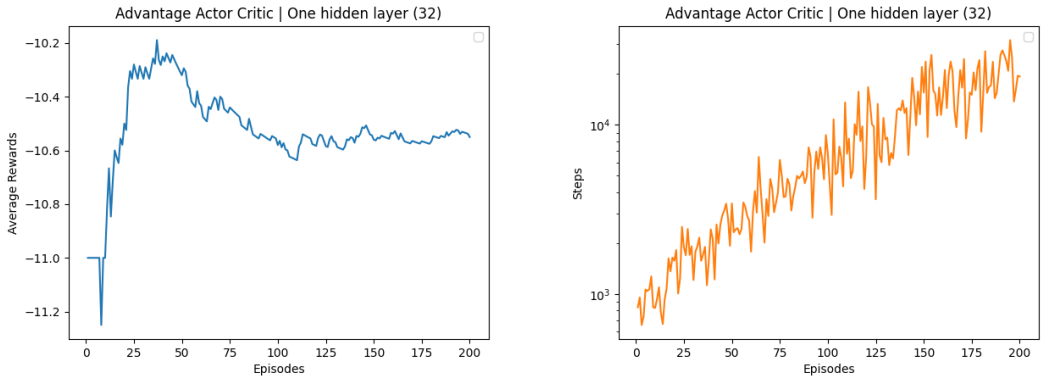

Figure 5.4: A2C algorithm with one hidden layer of 32 nodes

Since the algorithm with one hidden layer with a size of 128 performed slightly better than the one with two hidden layers, it was decided that other training sessions should experiment with different neural network dimensions. Figure 5.4 and Figure 5.5 show the results of the training sessions when the size of the neural network was set to 32 and 64. While the network with a size of 64 shows a steadier increase in the rewards received,

at the end of the 100 episodes the penalties are roughly the same, so these two networks have to be compared further.

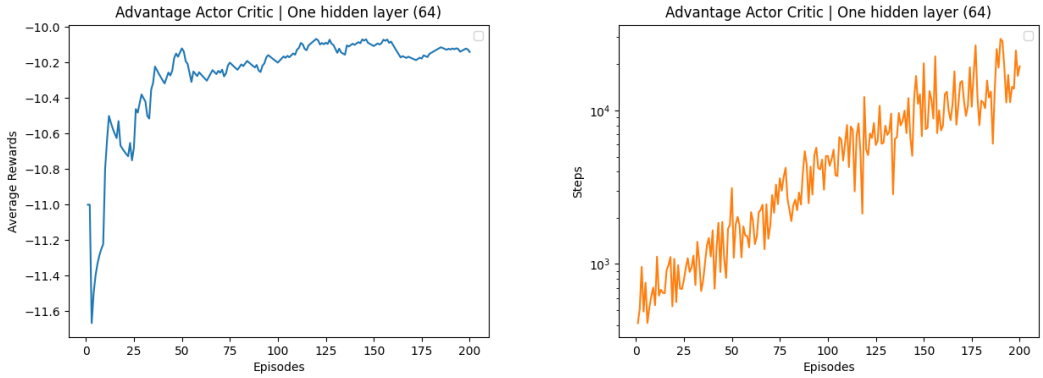

Figure 5.5: A2C algorithm with one hidden layer of 64 nodes

During the training sessions discussed above, the parameter γ (gamma), was set to 0.99. As mentioned in Section 2.3, the closer that γ gets to 1, the more importance is placed on future rewards. In the case of Yahtzee, the simple procedure of picking a category requires a sequence of $3 - 4$ actions, so placing an importance on foresight aids in the training process. During one of the training sessions, γ was set to a very low amount ≈ 0.1, but this proved to be very inefficient, and the agent struggled to complete even 50 episodes. The training process is deemed unfinished when the completion of one episode starts exceeding the one-hour mark, since the constant computing causes a large strain on the system used. A longer computing time is expected in the later stages of the training since the agent has accumulated more data about the different state action pairs, but for it to happen in the early stages of the training means that the learning can not be carried out sufficiently.

5.3.2 Policy Gradient algorithm

The Policy Gradient algorithm's main goal is to find a certain parameter, or set of parameters, for which the agent can achieve the maximum reward. Figure 5.6 shows the implementation of how this optimization is done. The first section of the code snippet calculates the discounted cumulative reward by summing up the total of the rewards stored with γ minimizing the rewards that are further away. The second section starts by collecting the output of the neural network, which is a list of probabilities for each action,

and then uses a PyTorch sampler[7] to choose one action. As mentioned in Equation (2.5) the parameter J is calculated, which combines the discounted cumulative reward with the probability of action a in state s. So that the agent can score the highest amount of points, this parameter needs to be maximised and gradient ascent needs to be performed. A PyTorch optimizer is used to perform the gradient shift shown in Equation (2.3) and then to update the weights of the neural network.

```
total_rewards = 0
discounted_reward_sum = []
rewards.reverse() # rewards stored during the course of one episode
for reward in rewards:
    total_rewards = rewards + gamma * total_rewards
    discounted_reward_sum.append(sum_reward)
discounted_reward_sum.reverse()

# the probability of each action is calculated through the neural network
probability = self(states)
sample = Categorical(probability)
log_probabilities = sampler.log_prob(actions)

J = torch.sum(log_probabilities * discounted_reward_sum)
loss = -J

self.optimizer.zero_grad() # clear previous gradients
loss.backward()
self.optimizer.step()
```

Figure 5.6: Policy Gradient algorithm implementation

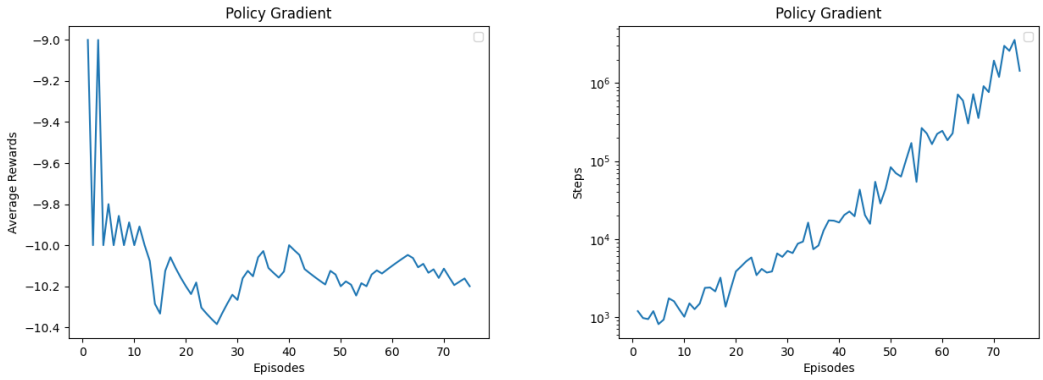

Figure 5.7: Policy Gradient algorithm

The Policy Gradient algorithm contains a network with one hidden layer with a size of 128, and $\gamma = 0.99$, the results of which can be seen in Figure 5.7. The algorithm had a maximum training duration of 100 episodes, but they were not able to be complete.

[7] Categorical

The reason for this could be because the average number of steps needed to complete one game rose up to over 6 digits, but since this also happened during the Actor Critic algorithm training session, the culprit could be the Policy Gradient algorithm itself. Over the course of the episodes, the trend of the average rewards oscillates quite a lot, but at the end of the training session the average lies at ≈ -10.2.

5.3.3 Result comparison

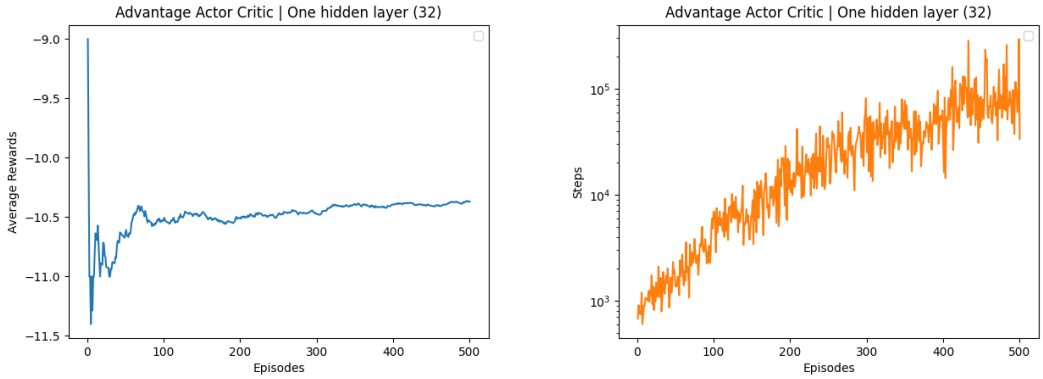

Figure 5.8: Advantage Actor Critic algorithm with one hidden layer (32) | 500 episodes

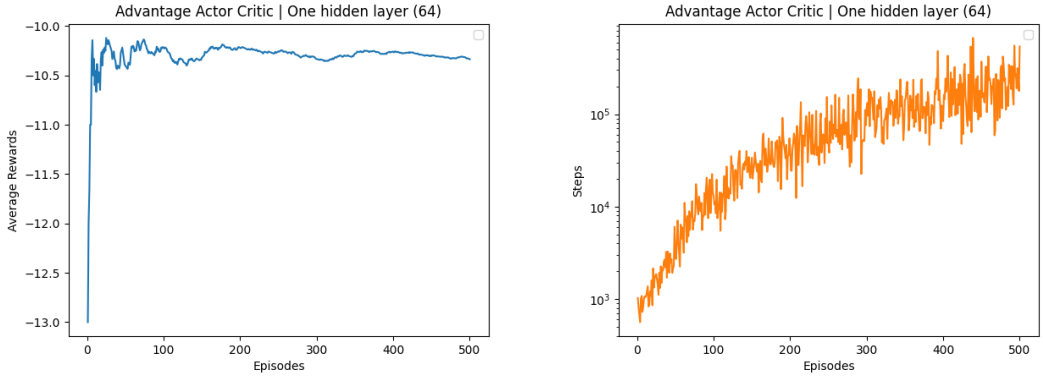

Figure 5.9: Advantage Actor Critic algorithm with one hidden layer (64) | 500 episodes

Based on the results from the experiments above, it is clear that the Advantage Actor Critic algorithm fits the Yahtzee game model better. While it is difficult to make definitive comparisons, since all the algorithms ran for the same amount of time, the Policy Gradient method did prove to be quite inefficient time wise. As for the A2C algorithm,

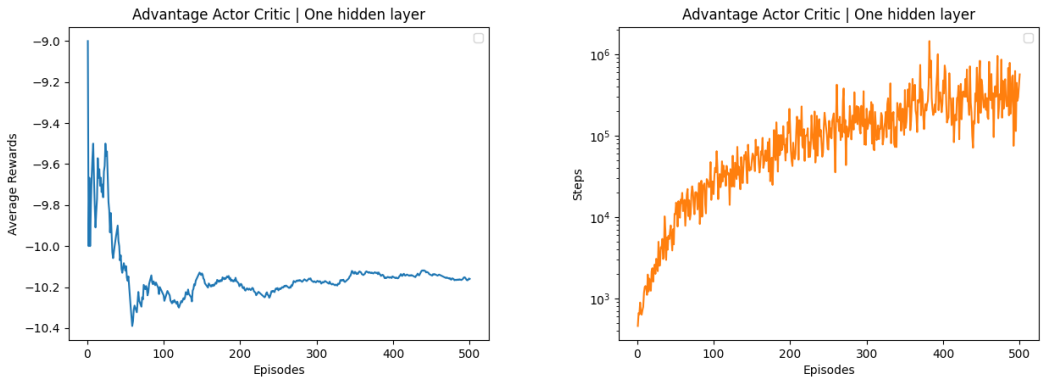

Figure 5.10: Advantage Actor Critic algorithm with one hidden layer (128) | 500 episodes

the networks with one hidden layer of size 32, 64, and 128 were chosen to run for a longer amount of episodes, since in this project the main goal is to see a rising trend in the average rewards received and there does exist a time and computing constraint.

The results of the longer training sessions can be shown in Figure 5.8, Figure 5.9, and Figure 5.10. From these figures it can be seen that the network with a layer of size 64 did not perform well since there was no improvement shown during the 500 episodes. The network with a layer of size 128 showed a slightly larger increase in results compared to the network with a layer of size 32, but the latter managed to complete the training session in a shorter amount of time. What makes choosing between different reinforcement learning algorithms difficult is the risk-benefit trade-off, which is shown well in this case. One algorithm runs faster but has a slightly lower improvement rate, whereas the other has a higher improvement rate, but this comes at the cost of time. In the end, both networks do show improvements over time and they fit quite well with the game model.

6 Summary and Conclusions

This chapter will discuss the overall progress of the thesis and the software solution, any problems that arose and how this work could be continued in the future. The observations and learnings from this project will be split into two sections, one regarding the reinforcement learning process, and the other regarding the conversion of Yahtzee into a software model.

6.1 Yahtzee

The game Yahtzee is quite a simple game in and of itself, since the only things a player needs are five dice and a list of the point-categories. All these elements can easily be programmed, however the difficulties lie in the interactions between them. In the beginning of this process, a lot of emphasis was placed on trying to get the agent to simulate a human player; an unfortunate consequence of this ended up being that the agent was required to make a lot of unnecessary decisions during the game. A human player might take shortcuts through certain decisions or bypass them completely, but the agent had to make a clear, well thought out decision each time, which in some cases hindered its progress. Some examples of these decisions are mentioned below:

- As mentioned in Chapter 5, handling invalid actions was quite difficult to do, and the method chosen ended up slowing down the training process immensely. Most reinforcement learning algorithms that were researched, use static action spaces, so the action handling had to be built around it. However, if this project was built upon in the future, spending more time trying to find a reinforcement learning method that can accommodate dynamic action spaces, would be a large improvement. This way the valid actions can be communicated directly to the agent, which allows for faster gameplay.

- At the beginning of each round, the agent would have to make the decision to roll all five dice, which translates to picking the action $a = 0$. This also added to the time needed for the training sessions, since during the early stages, the agent does not know that each round should begin with this action, so it would keep trying to sample randomly from the action space.
- During the first two dice rolling rounds, the agent can choose to exit the dice rolling round and enter the category picking round, by choosing to keep all the dice, with the action $a = 31$. This was created to mimic human gameplay, since a human player would mentally confirm the dice before picking a category. This feature is quite an important one to have, since it does not add much to the computation power needed, and it allows the agent to figure out the shortcut of confirming the dice earlier on, on its own. If the agent could easily skip from the first dice roll round to the category picking round, then it would focus on getting the rewards as quick as it can, instead of using the benefits that the second dice roll provides.

6.2 Reinforcement learning

This project was the first foray of the author into the field of deep learning, and reinforcement learning in particular. This inexperience in working with deep learning, meant that choosing the optimal reinforcement learning method was challenging. While theoretically, all reinforcement learning methods could have been used, having knowledge of more modern methods would have made the training process quicker and more efficient. Below, some issues that arose during the implementation of the reinforcement learning algorithm will be listed:

- During the training sessions, networks with one or two hidden layers with sizes of 128 or 256 were used, which was mostly to keep the training time low. Other experiments could be done with even larger neural networks, both in terms of size and number of layers, and it could be observed how those factors affect the learning performance
- In the implementation phase, most of the training sessions had a length of 100 episodes, which led to the agent not having a good overview of the possible states and actions, and thus not correcting its performance. If the training sessions had lasted for thousands of episodes, the agent would have had more learning time, and

the efficiency of the different reinforcement learning algorithms could be compared better.

- One resource which was not used enough during the implementation, was the hpc cluster available at the university. HPC (high-performance computing) clusters use a network of computers or servers to allow the resolving of complex problems [26]. The author had not worked with hpc clusters before, and found it quite difficult to navigate the environment and to transfer the files from the Python development environment to the file system used by the hpc cluster.
- The development of the reinforcement learning aspect of this project could also have been started earlier. More focus was spent on making sure the game model was developed correctly, and that it communicated the rules of the game clearly to the agent. This led to certain design mistakes being revealed too late in the training process; therefore, developing the game model and the deep learning concurrently would have been beneficial.

6.3 Conclusion

At the beginning of this research paper, the goal that was stated, was to train the agent to learn the rules of the game Yahtzee and to learn from its previous mistakes and try to improve its performance. This section will discuss whether the project met this goal and whether the goal was correctly set.

The first part of this goal refers to both the game model and the agent. The game model creates and communicates the rules to the agent, who in turn is responsible for using deep learning to try and remember those rules so that the lowest number of invalid actions are chosen. At the end of the implementation stage the agent still required a very large amount of steps to complete one game, which means that this part of the goal has not yet been achieved. The second part of the goal applies to only the agent and the neural networks that it contains. As was shown in Chapter 5, there were a lot of different factors which affected the performance of the agent. Although, while the agent did still receive penalties during the training session, it did manage to improve its performance and increase its rewards over the training period. Thus, the second part of the goal has been achieved.

Taking into consideration the author's inexperience with working with deep learning, this goal might have been a bit too ambitious. With a game this large in size, and a variety of different reinforcement learning algorithms that could be used, it would take a lot of resources for the agent to significantly improve its gameplay in a relatively short amount of time. The points mentioned in the first two sections, show that even though there were issues in the development process, the game Yahtzee is capable of being learned by a reinforcement learning agent, and this thesis could serve as a foundation for other future works. The author would advise future researchers or students who try and build upon this project, to develop the game model and the deep learning aspects concurrently, and to focus on deep learning algorithms, that while may be more complex or show less improvement over time, might also fit the game model better.

Bibliography

[1] Achiam, J. and Abbeel, P.: *Part 2: Kinds of RL Algorithms.* – URL https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html. – [Online; accessed 12-October-2022]

[2] Aggarwal, C.C.: *Neural Networks and Deep Learning: A Textbook.* Springer International Publishing, 2018. – URL https://books.google.de/books?id=AsTswQEACAAJ. – ISBN 9783319944647

[3] Arulkumaran, Kai ; Deisenroth, Marc P. ; Brundage, Miles ; Bharath, Anil A.: Deep reinforcement learning: A brief survey. In: *IEEE Signal Processing Magazine* 34 (2017), Nr. 6, S. 26–38

[4] Bhattacharyya, S. ; Snasel, V. ; Hassanien, A.E. ; Saha, S. ; Tripathy, B.K.: *Deep Learning: Research and Applications.* De Gruyter, 2020 (De Gruyter Frontiers in Computational Intelligence). – URL https://books.google.de/books?id=OlXrDwAAQBAJ. – ISBN 9783110670929

[5] Campbell, Murray: Mastering board games. In: *Science* 362 (2018), Nr. 6419, S. 1118–1118

[6] Dike, Happiness U. ; Zhou, Yimin ; Deveerasetty, Kranthi K. ; Wu, Qingtian: Unsupervised learning based on artificial neural network: A review. In: *2018 IEEE International Conference on Cyborg and Bionic Systems (CBS)* IEEE (Veranst.), 2018, S. 322–327

[7] Gläscher, Jan ; Daw, Nathaniel ; Dayan, Peter ; O'Doherty, John P.: States versus rewards: dissociable neural prediction error signals underlying model-based and model-free reinforcement learning. In: *Neuron* 66 (2010), Nr. 4, S. 585–595

[8] Gridin, I.: *Practical Deep Reinforcement Learning with Python: Concise Implementation of Algorithms, Simplified Maths, and Effective Use of TensorFlow and*

PyTorch (English Edition). BPB Publications, 2022. – URL https://books.google.de/books?id=9Gh7EAAAQBAJ. – ISBN 9789355512055

[9] Grondman, Ivo ; Busoniu, Lucian ; Lopes, Gabriel A. ; Babuska, Robert: A survey of actor-critic reinforcement learning: Standard and natural policy gradients. In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42 (2012), Nr. 6, S. 1291–1307

[10] Krohn, J. ; Beyleveld, G. ; Bassens, A.: *Deep Learning Illustrated: A Visual, Interactive Guide to Artificial Intelligence*. Pearson Education, 2019 (Addison-Wesley Data & Analytics Series). – URL https://books.google.de/books?id=J1enDwAAQBAJ. – ISBN 9780135121726

[11] Kumar, Arun ; Panda, Supriya P.: A survey: how Python pitches in it-world. In: *2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon)* IEEE (Veranst.), 2019, S. 248–251

[12] Merriam-Webster: *The Merriam-Webster Dictionary*. Merriam-Webster, Incorporated, 2022. – URL https://books.google.de/books?id=CIqQzgEACAAJ. – ISBN 9780877790952

[13] Mnih, Volodymyr ; Kavukcuoglu, Koray ; Silver, David ; Graves, Alex ; Antonoglou, Ioannis ; Wierstra, Daan ; Riedmiller, Martin: Playing atari with deep reinforcement learning. In: *arXiv preprint arXiv:1312.5602* (2013)

[14] Nasteski, Vladimir: An overview of the supervised machine learning methods. In: *Horizons. b* 4 (2017), S. 51–62

[15] Plaat, A.: *Learning to Play: Reinforcement Learning and Games*. Springer International Publishing, 2021. – URL https://books.google.de/books?id=k4-0zgEACAAJ. – ISBN 9783030592400

[16] Plaat, A.: *Deep Reinforcement Learning*. Springer Nature Singapore, 2022. – URL https://books.google.de/books?id=06V0EAAAQBAJ. – ISBN 9789811906381

[17] Puterman, Martin L.: Markov decision processes. In: *Handbooks in operations research and management science* 2 (1990), S. 331–434

[18] Ravichandiran, Sudharsan: *Hands-on reinforcement learning with Python: master reinforcement and deep reinforcement learning using OpenAI gym and TensorFlow*. Packt Publishing Ltd, 2018

[19] Russell, S. ; Norvig, P.: *Artificial Intelligence: A Modern Approach*. CreateSpace Independent Publishing Platform, 2016. – URL https://books.google.de/books?id=PQI7vgAACAAJ. – ISBN 9781537600314

[20] Shao, Kun ; Tang, Zhentao ; Zhu, Yuanheng ; Li, Nannan ; Zhao, Dongbin: A survey of deep reinforcement learning in video games. In: *arXiv preprint arXiv:1912.10944* (2019)

[21] Sutton, R.S. ; Barto, A.G.: *Reinforcement Learning, second edition: An Introduction*. MIT Press, 2018 (Adaptive Computation and Machine Learning series). – URL https://books.google.de/books?id=sWV0DwAAQBAJ. – ISBN 9780262039246

[22] Walsh, T.: *Timeless Toys: Classic Toys and the Playmakers Who Created Them*. Andrews McMeel Publishing, 2005. – URL https://books.google.de/books?id=jftapGDTmYUC. – ISBN 9780740755712

[23] Wang, Zhikang T. ; Ueda, Masahito: A convergent and efficient deep q network algorithm. In: *arXiv preprint arXiv:2106.15419* (2021)

[24] Wikipedia contributors: *Rule of product — Wikipedia, The Free Encyclopedia*. 2021. – URL https://en.wikipedia.org/w/index.php?title=Rule_of_product&oldid=1061913371. – [Online; accessed 16-October-2022]

[25] Wikipedia contributors: *Applications of artificial intelligence — Wikipedia, The Free Encyclopedia*. 2022. – URL https://en.wikipedia.org/w/index.php?title=Applications_of_artificial_intelligence&oldid=1114076534. – [Online; accessed 8-October-2022]

[26] Wikipedia contributors: *High-performance computing — Wikipedia, The Free Encyclopedia*. 2022. – URL https://en.wikipedia.org/w/index.php?title=High-performance_computing&oldid=1112386275. – [Online; accessed 16-October-2022]

[27] Wikipedia contributors: *Yahtzee — Wikipedia, The Free Encyclopedia*. 2022. – URL https://en.wikipedia.org/w/index.php?title=Yahtzee&oldid=1102071095. – [Online; accessed 18-August-2022]

A Appendix

A.1 Action Values

Table A.1 maps the actions values in the action space with what the actions concretely represent in the game.

Table A.1: Dice Actions in Yahtzee

Action Value	**Action Type**	**Description**
0	[0,0,0,0,0]	All dice are rolled
1	[0,0,0,0,1]	Dice 5 is kept
2	[0,0,0,1,0]	Dice 4 is kept
3	[0,0,0,1,1]	Dice 4, 5 are kept
4	[0,0,1,0,0]	Dice 3 is kept
5	[0,0,1,0,1]	Dice 3, 5 are kept
6	[0,0,1,1,0]	Dice 3, 4 are kept
7	[0,0,1,1,1]	Dice 3, 4, 5 are kept
8	[0,1,0,0,0]	Dice 2 is kept
9	[0,1,0,0,1]	Dice 2, 5 are kept
10	[0,1,0,1,0]	Dice 2, 4, are kept
11	[0,1,0,1,1]	Dice 2, 4, 5 are kept
12	[0,1,1,0,0]	Dice 2, 3 are kept
13	[0,1,1,0,1]	Dice 2, 3, 5 are kept
14	[0,1,1,1,0]	Dice 2, 3, 4 are kept
15	[0,1,1,1,1]	Dice 2, 3, 4, 5 are kept
16	[1,0,0,0,0]	Dice 1 is kept
17	[1,0,0,0,1]	Dice 1, 5 are kept
18	[1,0,0,1,0]	Dice 1, 4 are kept

19	[1,0,0,1,1]	Dice 1, 4, 5 are kept
20	[1,0,1,0,0]	Dice 1, 3 are kept
21	[1,0,1,0,1]	Dice 1, 3,5 are kept
22	[1,0,1,1,0]	Dice 1, 3, 4 are kept
23	[1,0,1,1,1]	Dice 1, 3, 4, 5 are kept
24	[1,1,0,0,0]	Dice 1, 2 are kept
25	[1,1,0,0,1]	Dice 1, 2, 5 are kept
26	[1,1,0,1,0]	Dice 1, 2, 4 are kept
27	[1,1,0,1,1]	Dice 1, 2, 4, 5 are kept
28	[1,1,1,0,0]	Dice 1, 2, 3 are kept
29	[1,1,1,0,1]	Dice 1, 2, 3, 5 are kept
30	[1,1,1,1,0]	Dice 1, 2, 3, 4 are kept
31	[1,1,1,1,1]	All dice are kept
32	Category 01	Aces
33	Category 02	Twos
34	Category 03	Threes
35	Category 04	Fours
36	Category 05	Fives
37	Category 06	Sixes
38	Category 07	Three of a kind
39	Category 08	Four of a kind
40	Category 09	Full house
41	Category 10	Small straight
42	Category 11	Large straight
43	Category 12	Yahtzee
44	Category 13	Chance

A.2 Point-categories

fig. A.1 shows the creation of the table that maps all unique dice combinations to their points for each category. The seven if-statements go through each dice combination, and check whether they fit the requirement for each category. If they do not, the points

are set to 0, and if they do, the points are calculated based on the rules mentioned in Table 2.1. There are three possible point calculations:

- The points are a set value, for example, the Yahtzee category.
- The points are the sum of all of the dice.
- The points are the sum of one specific dice value, for example, the Threes category sums up all the dice that have a value of 3.

```
dice_points = pd.DataFrame(columns=['Ones', 'Twos', 'Threes', 'Fours', 'Fives', 'Sixes',
                                    'Three of a kind', 'Four of a kind', 'Full house',
                                    'Small Straight', 'Large Straight', 'Yahtzee', 'Chance'])
dice_points['Dice'] = pd.Series(all_dice_combos())
dice_points = dice_points.fillna(0)

for i in range(len(dice_points)):

    # any dice set is valid
    if requirement_any_dice():
        dice_points.at[i, 'Ones'] = points_sum_of_one_dice(dice_points.iloc[i]['Dice'], 1)
        dice_points.at[i, 'Twos'] = points_sum_of_one_dice(dice_points.iloc[i]['Dice'], 2)
        dice_points.at[i, 'Threes'] = points_sum_of_one_dice(dice_points.iloc[i]['Dice'], 3)
        dice_points.at[i, 'Fours'] = points_sum_of_one_dice(dice_points.iloc[i]['Dice'], 4)
        dice_points.at[i, 'Fives'] = points_sum_of_one_dice(dice_points.iloc[i]['Dice'], 5)
        dice_points.at[i, 'Sixes'] = points_sum_of_one_dice(dice_points.iloc[i]['Dice'], 6)

        dice_points.at[i, 'Chance'] = points_sum_of_all_dice(dice_points.iloc[i]['Dice'])

    # dice set must have 3 of a certain dice value
    if requirement_same_dice(dice_points.iloc[i]['Dice'], 3):
        dice_points.at[i, 'Three of a kind'] = points_sum_of_all_dice(dice_points.iloc[i]['Dice'])

    # dice set must have 4 of a certain dice value
    if requirement_same_dice(dice_points.iloc[i]['Dice'], 4):
        dice_points.at[i, 'Four of a kind'] = points_sum_of_all_dice(dice_points.iloc[i]['Dice'])

    # dice set must have 5 of a certain value dice
    if requirement_same_dice(dice_points.iloc[i]['Dice'], 5):
        dice_points.at[i, 'Yahtzee'] = 50

    # dice set must have 2 of a certain dice value and 3 of another
    if requirement_same_dice_combination(dice_points.iloc[i]['Dice']):
        dice_points.at[i, 'Full house'] = 25

    # dice set must have 4 sequential dice
    if requirement_sequential_dice(dice_points.iloc[i]['Dice'], 4):
        dice_points.at[i, 'Small Straight'] = 30

    # dice set must have 5 sequential dice
    if requirement_sequential_dice(dice_points.iloc[i]['Dice'], 5):
        dice_points.at[i, 'Large Straight'] = 40
```

Figure A.1: Creation of table that holds the points for all dice combinations

Declaration

I declare that this Bachelor Thesis has been completed by myself independently without outside help and only the defined sources and study aids were used.

City	Date	Signature