LEARNING A PARTIALLY-OBSERVABLE CARD GAME HEARTS USING REINFORCEMENT LEARNING

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

BUĞRA KAAN DEMIRDÖVER

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

JUNE 2020

Approval of the thesis:

LEARNING A PARTIALLY-OBSERVABLE CARD GAME HEARTS USING REINFORCEMENT LEARNING

submitted by **BUĞRA KAAN DEMIRDÖVER** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Halil Kalıpçılar
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Halit Oğuztüzün
Head of Department, **Computer Engineering** _____

Prof. Dr. Ferda Nur Alpaslan
Supervisor, **Computer Engineering, METU** _____

Examining Committee Members:

Prof. Dr. Pınar Karagöz
Computer Engineering, METU _____

Prof. Dr. Ferda Nur Alpaslan
Computer Engineering, METU _____

Assist. Prof. Dr. Orkunt Sabuncu
Computer Engineering, TEDU _____

Date:

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: Buğra Kaan Demirdöver

Signature :

ABSTRACT

LEARNING A PARTIALLY-OBSERVABLE CARD GAME HEARTS USING REINFORCEMENT LEARNING

Demirdöver, Buğra Kaan
M.S., Department of Computer Engineering
Supervisor: Prof. Dr. Ferda Nur Alpaslan

June 2020, 60 pages

Artificial intelligence and machine learning are widely popular in many sectors. One of them is the gaming industry. With many different scenarios, different types, games are perfect for machine learning and artificial intelligence. This study aims to develop learning agents to play the game of Hearts. Hearts is one of the most popular card games in the world. It is a trick based, imperfect information card game. In addition to having a huge state space, hearts offers many extra challenges due to the nature of the game. These challenges are divided into smaller parts where learning is easier and assigned to different learning agents. These agents use temporal difference learning to learn assigned parts.

Keywords: supervised learning, reinforcement learning, card games, artificial neural networks, temporal difference learning

ÖZ

YARI GÖZLEMLENEBİLİR MAÇA KIZI KART OYUNUNU TAKVİYELİ ÖĞRENME METODU İLE ÖĞRENME

Demirdöver, Buğra Kaan
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Ferda Nur Alpaslan

Haziran 2020 , 60 sayfa

Günümüzde yapay zeka ve makine öğrenmesi bir çok sektörde popülerler. Bu sektörlerden bir tanesi oyun sektörü. Oyun sektörü çok farklı senaryolar, farklı oyun türleri ile yapay zeka ve makine öğrenmesine önemli bir çalışma alanı sağlıyor. Bu çalışmanın amacı Maça Kızı oynamayı öğrenen yapay zeka birimleri geliştirmektir. Maça kızı tüm dünyada en çok popüler olan kart oyunlarından bir tanesidir. Maça kızı sıra tabanlı, yarı gözlemlenebilir bir kart oyunudur. Maça kızı büyük bir durum alanına sahip olmasının yanında bir çok ekstra zorluğu da beraberinde getirmektedir. Bu zorluklar oyunun, diğer kart oyunlarından farklı kuralları dolayısıyla doğmaktadır. Bu çalışmada zorlukları bir bütün olarak çözülmek yerine küçük parçalara ayrılıp kolaylaştırılmıştır. Her bir küçük parça farklı bir yapay zeka birimi tarafından zamansal fark öğrenimi yöntemiyle öğrenilmiştir.

Anahtar Kelimeler: danışmanlı öğrenme, takviyeli öğrenme, kart oyunları, yapay sinir ağları, zamansal fark öğrenimi

To my family

ACKNOWLEDGMENTS

I would like to thank my supervisor, Prof. Dr. Ferda Nur Alpaslan who always guided and supported me. Her extensive knowledge and intuitions greatly helped me through hardest times. It has been a pleasure being guided by her, and I hope I will have a chance to work with her again.

I also would like to thanks Özgür Alan. He also offered his guidance as much as my supervisor.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

ABBREVIATIONS

AI	Artificial Intelligence
ANN	Artificial Neural Network
ML	Machine Learning
MC	Monte Carlo
MCTS	Monte Carlo Tree Search
NPC	Non-Player Character
RL	Reinforcement Learning
TD	Temporal Difference

CHAPTER 1

INTRODUCTION

1.1 Motivation and Problem Definition

Both Machine Learning (ML) and Artificial Intelligence (AI) are fundamental parts of our lives. They are used in various applications from social media to self-driving cars. The games industry is no exception to that. The games industry is tightly closed with AI. There is a mutually beneficial relationship between them. While games provide AI a great work of field with many interesting scenarios, AI helps to create better and more enjoyable games. For example, the behavior of NPCs in games is critical for a successful game, as well as they are good problems to tackle.

Many games use AI methods to give a better experience to their players. Having a good smart agent is a crucial factor in the games' success. These smart agents can be created in many ways. One of the easiest ways to do it is to develop a rule-based system in which a set of rules are determined and agent moves, or plays, following the rules. Although this is a very common approach, it is not applicable for many games. For example, determining the set of rules for a non-player character in shooting games would be extremely difficult because there would be many continuously changing factors affecting the decision. Even though a ruled based method is applicable, humans can adept this never-changing decision mechanism quite easily which makes the game boring. So, relying only on classical solutions leaves us with many limitations.

ML methods are more suitable for many games and they are both more adaptive and more performant than rule-based solutions. They can be used for modeling the players, estimating a state, feature construction, and many more. Important topics in

games using ML is explained in [2] (Bowling et al., 2006) as follows:

- **Learning to play game:** Game playing agents are a popular piece of AI due to their diverse environments, from deterministic to non-deterministic, continuous to discrete. There are many works for both classical and modern games where AI helped learning the game.
- **Modelling the player:** To learn the behavior of the player, whether that player is an opponent or friend of the player, is one of the biggest challenges in AI.
- **Adaptivity:** To maximize the joy in games, agents need to adapt themselves to players. It is not fun to play against an agent always making the same moves and hence all the time winning or losing. One of the aims of AI is to build adaptive models.
- **Model interpretation:** Understanding the state of the game and inferring from it another influential aspect of AI. This inference can be used for hints, negotiation, advice, etc.
- **Performance:** Resource management is always a big concern for games independent of platforms they are published to. Although ML methods are generally resource-intensive in offline, they do not require much CPU or memory.

1.2 Proposed Methods and Models

Like every other game genre, ML techniques are also widely used in classical games like Hearts. Hearts is a hard game to learn. There are some extra features in Hearts that other card games do not have which can complicate the learning process. In this study, Reinforcement Learning (RL) is used to develop Hearts-playing agents. Due to the challenging nature of Hearts, more than one network is used to learn the game. While two networks are trained to learn pass card, two other networks are trained to play after the pass. Also, there is an additional rule in Hearts stating that if a player takes all penalty points, he/she gains additional reward points (For more information about game-play and passing in Hearts, refer to Chapter 3.2). This rule makes the learning process exponentially harder unless new networks are introduced to learn

it. So four additional new networks are used to learn this rule. One is to decide when to try taking all penalty points, one is to pass, one is to play and the last is the prevent other players to achieve additional reward points. With this method, agents successfully learned to play Hearts by self-playing, without any supervision.

Proposed contributions of this thesis are:

- Passing, shooting the moon, and preventing other players to shoot the moon are learned as well as normal playing.
- The main novelty this thesis proposes is that agents are tested against real people. Thus, it is aimed to provide a comparison to everyone working on Hearts.

1.3 The Outline of the Thesis

The outline of the thesis is as follows:

- Chapter 2 reviews the related work and literature.
- Chapter 3 provides background information about both ML methods and Hearts game.
- In Chapter 4, the work in this study is explained in detail.
- In Chapter 5, experimental results are presented.
- Finally, Chapter 6 concludes and gives final remarks about the work.

CHAPTER 2

LITERATURE REVIEW

Machine learning and games have always been in an important relationship. Games provide a perfect testing ground for researchers and scientists working on machine learning since games are, generally, easy to simulate and have multiple challenging properties in terms of creating an intelligent agent. Especially reinforcement learning is widely used in games.

There are many reinforcement learning studies in the video games category. One of them is [3] called The Arcade Learning Environment (ALE). ALE provides domain-independent agents using deep reinforcement learning techniques. It also provides an interface to many Atari 2006 games and provides benchmark results. Another interesting study is OpenAI Gym [4]. OpenAI is a toolkit for RL. The purpose of OpenAI is to create a library that contains various RL problems and relative solutions. It proposes solutions to many interesting problems from balancing objects[1] to atari games[2]. One trend in reinforcement learning nowadays is using raw image data instead of constructing features by hand. A good example of such work in video games is [5]. They used Q-Learning with convolutional networks to learn different kinds of Atari games.

Another common work field of machine learning is the card and board games. These games are easier to define but not necessarily easier to learn. The earliest works are the checkers learning agents developed by Arthur Samuel [6] and [7]. These works built the foundation of many other works in the field. He used a search-based method to teach the agent checker and managed to play it at the human level. One of the earliest and the most impactful work is TD-Gammon [8] developed by Gerald Tesauro.

[1] https://gym.openai.com/envs/#classic_control
[2] https://gym.openai.com/envs/#atari

TD-Gammon is, as the name implies is backgammon playing agent. Tesauro used non-linear version TD(λ) and multilayer neural networks. TD-Gammon has managed to play on par with the world backgammon champions. In [9, 10] Campbell et. al, introduced Deep Blue, a chess a playing agent. They have managed to beat world chess champion, Garry Kasparov, in 1997. They have used an extensive master game database and handcrafted for features for the opening and ending of the game. They have enhanced the alpha-beta pruning used in tree search algorithms. One other successful example which is more recent is the game of Go agent AlphaGo[11] developed by DeepMind[3]. In this work, the monte carlo tree search algorithm is combined with reinforcement learning of both value and policy networks. Such as Deep Blue, AlphaGo has managed to beat world champion 5 games out 5. Thanks to the policy and value network used for choosing and evaluating moves, compared to Deep Blue, AlphaGo evaluated fewer possibilities even tough the search space of Go is larger than chess. In AlphaGo Zero[12], the monte carlo tree search used in AlphaGo is completely removed and training is done using only reinforcement learning. AlphaGo is used as a teacher in this study. AlphaGo Zero is trained to predict moves of AlphaGo. This way, a stronger agent training is ensured. Among 100 games, AlphaGo Zero wins all of the games without losing against AlphaGo. After the success of AlphaGo Zero, in [13], a more generic version of AlphaGo Zero is developed which is called AlphaZero. AlphaZero generalizes the principles used in AlphaGo Zero and trains without any domain knowledge. It has trained to learn chess, shogi and Go. In all of these games, AlphaZero achieves superhuman strength and plays at least as good as the current champions. In another study, a card game Batak is examined [14]. Batak is a card game similar to Spades. In that work, monte carlo reinforcement learning with artificial neural networks is used.

This study aims to learn the game of Hearts. There are also some previous works done on Hearts. Although some achieved success, none of them successfully learns Hearts completely by including both passing and moon shot phases of the game. There are some studies based on search methods as [15] and [16]. In [15], monte carlo tree search-based algorithms used. Passing is not included in the study and the agent managed to learn Hearts to some extend but far from expert level. [16] is a

[3] https://deepmind.com/

study containing many multiplayer games. It uses decision rule algorithms (paranoid, max_n) combined with pruning. Both passing and moon shot are integrated into the learning process and tested against a commercial program at that time. Even though tests did not include passing, it has managed to beat the program. Another work, [17], considered [16] as the expert player and presented result accordingly. They have used Temporal Difference (TD) learning method and mainly focused on feature generation. They have explicit features for the moon shot, however, the passing phase is left out. In a series of games, [17] has managed to beat down [16], however, both programs are not as strong as a human. One other work used TD learning is [18], however, its results were not promising. In [19] environment modeled as a partially observable Markov decision process. They estimated the opponents' cards and predicted the action accordingly. They did not involve moon shot or passing, however, they have presented good results against rule based agents.

CHAPTER 3

BACKGROUND

3.1 Machine Learning

Artificial Intelligence (AI) and Machine Learning (ML) methods are used now more than ever. One of the main reasons for that is digitalization. With every record, every program, simply everything moving to digital platforms, there are so much unprocessed data piled up. Most of the time we do not have the right algorithm to get a meaning from that data. Predicting the likely customer for a product is one such example. There is not a certain definitive algorithm to find which customers buy which products. It can change depending on the year, on time, location, and many other factors. However, it can be quite trivial to do it with machine learning methods and lots of data. What ML methods essentially doing is extracting the patterns in data and uses that information to makes predictions or to gain knowledge. Normally finding such meanings in data would be impossible. Every field requires a different algorithm for that job, even though what they are doing is actually the same. [20]

There are many fields where machine learning is used commonly:

- Vision
- Data Mining
- Robotics
- Games
- Retail
- Finance

For sure it is impossible to limit the usage of ML with the above fields. Its applications are virtually limitless and still becoming more and more popular.

As it is stated in [20] "Machine learning is programming computers to optimize a performance criterion using example data or past experience". In ML, a model is used for learning which is optimizing the parameters of that model using data or past experiences. There are two important things to consider when dealing with ML applications. One is developing good algorithms to solve problems and second is dealing with the complexity of the algorithm. There are main algorithms/methods used in ML [20].

- **Learning Associations:** The aim learning association rules is to find conditional probabilities among the data in the form of $P(X|Y)$ where X is the action to take on condition Y. This approach is commonly used when trying to find the likelihood of taking an action such as basket analysis where the goal is finding the probability of customers' buying a certain product.

- **Classification:** Classification is reading patterns from the input and diving the data according to that patterns. It has a wide range of applications. For example, classification is used in face recognition where the input is an image and classes are people's faces. The aim is to match the input image with people's faces. This is a hard problem because the input is multidimensional and there are so many classes to be learned. Also, not all images may be taken from the same angle and they may not be at the same quality. Another example of classification is credit scoring problems[1]. In this example, the input is the information about the customer, classes are low risk and high risk. This one is a rather easy problem comparing face recognition because the input is smaller and the output has only two classes.

- **Regression:** Very much like classification, regression is also used for prediction given data. However, regression is used for continuous inputs such as predicting the prices of houses. The input here would be the attributes of houses, how big it is, location, room count eg., and the output would be the price of the

[1] Credit scoring or credit assignment is calculating the risk of customers not paying the total sum of his/her credit. It is usually used when banks decide to give credit to customers

house. The aim of the regression is to find a function that learns output Y as a function of input X. Same as classification, regression is a supervised learning method and requires labeled input data to learn.

- **Unsupervised Learning:** Different from supervised learning, in unsupervised learning data is not labeled. That means there is no supervision. Unsupervised learning aims to find irregularities or patterns in the input. The input is divided by those patterns. The main and most common method to do so is called clustering. Clustering can be used in image compression, document clustering.

- **Reinforcement Learning:** Reinforcement Learning differs from both supervised and unsupervised learning in a way that it does not have any prior data, labeled or not. It generally learns by self-training, necessary data is created on the fly. In reinforcement learning, immediate actions are not important until a goal is reached. Only then all intermediate states and actions have a meaning. To reach the goal, the learning agent is told whether the action taken is right or wrong. By rewarding the right actions, the agent learns to act to get more rewards. The most common example is a robot trying to find the exit of the maze. Once the robot reaches the exit, it is given rewards to each action leading to that path. How these rewards are distributed among states is an important question in reinforcement learning which will be discussed in later sections. Another important example is gaming. RL is used widely in games because games are easy to define but hard to master. Games play a big role in developing new machine learning algorithms.

In this work, reinforcement learning methods are used to learn the game of Hearts. The following sections cover the necessary methods and provide detailed information about concepts used in the project.

3.1.1 Artificial Neural Networks

Artificial neural networks (ANN) provide a supervised learning method. As computers gain more processing power it becomes more popular. It can be used to solve both continuous, discrete, and vector-valued functions given enough data. Theoretically, it

is possible to solve every problem with artificial neural networks given enough power and enough data.

Human biology, specifically brain, is a big inspiration in developing artificial neural networks. Human brains are tools that go beyond any machine learning agents. The brain has a very complex structure of densely connected neurons. Neurons are brain cells that can process and transmit information. The brain consists of roughly 10^{11} neurons which are connected round to 10^4 other neurons. The average neuron switching time is 10^{-3}. Similar to brains, artificial neural networks are also a structure of the interconnected set of units. These units take an input and produce an output to feed the other units until a final output is produced. Although computer switches are a lot faster than biological counterparts, humans are far better learners. This is because brains work in a highly parallel fashion. This parallelity is also simulated in artificial neural networks.

Artificial neural networks are capable of solving many problems [1].

- They can accept a different range of input attributes from text to images. The target function to be learned can be expressed by any attribute.
- They can also produce an output of many attributes. It can be continuous, discrete, or vector.
- They can tolerate errors in the input data.
- ANN generally takes a longer time to train than other supervised learning methods.
- Despite taking longer training times, evaluation of ANN is fast
- Humans do not need to understand intermediate steps or the weights of the training units.

3.1.1.1 Perceptron

ANNs have units assuming similar jobs as neurons in the brain. One of these units is called a perceptron. A sample perceptron is illustrated in Figure 3.1. Perceptrons take

an input of vector, calculate a linear weighted combination of the input, and compares the result with a threshold. If the value is greater than the threshold, perceptron outputs 1, else it outputs -1. More formally, a perceptron is

$$o(x_1, x_2, ..., x_n) = \begin{cases} 1 & \text{if } w_0 + w_2x_1 + w_2x_2 + ... + w_nx_n > 0 \\ -1 & \text{otherwise} \end{cases} \tag{31}$$

where x_i's are inputs and w_i's are the weights determining the contribution of x_i.

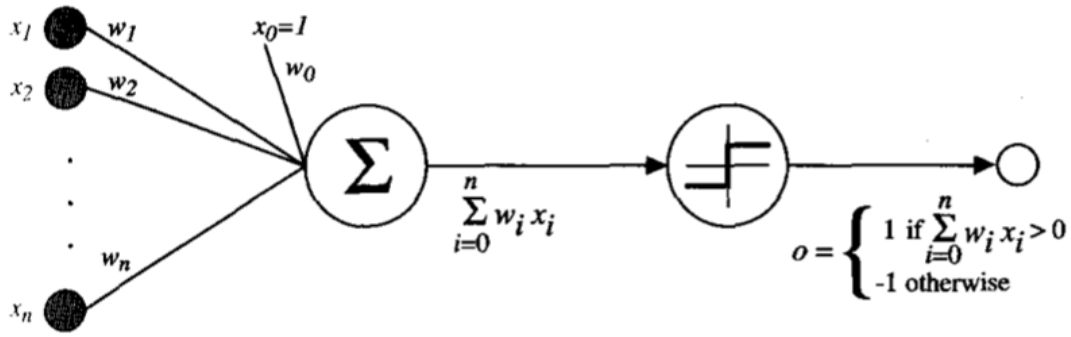

Figure 3.1: A perceptron [1]

Training the perceptron is finding appropriate weight values such that perceptron returns correct values for each example in training data. There are many algorithms for this learning task. Two of them are "Perceptron Training Rule" and "Gradient Descent".

3.1.1.2 Perceptron Training Rule

Perceptron training rule is simple. It starts with random weights. Then a perceptron rule applied to training examples whenever a misclassification happens. This iteration can take as many times as it is needed, there is not a limitation on the number of iterations. The rule to modify weights are as follows

$$w_i \leftarrow w_i + \Delta w_i \tag{32}$$

where

$$\Delta w_i = \eta(t - o)x_i \tag{33}$$

In the formula, t is the target or expected output while o is the real output value generated by the perceptron. η is the constant learning rate. The learning rate is generally set to some small value like 0.1. It sometimes decays as the iterations go on. The learning rate aims to adjust how much it is learned from previous iterations. A large learning rate generally makes learning faster but the model may not learn well. Whereas a small learning rate, on the other hand, increases the learning time but learning will converge to a more optimal state.

3.1.1.3 Gradient Descent

Another training algorithm, gradient descent, is used when training data is not linearly separable where perceptron training rule fails to converge.

Gradient descent is an important algorithm because it creates a base for the "Back-propagation Algorithm" which is the learning algorithm for multilayer networks. Gradient descent works by searching through hypothesis space and finding the best hypothesis that fits training data.

In search of the best hypothesis, a measure is needed for evaluation. That need is fulfilled with measuring the training error of the hypothesis. The following formula measures error for the training example

$$E(\vec{w}) \equiv \frac{1}{2}\sum_{d \in D}(t_d - o_d)^2 \tag{34}$$

where d is an instance of training examples D, t_d is target output and o_d is the actual output of linear unit of training example d with weight w. The aim is to choose the weight vector such that it minimizes the error E. Gradient descent begins the

[1] A dataset is linearly separable if it is possible to draw a line that can separate the data such that all classes (labels) would stay on the same side

procedure by assigning random initial weight vectors. The continues with modifying it in small steps. This continues until a global minimum is reached.

In gradient descent, weights are updated with the formula

$$w_i \leftarrow \Delta w_i + w_i \tag{35}$$

where

$$\Delta w_i = \eta \sum_{d \in D} (t_d - o_d) x_{id} \tag{36}$$

In the formula 36, η is the learning rate which specifies the step size. x_{id} is the input x_i for training example d. t_d is target output and o_d is actual output. The general algorithm is given in Algorithm 1.

Algorithm 1 Gradient-Descent$(training_examples, \eta)$ [1]

Each training example is a pair of the form of $< \vec{x}, t >$ where $\vec{x}$ is the vector of input values, and t is the target output value. η is the learning rate (e.g., .05)

Initialize each w_i to some small random value
Until the termination condition is met, Do
    Initialize each Δw_i to zero
    For each $< \vec{x}, t >$ in $training\_examples$, Do
        Input the instance $< \vec{x}$ to the unit and compute the output o
        For each linear unit weight w_i, Do
            $\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i$
    For each linear unit weight w_i, Do
        $w_i \leftarrow \Delta w_i + w_i$

The gradient descent algorithm starts with selecting an initial random weight vector. Then applies the linear unit to all training examples to compute o. Then computes Δw_i for each weight using Formula 36. Updates the w_i using 35 and keeps repeating these until the termination condition is met.

3.1.1.4 Multilayer Networks

There are some versions of ANNs. One version is adding new layers. These new layers are called hidden layers that are placed in between input and output layers. With only a single perceptron, only a limited number of functions can be expressed, however with a multilayer network it is possible to express a much larger variety of non-linear functions.

Deciding which training unit is important in multilayer networks. Using linear units or perceptrons would produce only linear functions by definition. In contrast, sigmoids can produce highly non-linear functions. It produces non-linear functions from its input, also its output is a differentiable function of its input. The sigmoid unit is very similar to the perceptron. The difference between them is their threshold units. A sample sigmoid is shown in Figure 3.2

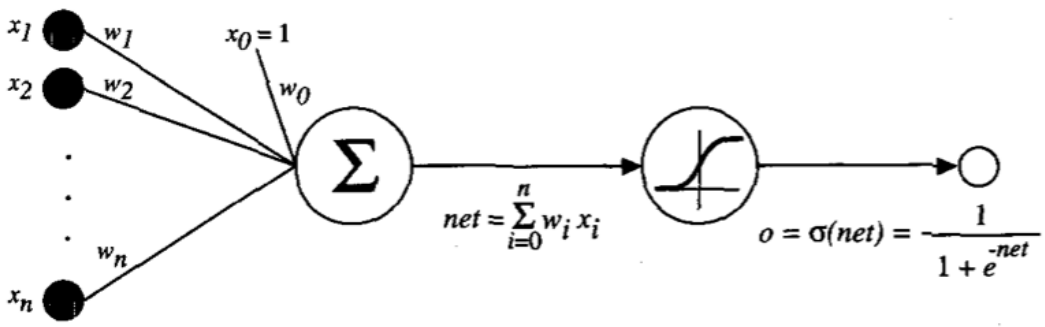

Figure 3.2: A sigmoid [1]

Similar to perceptron, sigmoid first computes a linear combination of its input and applies a threshold function. However, sigmoid threshold function outputs a continuous function. There is no discontinuity like perceptron. Sigmoid computes its output as follows

$$o = \sigma(\vec{w} \cdot \vec{x}) \tag{37}$$

where

$$\sigma(y) = \frac{1}{1 + e^{-y}} \tag{38}$$

The threshold function's output is in the range $[0, 1]$ and increases monotonically. This is why it is sometimes called squishing function.

One important aspect of multilayer networks is how the weights are calculated. For this, multilayer networks use Backpropagation Algorithm. It uses gradient descent to calculate the error of actual output values and target output values. This time, a different error function is used

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 \tag{39}$$

where $outputs$ is the outputs of network, t_{kd} and o_{kd} are the target and output values of the k^{th} output unit and training example d.

Backpropagation algorithm is given in Algorithm 2. There is a big difference distinguishing multilayer networks from single layers. In multilayer networks, the error surface can have multiple local minima. Gradient descent is guaranteed to converge toward a minima, however, that minima is not necessarily a global one. Fortunately, backpropagation algorithm has produced very good results in real life problems despite local minimums.

There are some variations of the backpropagation algorithm due to its popularity. One of them is obtained by changing the Formula 310 in the Algorithm 2 with following

$$\Delta w_{ji}(n) = \eta \delta_j x_{ji} + \alpha \Delta w_{ji}(n-1) \tag{311}$$

Where $\Delta w_{ji}(n)$ is weight update in n_{th} iteration and α is constant in range $[0, 1)$ called momentum. This momentum is added to the algorithm so that it can overcome local minimas and converge to the global minima.

Algorithm 2 Backpropagation$(training_examples, \eta, n_{in}, n_{out}, n_{hidden})$ [1]

Each training example is a pair of the form of $< \vec{x}, \vec{t} >$ where $\vec{x}$ is the vector of network input values, and $\vec{t}$ is the vector of target output values. η is the learning rate (e.g., .05)

η is the learning rate (e.g., .05). n_{in} is the number of network inputs, n_{hidden} is the number of units in the hidden layer, and n_{out} is the number of output units.

The input from unit i into unit j is denoted x_{ji}, and the weight from unit i into unit j is denoted w_{ji}.

Create a feed-forward network with n_{in} inputs, n_{hidden} hidden units, and n_{out} output units.

Initialize all network weights to small random numbers (e.g., between -.05 and .05).

Until the termination condition is met, Do

For each $< \vec{x}, \vec{t} >$ in $training\_examples$, Do

Propagate the input forward through the network:

1. Input the instance x to the network and compute the output o_u, of every unit u in the network.

Propagate the errors backward through the network:

2. For each network output unit k, calculate its error term δ_k

$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$

3. For each hidden unit h , calculate its error term δ_h

$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{kh}\delta_k$

4. Update each network weight w_{ji}

$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$

where

$$\Delta w_{ji} = \eta \delta_j x_{ji} \tag{310}$$

3.1.2 Reinforcement Learning

Unlike supervised or unsupervised learning methods, the aim in the reinforcement learning is to maximize a numerical reward. There is no supervisor to tell which ac-

tions to take, instead, after an action or a series of actions are taken, the learning agent is given a reward. The agent tries possible actions and sees which one gives the most reward by trial and error. In some cases, actions affect not only the immediate reward but also the ones after that. Those cases arise the concept of delayed reward. Trial error and delayed reward are the most important features of reinforcement learning. Since the agent acts alone in the environment, it must sense the state of the environment and must take actions accordingly. The actions agent takes will alter the state and the agent becomes closer to its goal state. [21]

One big challenge in reinforcement learning is deciding between exploration and exploitation. Exploration is visiting states that are not visited before whereas exploitation is visiting a previously visited state where it is known to yield a good result. The agent tries to get as much reward as possible. To do so, it must take actions yielding the most reward based on its experience. However, discovering those states requires exploration. This creates a dilemma. The agent must exploit to maximize reward but it also needs to explore to make better decisions in the future. Choosing exploration over exploitation or vice versa results in failing in the learning. The agent must try a combination of both to best utilize learning.

Before diving deeper, it is good to explain some terms. As it is stated in [21], a *policy* defines the agent's way of behaving at a given time and it is denoted with π. It is a mapping from states to actions to be taken in that state, $\pi : S \rightarrow A$. The policy defines actions to be taken as $s_t : a_t = \pi(s_t)$. A *reward signal* is a single number defining what is good and what is bad for the agent. The final goal of the agent is to maximize the total reward it gets. The *value function* is the cumulative reward given to agent over the future starting from state at time t, it is denoted with $V^{\pi}(s_t)$. An *episode* or *trial* is the sequence of actions from the first state to terminal state. Finally, the *model* is the environment the agent resides in.

In reinforcement learning, time is taken in discrete intervals and denoted as t. $s_t \in S$ is the state of time t and S is the all states that agent can be in. $a_t \in A(s_t)$ is the action taken in time t and $A(s_t)$ is the all action that can be taken in state s in time t. When the learning agent takes the action a_t in state s_t, a reward $r_{t+1} \in R$ is given where R is the set of all rewards. Then state is changed to s_{t+1}. This problem is modeled

using Markov Decision Model. In Markov Decision Models the future is independent of the past given the present. Meaning that, to decide the next action it is enough to know only the current state.

We know that value function is the cumulative expected rewards for the future. It is generally calculated with discounted future rewards as shown in Equation 312.

$$V^{\pi}(s_t) = E[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ...] = E\left[\sum_{i=1}^{\infty} \gamma^{i-1} r_{t+i}\right] \tag{312}$$

where γ is the discount factor and in the range of $[0, 1]$. When it is 0, only the immediate reward given in the state is considered. A s the value of γ increases, the effect of future rewards also increases.

For each policy π, there is a value function $V^{\pi}(s_t)$, and we want to find the optimal policy $\pi^{\star}$ such that

$$V^{\star}(s_t) = \max_{\pi} V^{\pi}(s_t) \forall s_t \tag{313}$$

In some cases, instead of values of states, value action pairs are used, $Q(s_t, a_t)$. Values of states define how good to be in that state. However, $Q(s_t, a_t)$ states that how good to take the action a_t when in state s_t. Then the equation 314 becomes

$$V^{\star}(s_t) = \max_{a_t} Q^{\star}(s_t, a_t) \tag{314}$$

where

$$Q^{\star}(s_t, a_t) = E[r_{t+1}] + \gamma \sum_{s_{t+1}} P(s_{t+1}|s_t, a_t) \max_{a_{t+1}} Q^{\star}(s_{t+1}, a_{t+1}) \tag{315}$$

In each possible next states $t + 1$, we move with probability $P(s_{t+1}|s_t, a_t)$, and continue with the optimal policy, the possible rewards in that state is $\max_{a_{t+1}} Q^{\star}(s_{t+1}, a_{t+1})$. These are also discounted because it is one time step later. Finally after adding the immediate expected reward, total expected cumulative reward for action a_t is reached.

By the equations 314 and 315, optimal policy can be defined as

$$\pi^\star(s_t) = \text{ Choose } a_t^\star \text{ where } Q^\star(s_t, a_t^\star) = \max_{a_t} Q^\star(s_t, a_t) \tag{316}$$

3.1.2.1 Value Iteration

Optimal policy can be found using optimal value function. Value iteration is the method to do so. This iteration is obtained by turning Equation 315 into an update rule. Using that equation is shown to converge to the correct $V^\star$ value. The iteration stops, $V(s)$ converges, once the value function changes by only a small amount. The algorithm is given in the Algorithm 3

Algorithm 3 Value Iteration [20]

Initialize $V(s)$ to arbitrary values
Repeat
  For all $s \in S$
    For all $a \in A$
      $Q(s,a) \leftarrow E[r|s,a] + \gamma \sum_{s \in S} P(s'|s,a)V(s')$
    $V(s) \leftarrow \max_a Q(s,a)$
Until $V(s)$ converge

3.1.2.2 Policy Iteration

Instead of trying to find optimal policy through value function, in Policy Iteration, this is done directly by calculating the policy. Value function is iteratively calculated by solving linear equations. Then, policy is checked if it can be improved with new values. It can be improved, it is updated, otherwise, it is guaranteed to be the optimal policy. The algorithm is given in the Algorithm 4

Algorithm 4 Policy Iteration [20]

Initialize a policy π' arbitrarily

Repeat

$\pi' \leftarrow \pi$

Compute the values using π by solving the linear equations

$$V^{\pi}(s) = E[r|s, \pi(s)] + \gamma \sum_{s \in S} P(s'|s, \pi(s)) V^{\pi}(s')$$

Improve the policy at each state

$$\pi'(s) \leftarrow argmax_a(E[r|s, a] + \gamma \sum_{s \in S} P(s'|s, a) V(s'))$$

Until $\pi' = \pi$

3.1.2.3 Temporal Difference Learning

Both value and policy iterations are model based methods where there is no need for exploration. These can just be solved for optimal value or function. Temporal difference learning, however, is not a model based method. In many cases, we do not have enough information about the environment to build a model, so model-free learning is important. This model-free learning can be deterministic and non-deterministic.

In deterministic case, the state after every action is known. For every state-action pair, there is a certain reward and certain next state. Thus, Equation 315 can be reduced to

$$Q(s_t, a_t) = r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) \tag{317}$$

Then, value of the previous action is updated as

$$\hat{Q}(s_t, a_t) \leftarrow r_{t+1} + \gamma \max_{a_{t+1}} \hat{Q}(s_{t+1}, a_{t+1}) \tag{318}$$

Considering there is immediate rewards, at first, all $\hat{Q}(s_t, a_t) = r_{t+1}$ values are 0. They are updated in time. In Equation 320, Q value is updated with next state's reward and value. Therefore, all values are 0 until the terminal state where a reward is given. When in a terminal state, the reward r is given and previous Q value is updated as γr. Q value before that is discounted by γ and immediate reward is zero, so it is updated as $\gamma^2 r$ and so on.

Algorithm 5 Q Learning [20]

Initialize all $Q(s, a)$ arbitrarily

For all episodes

Initialize s

Repeat

Choose a using policy derived from Q, e.g., $\epsilon-$greedy

Take action a, observe r and s'

Update $Q(s, a)$ as

$Q(s, a) \leftarrow Q(s, a) + \eta(r + \gamma \max_{a'} Q(s', a') - Q(s, a))$

$s \leftarrow s'$

Until s is terminal state

In the non-deterministic case, we have a probability distribution for rewards and the next states. With these distributions, Equation 319 becomes

$$Q(s_t, a_t) = E[r_{t+1}] + \gamma \sum_{s_{t+1}} P(s_{t+1}|s_t, a_t) \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) \tag{319}$$

Since rewards and states are non-deterministic a running average is kept.

$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \eta(r_{t+1} + \gamma \max_{a_{t+1}} \hat{Q}(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \tag{320}$$

The algorithm for this update rule is given in Algorithm 5 and it is called the Q-Learning algorithm. Note that, this is an off-policy method as the value of the best next action is used without using the policy. There is an on-policy version of Q-Learning which is called Sarsa algorithm. Its pseudocode is given in Algorithm 6.

In Sarsa, instead of looking for all possible next actions a' and choosing the best, Sarsa uses the policy derived from Q values to choose the next action a' and uses its Q value to calculate the temporal difference.

Algorithm 6 Sarsa [20]

Initialize all $Q(s, a)$ arbitrarily

For all episodes

Initialize s

Choose a using policy derived from Q, e.g., $\epsilon-$greedy

Repeat

Take action a, observe r and s'

Choose a' using policy derived from Q, e.g., $\epsilon-$greedy

Update $Q(s, a)$ as

$Q(s, a) \leftarrow Q(s, a) + \eta(r + \gamma \max_{a'} Q(s', a') - Q(s, a))$

$s \leftarrow s', a \leftarrow a'$

Until s is terminal state

3.2 Game Environment

Hearts is one of the most popular trick-taking card games in the world. It was first seen in America in the 1880s and has many versions. The most popular version is inspected in this study.

Hearts is an imperfect information[2] trick taking card game which is played with n ($n \geq 2$) player (It generally is played with 4 people).

Although there are some simplified versions of the game Hearts[4], this study focuses on the most popular and known version. The game is used with a standard 52-card deck with no Jokers. There are four suits, namely spades, diamonds, clubs, and hearts. All suits have equal strength, but the strength of cards increases with the following order: 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K and A.

At the start of the game, each player is dealt 13 cards. The game is played in clockwise order. Each player selects a card to play when all players played a card from their hands, it is called a trick. A round consists of 13 tricks. The game ends when a player

[2] Imperfect information games are games where players have limited information about the game. Unlike perfect information games such as checkers or chess where players can see the entire board and pieces, in imperfect information games like Hearts, the hands of other players are hidden.

[3] `https://play.google.com/store/apps/dev?id=5890018970979485285`

[4] `https://en.wikipedia.org/wiki/Hearts_(card_game)#Variants_2`

Figure 3.3: A screenshot of the game Hearts from SNG Studios[3]. AI agents in this project are added in this game and tested against real human players

reaches a pre-determined point, which usually is 50 or 100, after a certain amount of rounds. Only hearts and Queen of Spades have score values, each heart card is one point and Queen of Spades is 13 points. The rules of the game are explained below.

- Each round consists of two phases, the passing and the playing. Each round starts with the passing phase and continues with the playing phase.
- At the beginning of each round, players pass 3 of their cards face down to another player. In the first round, cards are passed to the player on the left; on the second round they are passed to the player on the right; on the third round, they are passed to the player across to the table. The fourth round is no passing round. After that, the rotation repeats itself.
- Once cards are passed, the playing phase starts. The player holding two of Clubs begins the game. First played in each trick is called leading card.
- If there is no card played in that round, namely the player is leading, the player can lead with any card in his/her hand except with a heart. To lead hearts, a

card with the suit of hearts must be played in previous rounds (This is called "Breaking Hearts").

- If there are cards played, namely the player is following, the player must play a card that is the same suit with the leading card.
- When the player is following and has no card with the suit of the leading card. He/she may play any card in his/her hand. If that happens and a hearts is played the first time in the game, hearts are broken and can be led.
- After the trick is played, the player who has played the strongest card with the same suit as the leading card wins the trick. The winner takes all the cards in the trick.
- There are a total of 26 points that can be taken in a round. Normally taking points is not something desirable; however, if a player takes all 26 the points in a single round, instead of getting penalized the player is rewarded with 0 points whereas other players are penalized with 26 points each. This is called "Shooting the Moon".
- At the end of the game player with the lowest score wins.

CHAPTER 4

PROPOSED WORK

In this work, reinforcement learning is used to learn playing Hearts. Specifically, Temporal Difference (TD) learning is applied to find an optimal value for each action (cards that can be played). Since the state space of Hearts is too big, instead of classical reinforcement methods, neural network is used.

4.1 State-space Representation

As mentioned in Section 3.2, the game of Hearts is a complicated game consisting of more than one phase and some tricky rules for an AI agent to learn. To overcome this complex structure, the game is divided into smaller pieces and each piece is learned by a different AI agent. Factors contributing to this complexity are passing and moon shot. Hearts consists of two phases, passing and playing. In the passing phase, players select three of their cards to give opponents. Since this phase is different from classic gameplay it adds up to the complexity. Moon shot is a special case that can happen in the playing phase. Normally, players avoid taking points. The fewer points they have the better. However, taking all 26 points in a round results in the moon shot. In this case, the player who shoots the moon takes zero points while others take 26 points. So, in this study, there are seven different networks are used; two of them are for normal play, two of them are moon shot play and one to decide which set of networks to be used in given cards.

The simplified flow of the overall process is given in Figure 4.1. In the figure, input and the state of the environment are different for each process while the output is the same and it is an action set that can be played by the agent. "Moon Shot Decision"

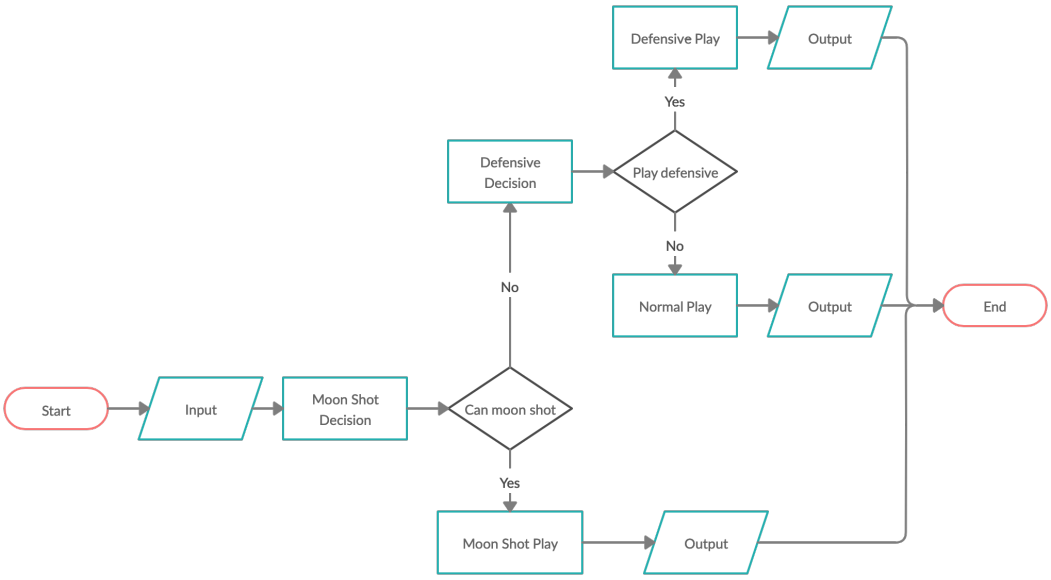

Figure 4.1: Simplified flowchart of AI workflow

process is a supervised network deciding if the player should go for shooting the moon or just stick with classic game after the cards are dealt. "Defensive Decision" process is also a supervised network, however, it checks if other players are going for the moon shot and if it is needed to play defensively to prevent unwanted moon shots. If the decision favors defense then "Defensive Play" process is run. This process is an RL network for the playing phase of the game. The other processes, namely "Moon Shot Play" and "Normal Play", consist of two RL networks. One for passing phase and one for playing phase. These networks are multilayered feed-forward fully connected neural networks that are trained for playing the game with the backpropagation algorithm. Detailed explanations about these networks are given in Chapter 3.1. Once the supervised network decides which process should continue with the game, first, the pass network decides the cards to be passed and then the play network continues with playing tricks. "Normal Play" learns how to avoid taking point cards and "Moon Shot Play" learns to take all of the point cards. Although the general structure of the used networks for these two processes are similar, their goals thus rewards fed to networks are completely different. Due to that difference, these processes are assigned to different networks.

Since some networks have play and pass parts, to explain any network, the same terms

will be used, this will create conflicts. To avoid that the following naming convention will be used throıhout the study. The supervised network in "Moon Shot Decision" process will be called "moon shot decision network" and "Defensive Decision" will be called "defensive decision network". The network in "Defensive Play" will be referred as "defensive play network". Networks in "Normal Play" process will be called "play network" and "pass network". Finally, networks in "Moon Shot Play" process will be called "moon shot play network" and "moon shot pass network".

Detailed algorithms and how the networks are trained are explained in later sections.

4.2 Feature Construction

Developing a successful RL agent requires a strong field knowledge because using that knowledge, one must construct an accurate and useful set of features unless they are working with raw image data. In this study, the input is text-based, so construction of features has a crucial importance. Selecting the right set of information and utilizing it efficiently can make a big difference in training. There are some works solely dedicated to this topic and some methods in feature construction. In this study, since the flow is made up of multiple processes including both ANN and RL networks, the data and feature sets that these networks use are different.

Pass and play networks, though they are both RL networks, also differ. Pass network and play network take different inputs, thus they have different features. Both pass network and moon shot pass networks take the same set of input, have the same states, and outputs the same information. The only difference is how these states are rewarded. Just like the pass networks, play networks also have the same structure for both processes. In addition to that, pass networks need much smaller state space to learn how to pass. They must decide after cards are dealt, hence the initial cards given to playing agent is all they need to consider. On the other hand, play networks need more information on the game like which player played which card on the trick, which cards are played out, and so on. However, finding the specific information and organizing it in a way that networks can understand is not an easy job. Hearts is an imperfect information game. This means the data accessible to the learning agent

is limited. The agent has no information about the cards the players have. It can only know cards in its hands, cards played so far in the current trick, and cards that are consumed. Consumed cards are the ones played in previous tricks and that will not be included in the game until the round is over. To best utilize this information, features are constructed based on card positions in the game. This state representation is also used in [14] for learning the card game Batak which is very similar to Spades. Cards can be in total 5 different positions as far as the playing agent knows: in agent's hand, in any other player's hand, played in the current trick by the agent, played in the current trick by other player or consumed. There are 5 different possibilities for a card, thereby features are represented as vectors of five which are encoded as one-hot vectors as follows:

in playing agent's hand	$\rightarrow [1, 0, 0, 0, 0]$
in any other player's hand	$\rightarrow [0, 1, 0, 0, 0]$
played in the current trick by the agent	$\rightarrow [0, 0, 1, 0, 0]$
played in the current trick by other player	$\rightarrow [0, 0, 0, 1, 0]$
consumed	$\rightarrow [0, 0, 0, 0, 1]$

There are 52 cards in a deck, each is encoded as vectors of 5, thus the state is represented with 260 bits. The first 65 bits(13 cards) of this state are for suit clubs, next 65 is for diamonds, next is for hearts and last 65 is for spades. With this, play networks have all the information it needs to learn the game.

Constructing states of pass networks is a little bit easier. As mentioned before, they only need to know if the player has the card or not. For them, cards can only be in two places. So, if the agent has the card it is encoded as $[1]$, else it is encoded as $[0]$. Since this time cards are represented with 1 bit, the total length is 52 bits. Nevertheless, this alone is not enough to learn to pass. The network still needs to know which cards to pass. Similarly, this information is fed to the networks using one-hot vectors. On top of the first 52 bits, another 52 bits added to state representation where cards that are passes encoded as $[1]$ and cards that kept in hand as $[0]$. This way both pass

and play networks have all the necessary information in their states. The final state representation of pass networks is as follows

in the hand	$\rightarrow [1]$	passed	$\rightarrow [1]$
else	$\rightarrow [0]$	kept	$\rightarrow [0]$

Unlike RL networks, the moon shot decision network which decides if the agent should play for taking minimum points or go for moon shot is a supervised network. Because of that reason, the data for that network is generated beforehand. The network reads and gets trained on it. However, both ANN and RL networks use the same representation of data. That means the data ANN reads consists of one-hot encoded vectors based on cards' positions. To find out what exactly the network needs to decide about moon shot, it is beneficial to examine expert players' behavior. Expert players can decide if they are going to try to moon shot the moment they see their initial hand. In fact, the most moon shot decisions are made when hands are dealt. If a player decides to shot moon later in the game, it is likely that attempt will not succeed. Moon shot decision network must decide after the hands are dealt. Hence the data must contain hands (total of 13 cards) and labeled with if given the hand player shot the moon. For this purpose, several games are played between previously trained networks and the 3 datasets are prepared step by step. The reason why there are 3 different datasets and how they are prepared are explained in the next chapter along with RL networks' implementation details.

The defensive decision network is also a supervised network and it follows exactly the same procedure as the other networks while constructing features. Its feature set is the same as play networks features. Detailed explanation about why it is the case is given in Chapter 4.3.6.

4.3 Algorithms and Methods

The main difference among networks used in the study is how they use the features that are explained in Chapter 4.2. The focus of this chapter will be on this difference and main flow of the learning process. Since performance and the learning time are secondary concerns, these two aspects will not be examined.

Before the learning agent, there are some components implemented. These are a game environment and two Hearts playing agents aside from the learning agents. One of the playing agents is a rule based agent implemented based on the tips from the website[1]. This agent will be referred to as rule based agent. Among 2 non-learner agents, this is the worst player. The other one is a combination of MCTS and rule based algorithms. It is good to note that this agent is based on the study in [15] and improved with hard-coded rules. It is implemented for the hearts game published by SNG Studios[2] and will be referred to as monte carlo agent.

There are three important implementation details in reinforcement learning in this work; one is whether to give a reward immediately after an action is taken, one other is to choose giving how much reward to the states leading to victory and the last one is the exploration strategy. Thompson Sampling[3] is used as an exploration strategy in this study. Other than the current playing agent, some other agents also learned to play the game created by tweaking these options. Some of these agents will be explained in Chapter 5 by comparing them to the current playing agent and explaining why they are not as successful.

Apart from implementation details, one other important factor is that networks do not try to find out which cards are allowed to play by the rules. They are given the allowed cards and make their choices. Learning the rules of Hearts is not the aim here.

[1] http://mark.random-article.com/hearts/
[2] https://www.sngict.com/
[3] https://en.wikipedia.org/wiki/Thompson_sampling

4.3.1 The Play Agent

The first trained agent was the playing agent. In this training period, passing and moon shot are excluded. The game consists of only the play phase so that rewards given for the play agent would only measure the quality of the playing phase of the game. For this network, the mean squared error function is used for error (loss) function and sigmoid function is used as the activation function. There are a total of 13 states before the round is concluded with a victory or loss. As stated in Chapter 4.2, states consist of one hot coded vectors based on cards' possible locations. Each action(playing a card) causes a change in the state. Like in a classical reinforcement learning problem, states that lead to the desired outcome has rewarded. However, instead of giving discounted rewards to the states as we get closer to the initial states, all states are rewarded by the same amount. The reason is that in a card game all actions are almost equally important. Although it is not feasible to say all actions are equally important, indeed, the importance is not relevant to the playing order. Measuring the quality of an action based on its impact on the score rather than when it is made is a better approach. For example, an early move of getting rid of the high spades (to avoid taking queen of spades) or sloughing[4] a point card is so much better than taking points in a late trick. Because of the explained reasons, the same rewards are given to every state, not just in playing agents but also in moon shot play agent.

Giving the same reward to all states makes training more reliable. Nevertheless, this approach misses something important. If all the states are given the same reward, the network cannot distinguish a good move from a bad one in the same trick because they are both given the same reward. Therefore, good moves are supported with a positive immediate reward while bad moves are punished. One thing to point out is that immediate reward is given only for play agent and moon shot play agent. Pass agents do not need such tweaking since their state space is smaller than play networks. In play network, immediate rewards are given in such a fashion that it is in the range $[0, 0.5]$ and they are given only when the learning agent takes point in the trick. Immediate rewards are given in a range of $[0, 0.5]$ to down their effect in the learning process. They are not as important as the terminal reward given at the end

[4] Sloughing is playing a card with a different suit than the leading card. Sloughing is allowed only if the player has no card with the same suit as leading card

of the trick. Immediate rewards are aimed to be little corrections and not shadow the terminal rewards. To determine giving how much immediate reward is the optimal, a couple of tests are applied. In a round, a player can take at most 16 points (13 for the queen of spades and 3 for hearts). However, taking 16 points is unlikely. In most cases, the queen of spades is sloughed alone which causes the player to take 13 points. Besides, out of 13 tricks in a round, only in 1 trick queen of spades will be played and in the remaining rounds, the maximum point to be taken would be 4 when the player takes 4 hearts. In the light of these information, 4 different immediate reward formulas are tested while training play network. These formulas are represented in Table 4.1.

Table 4.1: Immediate reward formulas tested for the play network

Formula_1	$r(s) = max(0, \frac{4-points_s}{8})$
Formula_2	$r(s) = max(0, \frac{8-points_s}{16})$
Formula_3	$r(s) = max(0, \frac{13-points_s}{26})$
Formula_4	$r(s) = max(0, \frac{16-points_s}{32})$

In the formulas, $points_s$ represents the point taken the trick leading to that state. As it can be seen in the table, all formulas are basically doing the same job. The difference is how much points they tolerate to take points in a trick. Formula_1 gives 0 as a reward when the agent takes 4 points or more while Formula_2 gives o reward upon taking 8 or more points, Formula_3 gives 0 reward upon taking 13 or more points, Formula_4 gives 0 reward when 16 or more points taken. In most cases, the maximum points taken in a trick is 4. This is why among all 4 formulas, Formula_1 becomes the most successful version and it is used in training play agent.

Another aspect of training play agent is giving the terminal reward. Terminal rewards are given when all 13 tricks are played out. As it is explained before, the terminal reward of all states is the same. They are in range $[0, 1]$. As is the case with immediate reward, there have been made some small tests to determine the best terminal reward. These are represented in Table 4.2. In the table, all formulas give full reward when agent tales 0 points. They differ on the upper limit of the points taken. Formula_1

gives 0 reward when the agent takes 10 or more points, Formula_2 gives 0 reward when the agent takes 13 or more points, Formula_3 gives 0 reward when the agent takes 26 or more points.

Table 4.2: Terminal reward formulas tested for the play network

Formula_1	$r(s) = max(0, \frac{10 - points_s}{10})$
Formula_2	$r(s) = max(0, \frac{13 - points_s}{13})$
Formula_3	$r(s) = max(0, \frac{26 - points_s}{26})$

The straightforward approach to give a terminal reward is to give 0 when the learning player takes all the points and give 1 when the player takes no point. However, as it is in immediate rewards, this approach fails because players generally do not take much more than 13 points. This is why when Formula_3 is used agents are hardly trained with more than 0.5 rewards. To overcome this problem, Formula_1 and Formula_2 is also tested. Among these three, Formula_1 is surpassed the others. When the play agent is trained using it, the win ratio of the agent rose significantly.

The play network is trained with the above rewards and states and the exploration strategy explained before is applied.

4.3.2 The Pass Agent

After successfully training the play agent, the pass agent is trained. As in play network, sigmoid is used as the activation function in the pass network. No immediate reward is used in the pass network since there is no set of prior moves that can affect passing. Passing is the first thing to do in every round.

At the beginning of each round, every player passes three cards to the player to their left, right and across alternately. Determining how good is that passing is impossible by just looking at the cards passed. In this study, passing is evaluated by letting round ends after passing and giving rewards according to the points taken at that round. Therefore, to train the pass network, having a strong play agent is a must. So, the

pass agent is trained using the play agent.

Similar to play agent, pass agent also uses the same formula to terminal rewards. Since Formula_1 in Table 4.2 is seemed to work well, it is used in pass training as a terminal reward.

Unlike choosing which card to play in a trick, passing does not have any limitations. Players have to play by the rules when selecting their cards to play, they have to follow suit whenever possible, they cannot play a point card in the first trick, etc. However, in passing, players are free to choose any 3 cards from their hands of 13 cards. Thus, when choosing the cards to pass, the pass network tries each combination of 3 cards out of 13 and picks the cards according to the exploration strategy.

4.3.3 Fine Tuning Normal Play Process

Playing the game of Hearts successfully means playing both passing and playing phases successfully. It also means that passing and playing must be played serving the same strategy. Both passing and playing must be made with the other in mind. They are two phases that complete each other in the game. It is very important to have a harmony between two phases is a key element in reaching success.

So far in the training, the pass agent is trained using the previously trained play network. That lets the pass network not just to learn how to pass but to learn how to play adaptive to the play network. This allowed pass network to learn to best utilize play network's moves. However, the play network does not aware of any passing and play as it needs. To train networks to best utilize each other's moves, one more fine-tune training step is added. In this last step pass and play networks are trained together. For this training step, all passing is added back to the game. As in the training of pass agent, the play agent plays its games with the pass agent. For a certain number games play agent is trained using the last trained pass agent. Then the pass agent is trained with previously trained play agent.

This training step is seemed to increase the success of overall playing. Experimental results are presented in Chapter 5.

4.3.4 The Moon Shot Decision Agent

The next step in the training is to teach the agent how to shoot the moon. However, to do so, a decision network must be trained first. This decision network is important for two reasons. One, it tells the agent when to shoot the moon. The moon shot is really hard and dangerous. If a moon shot attempt fails, the player not only fails to shoot the moon but also takes a bunch of unnecessary points which may cause the player to lose the game. Hence, the player needs a good hand to even try shooting the moon. If it is tried every trick, player will lose the game with so many failed attempts to shoot the moon. Two, it helps training the moon shot agents. In a normal game, only 5 percent of total tricks are ended up with moon shot. That means if moon shot pass and play agents are trained without any filtering, the data they train on would be mostly non moon shot plays (about 95 percent). Moon shot plays being that rare results in unbalanced data. This issue is solved by discarding some non moon shot plays using decision agent. That way networks are trained with more even distribution of moon shot hands.

Unlike the trained agent, the moon shot decision agent is supervised. That means labeled data is needed to agent to learn from. As it is stated before, this data consists of players' initial hands with labels indicating if the player has managed to shot the moon with a given hand. Hands are represented with a binary array of length 52. If the player has the card, the corresponding index in the array is 1 else 0. Label is 1 if the player shot the moon, else it is 0.

The first version of the training data is prepared using play network. Play network does not know how to shoot the moon. Thus, preparing the training for a moon decision network with it does not yield good results. To achieve success, an agent that knows how to moon shot must be used to create training data. However, at this point in the training, no such agent is present. This is why the first version of the moon shot decision agent is trained on such data. So, the moon shot networks' training is started with the first version but proceeded with the second version of the decision agent. The second version has the same structure as the first one. The difference is that the second is prepared with moon shot play network. This ensures that the network trained on the second version would find hands that moon shot play agent

can play successfully. The rest of the moon shot play training is done with the second version. With the same principle, third and fourth versions of the dataset are created to train moon shot pass agent. These versions are prepared by games of pass agent and moon shot play agent. The final version is used in the final product ("Moon Shot Decision" process in Figure 4.1) to decide on which hands are suitable for the moon shot. In this version, the aim is to actually select hands with high probability of moon shot rather than helping training. Since the data is prepared using moon shot play and moon shot pass agents, this version finds hands good for specifically the moon shot agents. Like other versions, the final version also has the same structure.

Both versions of moon shot decision agents contain 1 million tricks. During these games, all four players' initial hands and whether they shot the moon or not are considered. Also, about half of the tricks one player has successfully shot the moon which provides that data is evenly distributed.

4.3.5 Moon Shot Play and Pass Agents

After creating a supervised network to decide when to shoot the moon, training an agent to do so is the next step. Both the moon shot play and the moon shot pass agents are trained with the same logic as their normal play counterparts and they have the same structure. They both use the sigmoid function as the activation function. Hence, rewards for each state are in the range $[0, 1]$. Just as in normal play agents, the first moon shot play network is trained. Then using the trained moon shot play networks, moon shot pass network is trained. Also, the same fine tuning process is applied where both play and pass agents are trained together one final time. Moreover, the same state representations are used here.

The difference between the moon shot agents and the normal play agents is the way of how their states are rewarded. One tempting way to reward states in moon shot play agent is thinking moon shot play agent as the opposite of normal play agent and just reversing the rewards. However, that would fail. Because when the reward is given as in Equation 41, not being able to shoot the moon is also rewarded. That way,

taking points is rewarded, not shooting the moon.

$$r(s) = \frac{points_s}{26} \tag{41}$$

So, instead of Equation 41, a more extreme approach is followed for moon shot play. When the player does not try to shoot the moon, he/she can take a few points yet it can be considered a good play. However, the moon shot does not tolerate mistakes. The player has to play perfectly and take all the points to shoot the moon. Thus, the maximum reward is given if the player successfully shoot the moon and 0 otherwise as in Equation 42

$$r(s) = \begin{cases} 1 & \text{if } points_s = 26 \\ 0 & \text{otherwise} \end{cases} \tag{42}$$

This extreme method substituted by a milder method in immediate reward. Immediate rewards are a result of only one state in the whole game and it is impossible to understand if the player shoots the moon from a state of the one trick. So being extreme here causes giving harsh and wrong rewards. Besides, immediate rewards must not shadow terminal rewards. This is why, they are limited in the range $[0, 0.5]$. The immediate reward formula is given in Equation 43. If the player takes 10 or more points in a trick, the maximum immediate reward is given. If the player takes less than 10 points, less reward is given accordingly. Although it is possible to take up to 16 point, in most tricks, a player can take at most 4 points. Since moon shot is not about taking more points but about taking all of the points, the player is not given a full reward when taking 4 points.

$$r(s) = min(\frac{points_s}{10}, 0.5) \tag{43}$$

The immediate reward is given only when the player takes points in a trick. Giving 0 reward when no point is taken would badly impact training. Throughout the training, there are many cases where the player takes no point in a trick and this does not always mean that player made a bad move. Due to the cards in hand or opponents' hands,

it is sometimes impossible to take points. To minimize these wrong evaluations, the immediate reward is given only if the player takes points in a trick.

The extreme approach is also applied in the moon shot pass agent. It is trained with a previously trained moon shot play agent. The same formula used in moon shot play agent, Equation 42, is also used in moon shot pass agent.

4.3.6 Defensive Agents

In the last step of the training progress, the agent is learned to prevent other players to shoot the moon. For that cause, first, a supervised agent is trained to determined when to play defensively and when to play as usual. Unlike moon shot decision agent which runs once at the beginning of the round, defensive decision agent runs in every trick before playing. Defensive play decision is generally made toward the end of the game. As the game continues and the learning player begins to infer opponent players' cards or play types, making a defensive decision makes more sense. This is the reason why that defensive decision agent is run in every trick.

Since the defensive decision network is supervised, the data is prepared beforehand. This data is the same as the general representation used in RL networks where it is based on card position. The data is labeled with moon shot occurrence at the end of the game. If there is a moon shot in a round, respected data is labeled with 1 otherwise 0. In an ordinary hearts game, generally, no more than one player tries to shoot the moon in a round. Considering that, while generating data, one moon shot player and three normal players are played against each other. The state of the moon shot player is saved and labeled. Along with deciding to play defensively, the defensive decision network is also used in training to even out the moon shot hands as in the moon shot decision network.

With the defensive decision agent is trained, defensive play agent is next. It has also the same structure as the other agents. It also uses the same state representation. The only thing different is the rewards given. All the other agents are rewarded with the points they take, however, defensive play agent is rewarded with other player's points so that more than one player takes points and moon shot occurs. The defensive agent

aims to let nobody shoot the moon. Hence it is rewarded generously if this is the case. The reward is calculated as in Equation 44.

$$r(s) = \begin{cases} 1 & \text{more than player takes points and the player takes } 0 \text{ points} \\ 0.1 & \text{more than player takes points and the player takes more than } 0 \text{ points} \\ 0 & \text{otherwise} \end{cases} \tag{44}$$

Aim here to prevent moon shot by making other players take points. This is the best case and rewarded with 1. The second best case scenario is nobody shoots the moon but the player takes points. In this case, the player manages to prevent the moon shot in the cost of taking points. This is rewarded with a small number. The last case is somebody shooting the moon. This is the least desirable case and no reward is given in this situation.

Another issue in the defensive play agent is the immediate reward. In other agents, the immediate reward is given when the player takes a point in the trick. In the defensive play network, the immediate reward is given only at the trick where the second player takes the point. This ensures that the action that clears away the possibility of a moon shot is given an extra push. The immediate reward is given in Equation 45.

$$r(s) = \begin{cases} 1 & \text{second player takes point and the player's points are } 0 \\ 0.1 & \text{second player takes point and the player's points are more than } 0 \end{cases} \tag{45}$$

In the defensive play process, no passing is learned because starting the game defensively is often disadvantageous. Understanding if any player is going to shoot the moon only looking at your hand is not something feasible. Passing is a very important part of the game and playing to prevent moon shot in this phase may cause more damage than benefit.

CHAPTER 5

EXPERIMENTS

Learning agents are tested against 3 different agents. One is rule based agent, one is monte carlo agent, and the last is real human players from SNG Studios Hearts game. The rule based agent is used as a baseline for experiments. Moreover, most of the human players tested against are at least intermediate level players. Of course, knowing that exactly is impossible, yet common player behavior is playing a game you are good at or you are winning. However, it is certain to say that strong players play the game more than weak players, thus contributing the results more.

In this section, the experiments are made to measure both individual and collective success of trained agents against the other agents. While measuring the success of an agent there are two important aspects: win ratio and average points taken in a round. So, results are presented with these two pieces of information. In addition, moon shot ratio is important for moon shot agents and defensive agent. In the test environment, the learning agent is played against 3 of the opponent agents. So, a winning ratio of 25% means both parties are playing the game equally well. Any percentage higher than 25% means learning agent is playing better than its opponents. There are a total of 26 points to take in a round. So, if players are on the same level of ability, on average they all take 6.5 points. Since the player with the least amount of point is the winner, any point less than 6.5 means the learning agent is better.

Experiments represented in graphs in the following sections follow the principle same principle. Each point in the graph represents 1000 games. The horizontal axis is the epoch number showing the training progress of the tested agent. The vertical axis can be the win ratio, moon shot ratio, or average round points depending on the graph.

5.1 Play and Pass Agents

First, the play agent is tested against both rule based and monte carlo agents. Since the aim is to see how well the pass agent has learned the game without passing introduced, in these experiments, passing and moon shot is disabled. These features will be introduced again later, as we examine other learning agents.

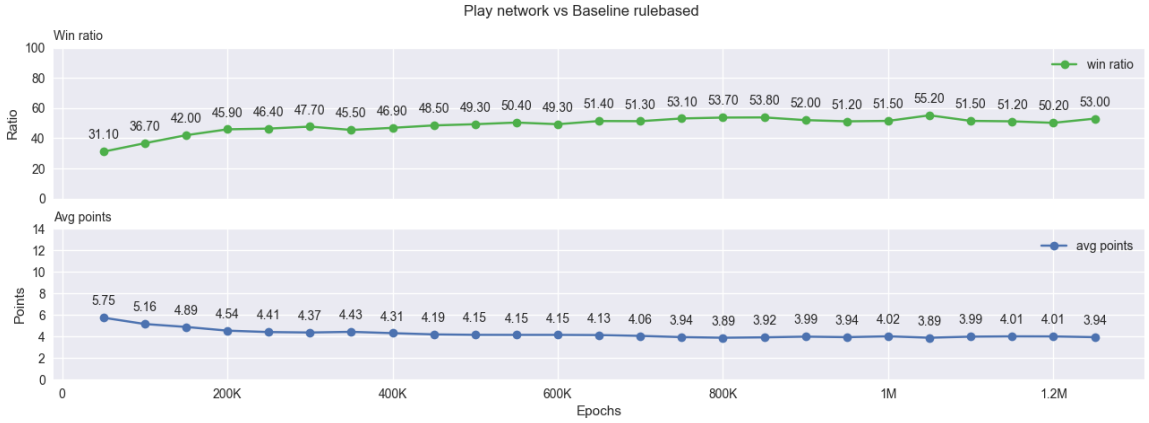

Figure 5.1: The play agent vs Rule based agent

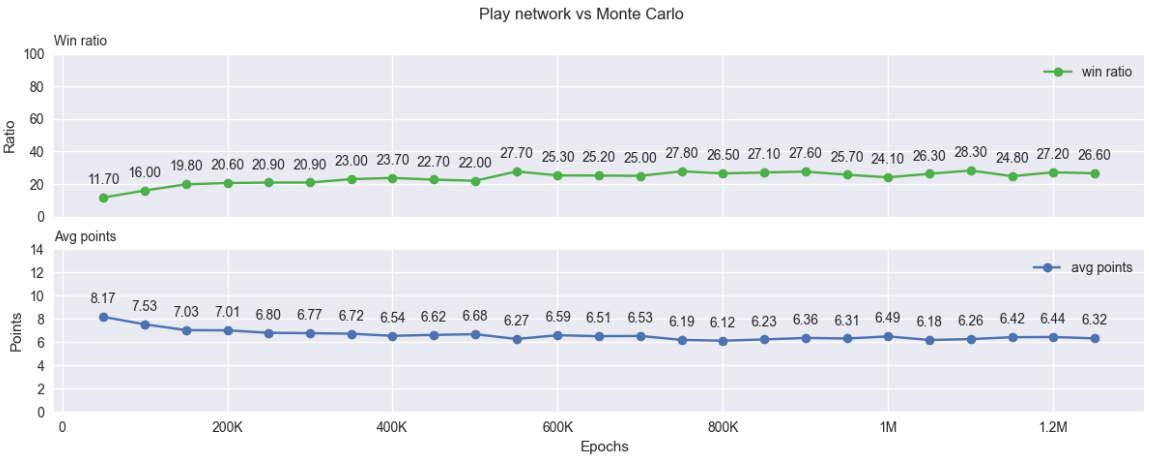

Figure 5.2: The play agent vs Monte carlo agent

In the Figure 5.1 and the Figure 5.2 learning progress of the play agent is presented. It is clear to see that the play agent is learning fast until around $300,000^{th}$ game. After that, there are some improvements, yet they are slow. Nonetheless, the agent kept learning until $800,000^{th}$ game. Beyond that point, no learning has occurred. Comparing the average point and winning ratios, the play agent is slightly better than

the monte carlo agent while it is way better than the baseline rule based agent. The play agent at the $800,000^{th}$ game is selected for the following experiments and it will be referred to as the play agent.

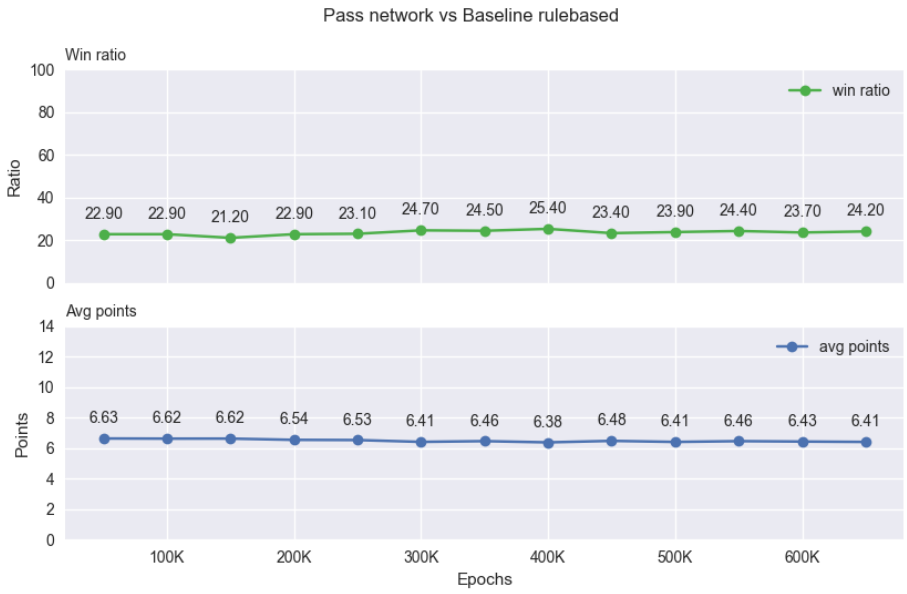

Figure 5.3: The pass agent vs Rule based agent

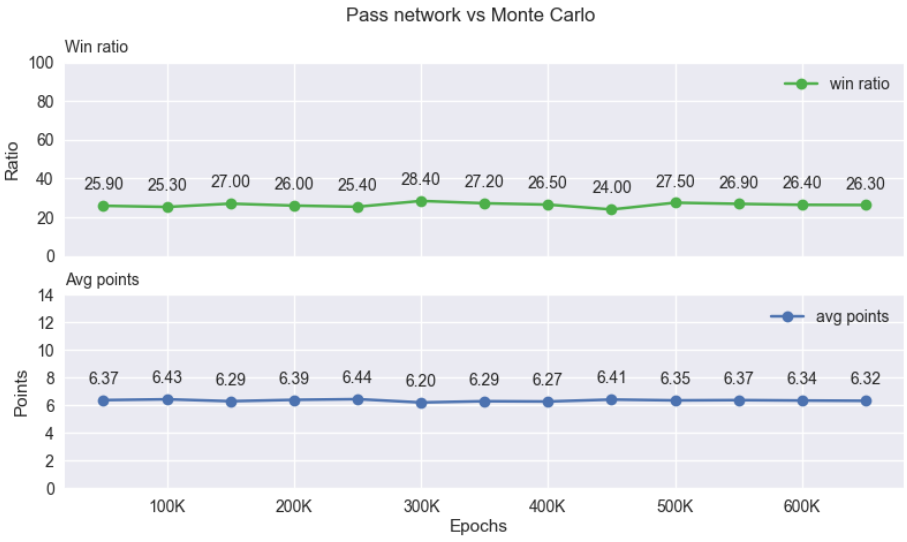

Figure 5.4: The pass agent vs Monte carlo agent

In the Figure 5.3 and the Figure 5.4, experiments are done with pass enabled and learning process of the pass agent is presented. Measuring the success of the pass agent alone is impossible, so experiments are conducted while the play agent at

$800,000^{th}$ epoch is kept fixed as the pass agent is trained. This means the x-axis of the graphs refers only the pass agent's game number. As can be seen from graphs, the pass network has completed its learning progress rather quickly within the first $50,000$ games. It performed only on the equal level of rule based and monte carlo agents. This is because it did not pass cards that the play network best plays with as is explained in the previous chapters. To overcome this, a final training process is applied where pass and play networks are trained together.

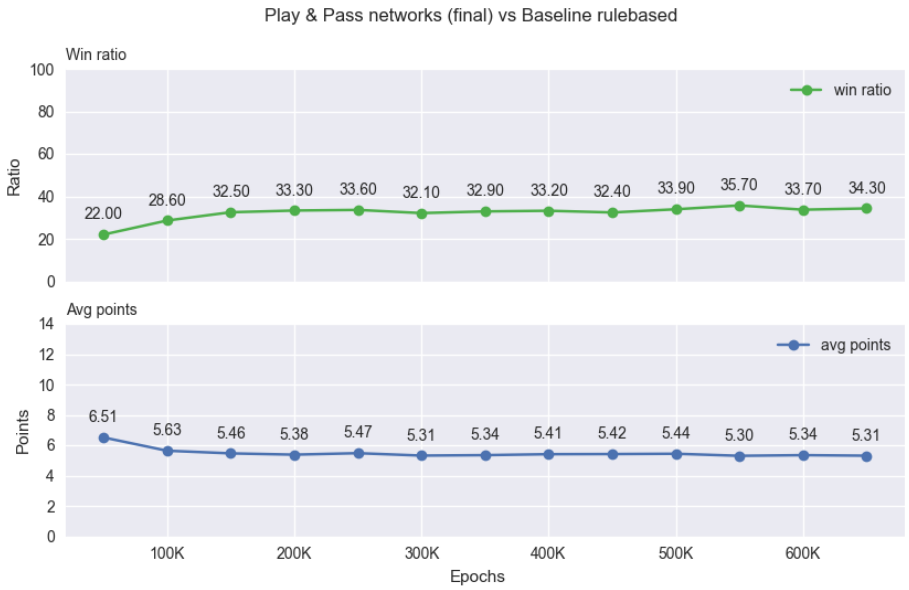

Figure 5.5: The pass and play agents vs Rule based agent

Result of final training in the play and pass networks are given in the Figure 5.5 and the Figure 5.6. After the first $200,000$ games, the learning process is slowed and not much is changed throughout the rest of the training. Results show that final training has granted networks to the ability to play together as they can manage to beat rule based and monte carlo agents with a point difference of around 2 points.

Table 5.1 show the result of the pass and play networks' final versions against human players. Moon shot is disabled during the experiment. The results are obtained among 2000 games. Three learning agents are played against one human player, this is why there is a big gap in the win ratios. In a completely equal game, humans would win 25% of the games whereas learning agents would win 75% of total games. Results show that the learning agent is slightly better than the human player because it has

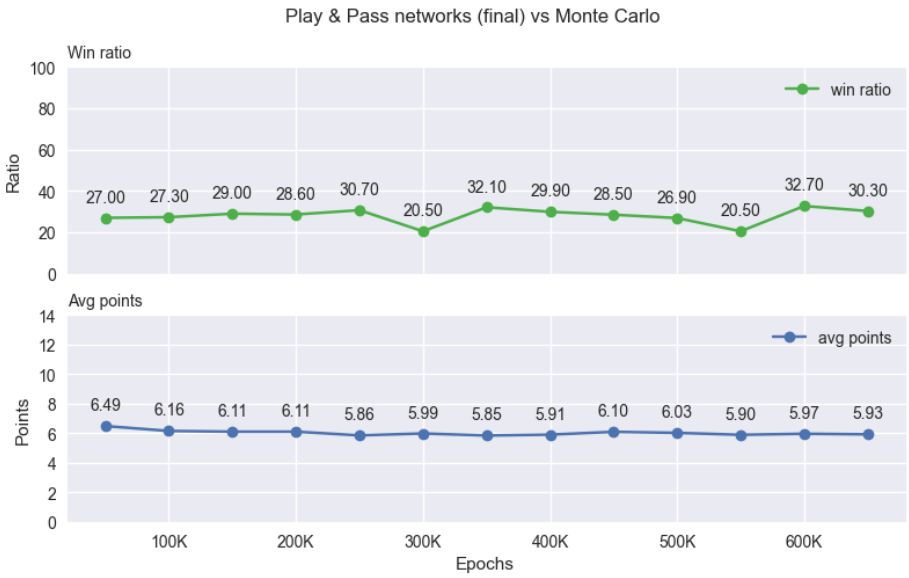

Figure 5.6: The pass and play agents vs Monte carlo agent

Table 5.1: Pass and plays agents vs Human players

	Win Ratio	Avg Points
Human	24%	6.69
Play and pass agents	76%	6.43

managed to collect fewer points and win slightly more games than the human player.

5.2 Moon Shot Agents

Moon shot process has three agents. Play, pass and decision. To test the decision agent, the labeled is divided into train data, test data. There are total of 1000000 tricks in the whole data. 20% of the data is used as test data and rest is reserved for training. Total loss and accuracy of the moon shot decision is agent is measured for all versions of the moon shot decision agent. However, only the final version is given here. While the loss is 10%, the agent can successfully predict 75% of the hands in the labelled data.

To test moon shot play agent, moon shot is enabled and passing is disabled. In the experiment, the moon shot play agent is accompanied by moon shot decision agent and normal play agent. Throughout the experiment, the same play agent and decision agent is used during the experiment while the moon shot play agent is tested in different epochs. The decision agent used in this experiment is not the final version. It is only used for training the moon shot play agent. Decision agents return the possibility of a successful moon shot.

Moon shots are rare in the game, on the average only 4% of the rounds result in moon shot. Thus, if the return value of the decision is at least 96%, meaning that given the hand, the agent is going to shoot the moon with the possibility of 96%, the agent tries moon shot, otherwise, the normal play agent decides moves.

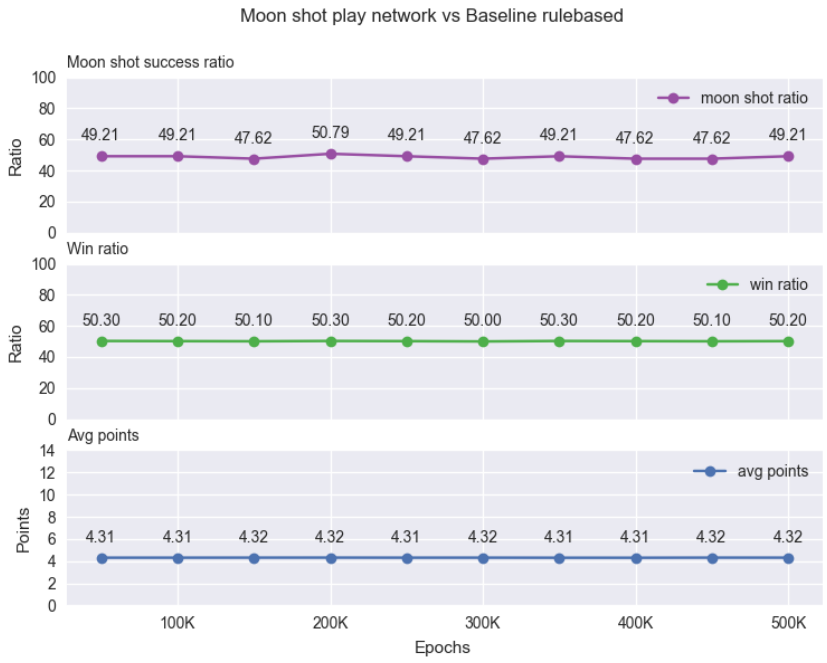

Figure 5.7: The moon shot play agent vs Rule based agent

In the Figure 5.7 and the Figure 5.8, it can be seen that the moon shot play agent has learned to play well quickly. It has managed to beat rule based and monte carlo agents. Since rule based does not prevent or try to moon shot, it performed very poorly against the moon shot play agent. On the other hand, monte carlo agent tries to prevent moon shot in a simple way, which improves the situation but the moon shot play agent still beats it.

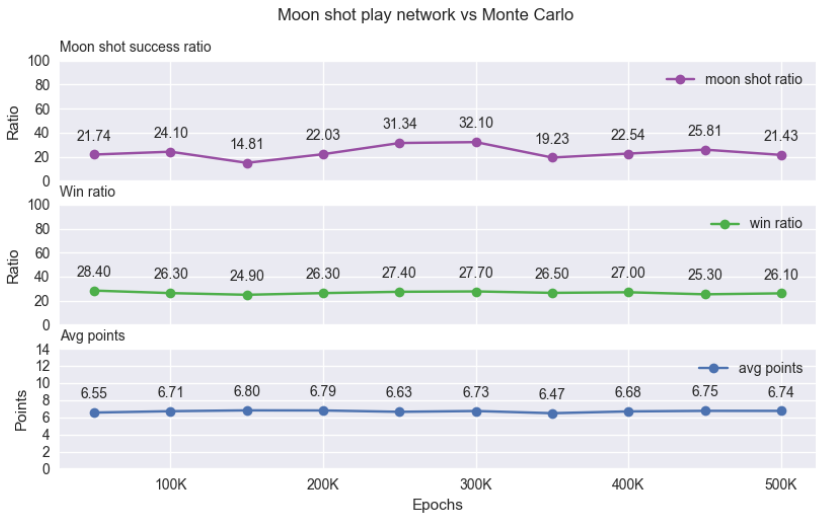

Figure 5.8: The moon shot play agent vs Monte carlo agent

The following experiment is to measure the success of the moon shot pass agent. In this experiment, both moon shot and passing are enabled. The method followed in this experiment is the same as the previous one with one difference. This time pass agent is added so that when the decision agent decides not to shoot the moon, the pass agent makes moves. The horizontal axis represents the epoch number of the moon shot pass agent while the best versions of other learning agents are used from previous experiments.

Figure 5.9 and Figure 5.10 shows us that, as it is with the moon shot play agent, moon shot pass agent also learned fast. As the epoch number increases, the win ratio and the average points taken do not change in either of the figures. However, moon shot success ratio increases. This means that the accuracy of the decision agent increases. Note that successful moon shots do not affect the player's average point positively. It rather increases the opponent's average point. This is why, in figures, even tough player's average point is not relatively small, its win ratio is high.

The last experiment is the fine tuning process of the moon shot agents. In this environment, the final decision agent is used. This time, the horizontal axis represents both moon shot play and moon shot pass network's epochs.

In the Figure 5.11 and the Figure 5.12, although it is unexpected, we do not see change that can be considered as an improvement. In the normal play and pass agents' case,

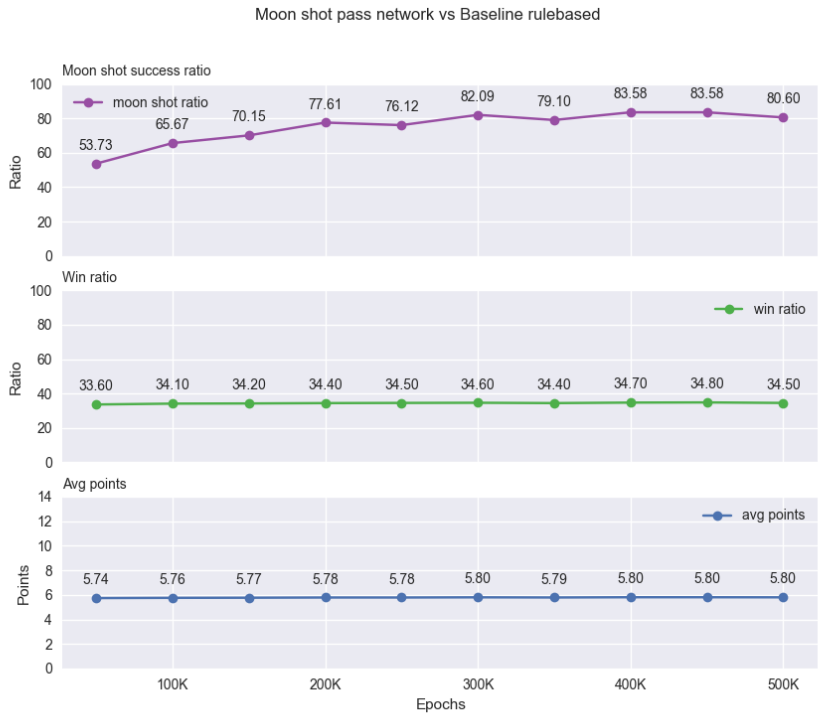

Figure 5.9: The moon shot pass agent vs Rule based agent

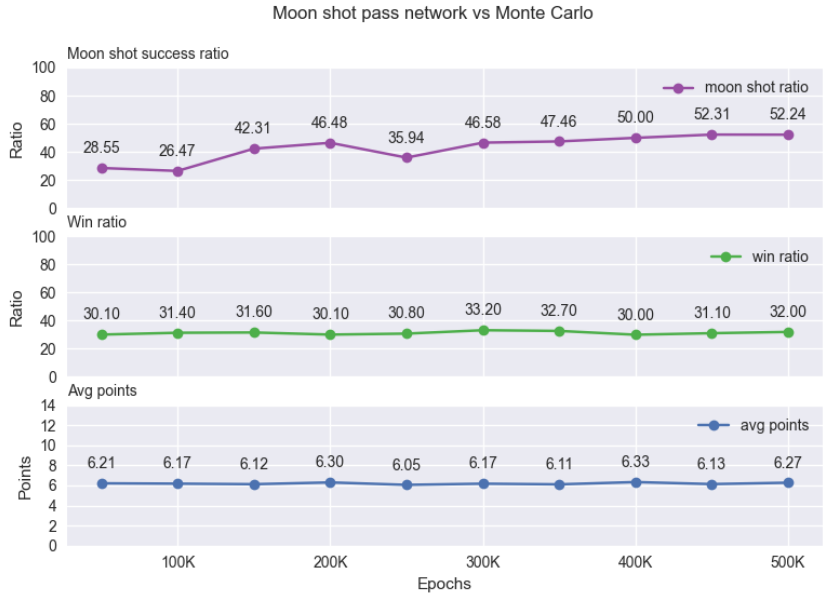

Figure 5.10: The moon shot pass agent vs Monte carlo agent

they have improved when trained together. Here, however, training together does not affect the success of agents. This means that moon shot play and pass networks are

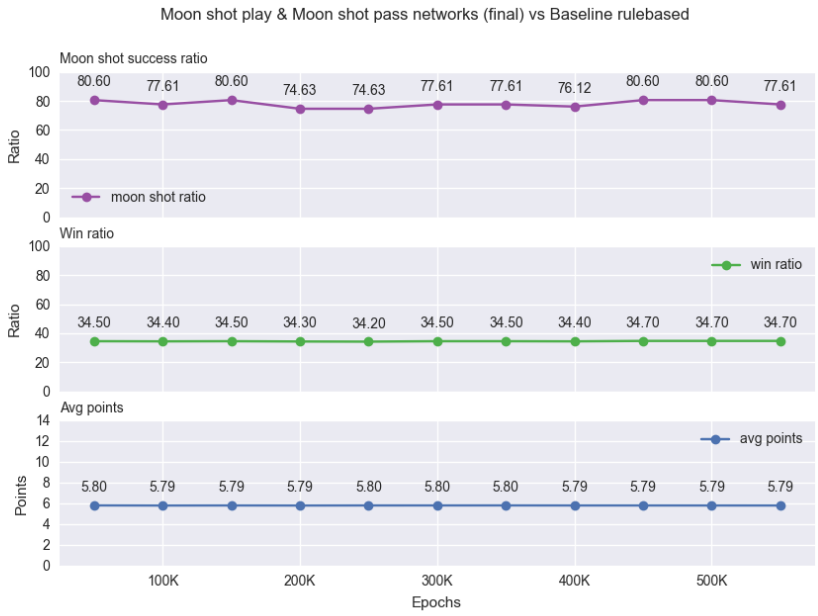

Figure 5.11: The moon shot pass and play agent vs Rule based agent

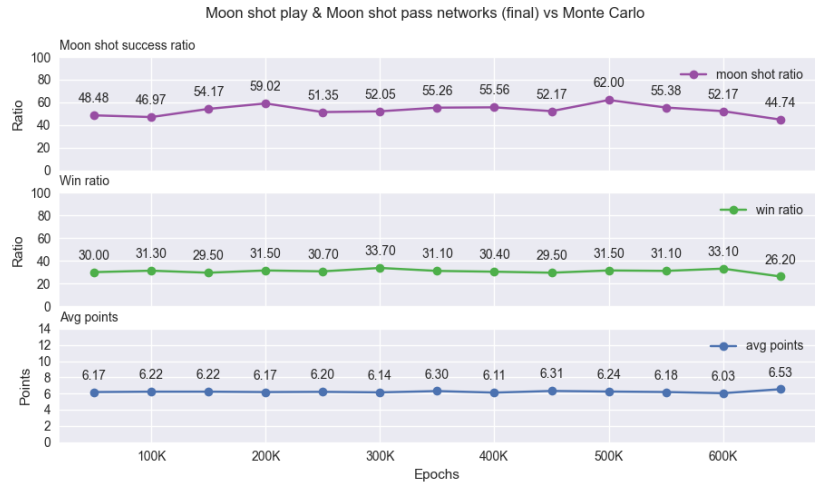

Figure 5.12: The moon shot pass and play agent vs Monte carlo agent

already playing compatible with each other.

Table 5.2 show the result of the moon shot pass and play networks' final versions against human players. Same as the play network experiment, the results are obtained among 2000 games. Three learning agents are played against one human player. When moon shot is added, the overall success of the learning agent is decreased. The average points are still pretty close, but, since humans are better at shooting the

Table 5.2: Moon shot pass and plays agents vs Human players

	Win Ratio	Avg Points
Human	30%	7.57
Moon shot agents	70%	7.65

moon, they win more games overall. The accuracy, percentage of successful moon shot attempts, of moon shot agent is 30%. When tested against monte carlo agent this ratio was over 50%. This means humans are more capable of preventing moon shot than monte carlo agent and that is directly reflected to experimental results.

5.3 Defensive Agents

The final experiment is to measure the success of defensive agents. First defensive decision agent is tested. Same as moon shot decision test, the labeled is divided into train data, test data. There are total of 1000000 tricks in the whole data. 20% of the data is used as test data and rest is used for training. The loss of the decision agent is 10%, while the agent can successfully predict 85% of the hands in the labelled data.

Since rule based agent and monte carlo agent do not try to shoot the moon, the defensive play agent is tested against non-defensive version of the learning agent. During this experiment, the games are played without any restrictions. Passing and moon shot are enabled. Also, as a defensive play agent, the final version of the learning agent is used as shown in Figure 4.1. In the experiments, one defensive play agent is played against three non-defensive learning agents. The horizontal axis represents the epoch of the defensive play agent. Like other agents, whichever version of them performed best in the previous experiments is used.

Experimental results are shown in the Figure 5.13. Adding defensive capabilities results in worse performance in win ratio. However, it has managed to cut down the total moon shot of the learning agent. It is expected because data of the defensive decision agent is prepared with games of the learning agent. Total moon shot count

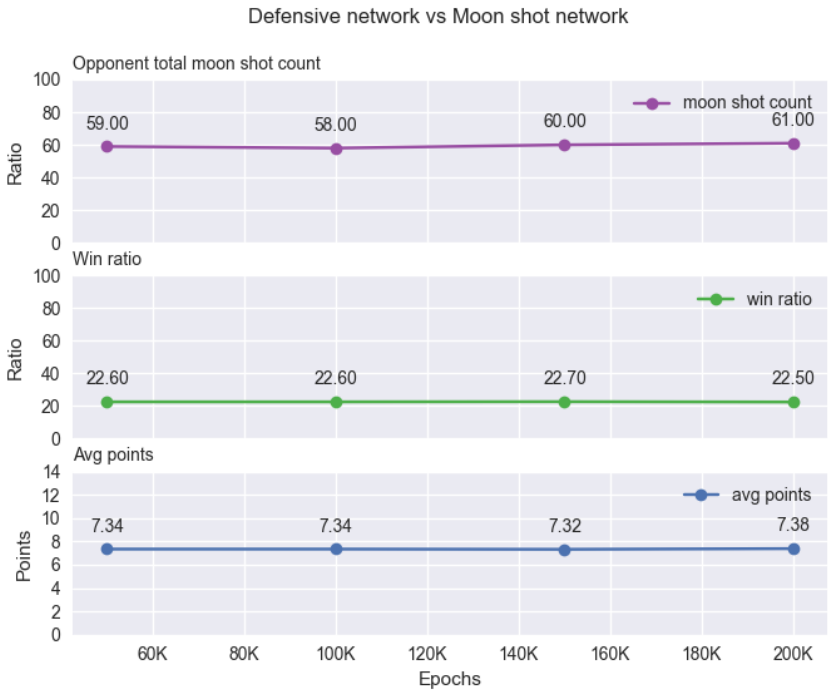

Figure 5.13: The moon shot play agent vs Rule based agent

Table 5.3: AI agent success with and without defensive

	Win Ratio	Avg Points	Opponent Moon shot Count
Defensive On	22.5	7.38	61
Defensive Off	24.8	7.48	111

of opponents are given in the figure. As defensive play agent trains, the opponent's moon shot count decreases to around 60 times per 1000 games. The decrease in the total moon shot count is not visible from Figure 5.13 since the defensive play agent has reached minima in quite a few epochs and its performance did not change after that.

The Table 5.3 compares defensive agent and non-defensive agent more clearly. To prepare the table, the same set of 1000 games are played against 3 non-defensive learning agents. In one set the defensive play agent, in the other non-defensive agent

is played against them. As it can be seen from the figure, the win ratio of the non-defensive agent is better compared to the defensive one. Even though the win ratio is worse, the average points of the defensive play agent may seem to be better. This is because defensive play agent lets fewer moon shots, thus, its average point is less affected by it. When the agent plays defensively, moon shot count decreases to 61 from 111 which is about 45%.

Table 5.4: Defensive agent vs Human players

	Win Ratio	Avg Points
Human	32%	7.22
Defensive agents	68%	7.81

Table 5.4 show the result of the defensive play network's final version against human players. The results are obtained among 2000 games. Three learning agents are played against one human player. When defensive capabilities are added, the gap between humans and the learning agent is increased. Humans have managed to shoot the moon 6.55% percent of total rounds before the defensive agent is added. The defensive agent decreased this ratio to only 5.90%. Humans still shoot the moon with an important success ratio. This has two main reasons. One, defensive decision agent has trained on the games of moon shot agent's games. The way human tries to shoot the moon and the way of moon shot agent's are different. That means the defensive agent is more successful against the learning agent. Two, overall moon shot success of learning agents is decreased since the defensive agent also tries to prevent other learning agents' moon shot attempts.

The defensive decision agent runs on every trick to decide if the agent should try to defend against a possible moon shot attempt or play normally. To test the defensive agent's success without any decision process, a pseudo experiment is conducted and the decision process is left out. In this pseudo experiment, until the moon shot threat is eliminated, namely two or more player collected points, the learning agent plays defensively. Then it continues to play normally. In this scenario, humans have managed to shoot the moon in only 2% of the total rounds. However, playing aggressively

defensive causes taking many unnecessary points, hence, the win ratio of human's jumps to 40%. The pseudo experiment shows that with a better dataset, tailored for human behavior, the defensive agent can achieve success.

CHAPTER 6

CONCLUSIONS

In this study, an RL method is presented to the game of Hearts which is a trick based, imperfect information card game. Hearts is one of the most popular card games in the world. Having some different rules, passing and moon shot, from other card games makes hearts challenging and hard to master. In hearts, at the beginning of each round players pass three cards to their opponents. This part is called the passing phase of the game. After the passing phase is completed, the normal play phase begins. This phase is common in many card games, players play a card, the highest card wins the trick. In hearts, taking less trick, thus fewer points, is better. However there is one exception, if a player takes all points in a trick, he/she gets 0 points while other players are punished with points instead.

While the common approach is to do training with one network, in this study, several networks are trained for each aspect of the game. It is hard to teach all of the challenging rules of hearts in a single network. By dividing the learning process into smaller parts, each agent is focused on a relatively easy part of the games and this makes learning easier and faster. Both one network and several networks approaches have advantages and disadvantages. When training is done with one network, it is hard to teach contradicting rules of the game such as normal playing, where the aim is to take fewer point, and moon shot, where the aim is to take all points. However, it is much easier to achieve good performance once these challenges are assigned to separate networks. In fact, it is easy to learn simple things, yet, maintaining cooperation between these networks is an issue. The cooperation problem is not present in one network approach, because there is only one network. The cooperation is achieved in this study by training networks together which referred as fine tuning process in

Chapter 4. This is why work is divided into smaller parts. The passing phase and playing phase are learned by different agents. Similarly, moon shot agents are separated. In the final version, at the start of each round, a supervised agent first checks if it is possible to shoot the moon. If yes, moon shot pass and moon shot play agents play the tricks. If it is not likely to shoot the moon, before playing a card, another supervised agent decides if opponents may try to shoot the moon. If yes, the defensive agent decides cards to play, else pass and play networks decides.

Experiments are incrementally conducted in this study. First, normal play and pass agents are tested. They have performed incredibly well against rule based and monte carlo agents. Moreover, it is shown in the experiments that they are slightly better than human players. Second, moon shot capability is added. As rule based and monte carlo agents are not good at preventing moon shot, they performed very poorly against the learning agent. Humans, on the other hand, are much better at detecting and preventing moon shot. Although the average points taken are very close, humans managed to beat the learning agent since they shot the moon more effectively. Last, defensive capabilities are added. In defensive mode, the learning agent tries to prevent a possible moon shot attempt and in the process may take some points. With defensive capabilities added, the learning agent managed to decrease the moon shot count of other learning agents. However, it affected games against human players in a bad way. This is mostly because training data of the defensive decision agent is sampled from the games of the learning agent. The way the learning agent tries to moon shot and the way humans try to do so are different. In spite of that, defensive agent, however small, decreases the moon shot ratio of the human player. As future work, the moon shot agent and the defensive agent can be further trained with human games samples.

REFERENCES

[1] T. M. Mitchell, *Machine Learning*. New York: McGraw-Hill, 1997.

[2] M. Bowling, J. Fürnkranz, T. Graepel, and R. Musick, "Machine learning and games," *Machine Learning*, vol. 63, pp. 211–215, 06 2006.

[3] M. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *Journal of Artificial Intelligence Research*, vol. 47, 07 2012.

[4] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016. cite arxiv:1606.01540.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013. cite arxiv:1312.5602Comment: NIPS Deep Learning Workshop 2013.

[6] A. L. Samuel, "Some studies in machine learning using the game of checkers. ii: Recent progress," *IBM J. Res. Dev.*, vol. 11, p. 601–617, Nov. 1967.

[7] A. L. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal of Research and Development*, vol. 3, pp. 210–229, July 1959.

[8] G. Tesauro, "Temporal difference learning and td-gammon.," *Commun. ACM*, vol. 38, no. 3, pp. 58–68, 1995.

[9] M. Campbell, A. J. Hoane, and F.-h. Hsu, "Deep blue," *Artif. Intell.*, vol. 134, p. 57–83, Jan. 2002.

[10] J. Tomayko, "Behind deep blue: Building the computer that defeated the world chess champion (review)," *Technology and Culture*, vol. 44, pp. 634–635, 01 2003.

[11] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Diele-

man, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–489, Jan. 2016.

[12] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of go without human knowledge," *Nature*, vol. 550, pp. 354–, Oct. 2017.

[13] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[14] Baykal and F. Alpaslan, "Reinforcement learning in card game environments using monte carlo methods and artificial neural networks," pp. 1–6, 09 2019.

[15] W. L. Santo and A. K. W. Wong, "Evaluating mcts in a new ai framework for hearts," 2015.

[16] N. R. Sturtevant and R. E. Korf, *Multiplayer Games: Algorithms and Approaches*. PhD thesis, 2003. AAI3089030.

[17] N. R. Sturtevant and A. M. White, "Feature construction for reinforcement learning in hearts," in *Computers and Games* (H. J. van den Herik, P. Ciancarini, and H. H. L. M. J. Donkers, eds.), (Berlin, Heidelberg), pp. 122–134, Springer Berlin Heidelberg, 2007.

[18] L. Kuvayev, "Learning to play hearts," in *AAAI/IAAI*, 1997.

[19] S. Ishii, H. Fujita, M. Mitsutake, T. Yamazaki, J. Matsuda, and Y. Matsuno, "A reinforcement learning scheme for a partially-observable multi-agent game," *Machine Learning*, vol. 59, pp. 31–54, 05 2005.

[20] E. Alpaydin, *Introduction to Machine Learning*. Adaptive Computation and Machine Learning, Cambridge, MA: MIT Press, 3 ed., 2014.

[21] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. The MIT Press, second ed., 2018.