Developing an Artificial Intelligence, using current Reinforcement Learning techniques, for games

MASTER’S THESIS
Department of Computer Science
Science & Technology
Aarhus University
June 2017

Advisor: Christian Storm Pedersen
The development of full artificial intelligence could spell the end of the human race.

— Stephen Hawking
ABSTRACT

Our focus for this thesis, is a look at the practical use for Reinforcement Learning (RL), in the context of creating games. In contrast to many other efforts in the field, we investigate the possible uses today’s technology have, when developing an Artificial Intelligence (AI) under the practical limitations games face today. In order to better reflect on the whole process of developing a game, together with a RL-AI, this thesis will cover the whole process, from developing a game, to looking at a wide range of techniques in order to develop the corresponding AIs.

The game was designed to be sufficiently complex to pose a challenge for the AIs we develop, and avoid being a “best-case” problem. We designed the core of the game-play based on our personal abilities and ideas, without RL being a key-factor.

Since RL is a huge subject, and many different techniques exists within this subject, we decided to focus our RL endeavours into deep learning, utilizing Q-value Temporal Difference (TD)- and evolutionary learning techniques. As well as the usage of game-recordings in order to learn from observing experts play.

Using these methods, we create a three step rocket of RL. The first step is the aforementioned training on logs, from other AIs playing the game to form an initial model. Afterwards, we use evolutionary learning to do a wide exploration of the feature space. In the final step, we use deep Q-learning to follow the gradients to a local minimum.

We found that designing both the game and parts of the RL, takes a great deal of care and diligence. The game that we designed might have been too much of a challenge for RL to cope with, under the limitations we had set for ourselves. Furthermore, designing the reward function for the deep Q-learning training proved to be a very complex issue, with great impact on the learning process. In spite of the issues we had during the thesis, we got some useful results from our experiments. We find that both the deep Q-learning and evolutionary learning techniques are useful for training an AI to play the game. However, we fail to surpass the hand-written AIs, written by our fellow students during a one-day hackathon.
ACKNOWLEDGMENTS

We want to acknowledge the time and work that our supervisor, Christian Storm Pedersen, has put into our thesis. We appreciate the informal meetings and the independence we have had throughout this thesis.

Lukas wants to acknowledge: My girl-friend Alexandra Aldershaab Hou Olsen, as she has provided me with incredible support during this thesis. The Torque3D community, for inspiring me and helping me getting introduced to Computer Science since I was 15. Last, my family who are always there to pull me through when things get rough, and of course all my friends.

Simon wants to acknowledge: My family for providing dozens of support. I will especially acknowledge my two brothers who helped correct the thesis and always provide me with fun and games. My girlfriend Joan also deserves an acknowledgement due to the hours and hours that she spend with me during the master thesis. Lastly i would like to acknowledge all my friends for enduring my personality and for being there for me always.
# CONTENTS

1 Introduction 1
   1.1 Game Intro 2
   1.2 Neural Networks 3
   1.3 Reinforcement Learning 6
      1.3.1 Markov Decision Process 9
      1.3.2 Value Functions 10

2 Related Work 13
   2.1 Playing Atari with Deep Reinforcement Learning 13
   2.2 DeepMind Lab 14
   2.3 OpenAI Gym 14
   2.4 Frameworks and Technologies 14
      2.4.1 TensorFlow 14

3 Analysis 17

4 Thesis Outline 19

I The Game

5 Design 23
   5.1 The game mechanics 23
      5.1.1 Actions 23
      5.1.2 Feature Vector 24
   5.2 Game Elements 26

6 Implementation 29

7 The Adversaries 31
   7.1 Evaluation of the tournament 32
   7.2 Continuing efforts 32

8 Evaluation 35
   8.1 Feature Vector 35
   8.2 AI Vision 35
   8.3 Extensions 36

II Learning and Training

9 Design 39
   9.1 Choice of Machine Learning technique 39
   9.2 Architecture 39
      9.2.1 Recurrent Network 40
      9.2.2 Deep Q 42
      9.2.3 Double Q 44
      9.2.4 Dueling Q 44
   9.3 Extensions 45
      9.3.1 Adaptive Computation Time 45
      9.3.2 Network Evolution 47
      9.3.3 Hybrid Training 48
| Figure 1.1 | Screenshot of the game                       | 3  |
| Figure 1.2 | Structure of a neuron                        | 4  |
| Figure 1.3 | Artificial Neuron                            | 4  |
| Figure 1.4 | Reinforcement Learning model                 | 7  |
| Figure 4.1 | The relations between the components of this thesis | 20 |
| Figure 9.1 | An overview of the network architecture      | 40 |
| Figure 9.2 | A RNN, figure from [18]                      | 40 |
| Figure 9.3 | An unrolled RNN, figure from [18]            | 41 |
| Figure 9.4 | A short-term dependency, figure from [18]    | 41 |
| Figure 9.5 | A long-term dependency, figure from [18]     | 41 |
| Figure 9.6 | A simple RNN layer, figure from [18]         | 42 |
| Figure 9.7 | A LSTM layer, figure from [18]               | 42 |
| Figure 11.1| An overview of the different experiments made in this thesis | 59 |
| Figure 11.2| Running times for different batch sizes      | 61 |
| Figure 11.3| Evolutionary learning with and without logs | 64 |
| Figure 11.4| Reward for \( \frac{1}{1000} \) updates      | 65 |
| Figure 11.5| Q and target Q-values for \( \frac{1}{1000} \) updates | 66 |
| Figure 11.6| Q-values for updates each training step      | 66 |
| Figure 11.7| Q-values for different pre-train steps       | 67 |
| Figure 11.8| Fitness for all 4 AIs                        | 69 |
| Figure 11.9| Estimated win-rate for AI trained against FreksenThink | 70 |
| Figure 11.10| Average vs. Final Q-values                   | 71 |
| Figure 11.11| Rewards vs. Q-values                         | 72 |
| Figure 11.12| Estimated win-rate for the AI trained against ScottSteiner | 73 |
| Figure 11.13| Average Q-values for RLS Wochenstein           | 73 |
| Figure 11.14| Estimated win-rate for the AI trained against BAI | 74 |
| Figure 11.15| Final Q-values for the AI trained against BAI | 75 |
| Figure 11.16| Estimated win-rate for the AI that trained against a mix of adversaries | 76 |
| Figure 11.17| Average Q-values for the AI that trained against a mix of adversaries | 76 |
| Figure 11.18| Fitness for extended evolution against BAI    | 78 |
| Figure 11.19| Selected actions through time                | 79 |
| Figure 11.20| Estimated win-rate for punishing AI           | 80 |
| Figure 11.21| Q-values for punishing AI                    | 80 |
| Figure 11.22| Histogram over the selected actions, by RLPunisher | 81 |
Figure 11.23  Histogram over actions taken after a continued learning on logs ......................... 82
Figure 11.24  Q-values gained during continued learning on logs ........................................ 83
Figure 11.25  Q-values gained when learning to copy the actions of BAI ................................. 84
Figure 11.26  The estimated amount of correct guesses, while learning to copy BAI ..................... 85
Figure 11.27  Actions of the AI trying to copy BAI, vs BAI itself ........................................... 85
Figure 11.28  Actions taken by Simple Log, Copy and BAI ....................................................... 86
Figure A.1   Round-robin brackets ................................................................. 99
Figure A.2   Double elimination brackets .............................................................. 99
Figure B.1   Final tournament match results, rounds 1 – 7, courtesy of http://challonge.com/ .... 101
Figure B.2   Final tournament match results, rounds 8 – 15, courtesy of http://challonge.com/ .... 102

LIST OF TABLES

Table 5.1    Descriptions of feature-vector Content ......................................................... 25
Table 7.1    Standings from the tournament ................................................................. 31
Table 7.2    Standings of the adversaries following the ongoing efforts ............................ 33
Table 9.1    Hyper-Parameters ....................................................................................... 51
Table 11.1   Naming convention of the different RL-AIs ............................................... 60
Table 11.2   Relative processing times for different adversaries ....................................... 69
Table 11.3   Final placements of the AIs ....................................................................... 88
<table>
<thead>
<tr>
<th>ACRONYMS</th>
<th>FULL FORM</th>
</tr>
</thead>
<tbody>
<tr>
<td>AI</td>
<td>Artificial Intelligence</td>
</tr>
<tr>
<td>RL</td>
<td>Reinforcement Learning</td>
</tr>
<tr>
<td>NN</td>
<td>Neural Network</td>
</tr>
<tr>
<td>ML</td>
<td>Machine Learning</td>
</tr>
<tr>
<td>DQN</td>
<td>Deep Q Network</td>
</tr>
<tr>
<td>FOV</td>
<td>Field Of View</td>
</tr>
<tr>
<td>LOS</td>
<td>Line Of Sight</td>
</tr>
<tr>
<td>DRN</td>
<td>Deep Recurrent Network</td>
</tr>
<tr>
<td>RNN</td>
<td>Recurrent Neural Network</td>
</tr>
<tr>
<td>LSTM</td>
<td>Long Short-Term Memory</td>
</tr>
<tr>
<td>ACT</td>
<td>Adaptive Computation Time</td>
</tr>
<tr>
<td>DDQNRN</td>
<td>Dueling Double-Q Recurrent Network</td>
</tr>
<tr>
<td>MDP</td>
<td>Markov Decision Processes</td>
</tr>
<tr>
<td>TD</td>
<td>Temporal Difference</td>
</tr>
<tr>
<td>POV</td>
<td>Point Of View</td>
</tr>
</tbody>
</table>
INTRODUCTION

The use of Reinforcement Learning (RL) in computer games, is a very popular topic. It gathers a lot of attention, and the data for the learner is relatively easy to retrieve, as it is already digitized. There are many different initiatives, trying to improve the Artificial Intelligence (AI)s in games using RL, the most prominent one being DeepMind Technologies [3]. Their Deep Q Network (DQN) ignited the field of RL in AI, when they successfully created an AI which could play Atari games [16], just by providing it with the pixels shown on the screen. They have since mastered the game of Go [4], and are currently trying to develop an AI for the game called Starcraft 21.

In this thesis, we take the position of an ambitious game developer, in the present day, who wants to create a sophisticated AI for his game. The game is a top-down real-time shooter, so the action will be instant and simultaneous. Machine Learning (ML) brings many advantages for this issue, the most prominent one being the small amount of human resources needed to actually improve the AI after the initial set-up; human resources are scarce for most game-developers, so they have minimal resources to devote to each aspect of the game.

Presently, the downside to ML is the immaturity of the field with regards to RL, mostly prominent in the necessity for high amounts of computation time needed for training. Therefore, we have chosen to investigate how the ML tools, of today, perform when given limited computational resources, for a fairly complex problem.

This thesis has two major parts, the first being about the game design and development, the second being about learning how to beat the game. In the first part, we describe how we created the game, the choices we made in the design etc. Furthermore, we describe an event we arranged, a so called hackathon, where our fellow students developed black-box adversaries which we would use later on, for evaluation and training of our AI.

In the second part of the thesis, we look at how our AI would learn how to play the game, by interacting with adversaries from the hackathon, which requires the usage of state-of-the-art techniques. RL is the natural fit for the problem and all the newest progress in the field, has been achieved with Neural Networks (NNs). We will therefore design a Recurrent Neural Network (RNN)-architecture and train it to solve the problem, using a broad range of techniques. We utilize both gradient-descent and evolutionary learning, as training

techniques. Furthermore, we apply a wide range of architectures like the Deep-Q [16], Dueling Q [24], Double Q [8] and Long Short-Term Memory (LSTM) [10] architectures. We also utilize a technique called experience replay [15]. These are all techniques that are used throughout the field already, and have proven their worth in many experiments.

After we designed our architecture and training techniques, we created a series of experiments. The primary focus was on the difference between learning efficiency, depending on the adversary our AI was playing against. Our primary experiment, was the following three-staged training process: In the first stage, we trained a base-line AI by letting it observe matches between two AIs. In the second stage, we used evolutionary learning to train against various adversaries, using the base-line AI as a starting point. In the third step we used gradient-descent training to further improve the result of the evolutionary learning.

We also had a secondary focus on exploring other learning techniques, and processes in order to create the best AI. We created a wide variety of AIs, and evaluated them by letting them compete in a final tournament. We experienced limited, but not insignificant, success. The AIs we developed were not able to win the tournament, but they were not failing completely either.

All along the way, there were countless subjects that calls for complete immersion. We could have taken a thorough look at the game-design, the coding-event we arranged and there are countless aspects of RL and its techniques that begs for further study. However, since we decided to take a look at the complete process of the ambitious game developer, we had to take a broad look over all the subjects and instead of a deep investigation of each individual technique, we focused on the application, and combination of the techniques.

This thesis is founded upon a broad foundation of knowledge, which is essential to the understanding of the whole. In order to alleviate the need for this knowledge, we will give an introduction to the underlying ideas that is the foundation for this thesis. First of all, we will give an introduction to the game itself, what kind of game it is and how we were inspired to make such a game. Secondly, we will give an introduction to NN in Section 1.2, as well as an introduction to RL in Section 1.3.

1.1 GAME INTRO

Originally inspired by a game in development by YorkshireRifles\(^2\), we decided to develop a top-down shooter. Our idea was originally, that players would have a limited Field Of View (FOV), similar to the game

\(^2\) [http://yorkshirerifles.blogspot.com/](http://yorkshirerifles.blogspot.com/)
Darkwood\textsuperscript{3}, so they would not know where the opponent was at all times. We had different ideas for how to add extra complexity to the game, such as power-ups, but decided that we would keep the game idea down to a simple shooter where two players engage in combat. In order to emulate a recoil effect, we compute a probability of hitting the enemy based on how accurately the player aims at that enemy.

Each time a new round starts, the map is populated by a random set of obstacles. This makes the game-play two-fold: the player has to both explore the map, in order to find the enemy, and do battle with the enemy. A visual of the final game, is presented in Figure 1.1, where the aforementioned FOV frustums are rendered with red lines from the players. We will explain the rest of the features that are present in Figure 1.1 in Part i.

![Figure 1.1: Screenshot of the game](http://store.steampowered.com/app/274520/Darkwood)

1.2 NEURAL NETWORKS

Using a NN to approximate some unknown function, is a technique that is widely used in ML, both for supervised learning (e.g. learning how to classify images), and RL. In this thesis, we will only consider the use of NNs in RL. NNs are generally used to approximate non-linear functions. They do this by using a model that is inspired by neurobiological brain research.
Looking at Figure 1.2 we see the structure of a neuron from the human brain. Each neuron is able to decide on whether or not a signal is propagated through the neuron, to the rest of the brain. The neuron sends a signal if the overall potential of the cell body is above some threshold. The individual input that the neuron accepts at the dendrites manipulate the cell potential, and as such is responsible for the signal propagation. The inputs are graded such that some might increase the potential and others might lower it. When we learn something, it is the grading of the neuron inputs that change. Changing the grading of the inputs might seem like such a small change, that it is difficult to comprehend how it results in the learning of complicated structures and reasoning. The human brain is able to learn this way, because it consists of billions of neurons that change these gradings all the time, allowing it to make billions of small changes to how it perceives and reacts to input.

Artificial Neuron

We will now define how one would model a artificial neuron (see Figure 1.3), from the description of the real neuron. The artificial neuron consists of a node containing an activation function, which re-
receives and emits signals. It receives the inputs $x_1, x_2, \ldots, x_n$ from $n$ different nodes connected with weights $w_1, w_2, \ldots, w_n$. The input signal is then multiplied by the weights, and summed such that we get the input to the node as:

$$s_{\text{node}} = \sum_{i=1}^{n} x_i \cdot w_i$$  \hfill (1.1)

From the neurobiology, we can take the idea of thresholds and supply a threshold activation function to our node, such that if $s_{\text{node}}$ exceeds some threshold then the node will propagate the signal to a number of output nodes, each weighted differently.

Given our model of a single artificial neuron, we will take a look at how several neurons are combined in order to make up a model of a brain. The combination of neurons can be done in numerous ways with e.g. recurrent and/or deep structures. Similar for most of the structures, is the fact that neurons are set-up in layers such that nodes at layer 0 are the input nodes and nodes at layer $n$ only transmits to nodes at layer $n+1$ and the last layer is the output of the network. If we look at Equation 1.1, we see that we can use matrix multiplication to represent the inputs at one layer.

$$S_l = \theta_l \cdot x$$  \hfill (1.2)

Where $\theta_l$ is the weights of the input to the nodes in layer $l$ laid out in rows, $x$ is the inputs from the layer below and $S_l$ is the vector of inputs to the nodes at layer $l$.

**Approximation**

In order to see that such a network approximates some non-linear function, we will assume that we have some function $g(x)$ which we want to approximate, using a NN with a weight matrix $\theta$. We can then define the function approximated by the NN by the function parametrized by the weight matrix as $f(x; \theta)$. Initially, we have no idea how to initialize the weights of the NN so we choose to use a random set of weights. Initializing the weights randomly is an important detail, since setting all the weights to the same constant would lead all the neurons to follow the same gradients in the gradient decent algorithm described later in this section, thus becoming identical. With the random weights on the NN, we perform a forward pass calculating the input at each layer, such that each node can activate if the signal exceeds the node’s threshold. From the forward pass, we receive some output $y_o$, since we used a random initialization of the weights, there is a very high probability that the predicted output $y_o$ is the wrong value, compared to the output of the target function $g(x)$. However, in order to determine that the value is, in fact, wrong, we
need some target value \( y_t \). How we achieve \( y_t \) can vary from problem to problem, so we will just assume it is given. Given the target value \( y_t \), we can define the error as \( \text{error} = \frac{1}{2} \| y_o - y_t \|^2 \) and minimize it with respect to the weights of the NN, \( \theta \). This minimization of the error, will continuously bring us closer to the target value. To do a minimization of the error, we need to employ the technique called gradient descent. To find the gradient of the error it has to be continuous. The target value is of no concern since it is a constant, so we only have to consider whether the approximated function is continuous. The adaptation of a hard threshold activation function from the neurobiology is an issue, since such an activation function lacks continuity. Luckily, there exists some continuous activation functions that are similar to a hard threshold, so we can keep some of the neurobiology concepts. Examples of such activation functions, that are widely used, are the sigmoid (\( \sigma \)) and hyperbolic tangent (\( \tanh \)).

We will now have to find the gradients of the network. We can find these gradients in \( O(\#\text{weights}^2) \) time, if we just do it naively. However, it turns out we can find the gradients of the network using an algorithm called backpropagation in just \( O(\#\text{weights}) \).

The gist of it is to define the error for each node \( i \) in layer \( l \), as a weighted average of the errors of the output. Except for the output layer \( l_0 \), which is the derivative to the error:

\[
\delta^{(n_{l})}_i = \frac{\partial}{\partial z^{(n_{l})}_i} \frac{1}{2} \| y_o - y_t \|^2 \\
\delta^{(l)}_i = \left( \sum_{j=1}^{s_{l+1}} \theta^{(l)}_{ji} \delta^{(l+1)}_j \phi' \left( z^{(l)}_i \right) \right)
\]

Where \( \phi \) is the chosen activation function, \( s_l \) is the number of neurons in layer \( l \) and \( z^{(l)}_i = \sum_{j=1}^{n_{l}} \theta^{(l)}_{ij} x_j + b^{(l)}_i \) i.e. the input to the node multiplied by the weights \( \theta \). Then we see that each error term for the layer \( l \) depends on the error term \( l + 1 \). Thus, we can propagate the derivatives of the error terms from layer \( l + 1 \) to layer \( l \), avoiding recomputation of these terms. Using this technique, each layer only has to compute the gradients for the nodes in that same layer, achieving the \( O(\#\text{weights}) \) running time. However, if the network has cycles then it would obviously not be possible to back-propagate the error terms in this way. Therefore, we need a workaround to handle cycles, which we will describe later in Section 9.2.1.

1.3 REINFORCEMENT LEARNING

In order to understand the need for the RL technique, we first have to describe the problems that we want to solve using it. There are a multitude of problems that we can use RL for, but they all revolve
around the basic concept of learning what actions to take with respect to maximizing a reward signal [20].

In RL, we usually have a model, with some agent that interacts with an environment at some discrete time steps $t = 0, 1, 2, \ldots, n$. Given some state $s_t$ the agent must choose to perform an action $a_t$ on the environment. This action triggers a reward $r_{t+1}$ and changes the state of the environment to $s_{t+1}$, see Figure 1.4. Given this definition of the RL model, we can define the state signal of some interaction with the environment as:

**Definition 1. State Signal**

A sequence of states $s_0, s_1, \ldots, s_t, s_{t+1}$ following from choosing actions $a_0, a_1, \ldots, a_t$ and resulting in reward $r_1, r_2, \ldots, r_{t+1}$

A state signal can be obtained by defining a policy $\pi$, which is a mapping from states to action probabilities, such that $\pi(a|s)$ is the probability of choosing $a$ when in the state $s$. If the agent chooses its actions based on the policy $\pi$, when it acts in the environment, then we obtain the state signal corresponding to the policy. The next step is to improve the policy at each time step $t$ such that we maximize the expected sum of all discounted future rewards $G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$ [20], where $\gamma$ is the constant “discount factor”. As an example of how we model a game, we will take a look at a simple example problem.

In the game of Tic-Tac-Toe, we are provided with an environment that consists of a $3 \times 3$ playing field and two players represented by $\times$ and $\circ$. The players take turns placing either a $\times$ or a $\circ$ into an empty space in the grid. A player wins, and receives a reward, if three of the corresponding symbols are in a line, either straight or diagonally. A policy is then a way to decide on a space in which to place the symbol, given some state of the game. Given such a policy, we can derive the state signal by playing this strategy against the opponent.

**Value estimation**

Tic-Tac-Toe is a very simple example, since there is a finite and relatively small, amount of states we can observe. The small amount of states makes it possible for us to create a table of all the states, in which we keep the expected value for the corresponding state. We
represent this table by a function of the state such that $V(s)$ represents the value at table entry $s$. At the beginning we have no idea which states are the “good” ones, so we fill the table with values of 0.5 representing a fifty/fifty chance to win. We can now use the Temporal Difference (TD) technique, explained in further detail later in Section 1.3.2, to update the estimates for each of the states. The update policy is as follows: If we make a move from some state $s$ to some other state $s'$ then we update the value as

$$V(s) = V(s) + \alpha \cdot [V(s') - V(s)]$$ (1.5)

Here $\alpha$ is a positive fraction called the step-size. We see, that if the move from $s$ to $s'$ ends up in a winning state then, $V(s') - V(s)$ is going to be positive and otherwise, if it is a losing state, then it is negative. This way we are able to reason about which states are “good”, since we can just look at the value of the state. An intuitive policy would be to simply choose the state which would grant the highest value when playing, but this is not always the optimal strategy. If there are states which has not been experienced, or explored, before then we will not know that the state with the highest value is always the optimal choice. This lack of exploration can be alleviated, by using some random choices in order to explore unseen states. Of course there has to be some kind of trade-off between the greediness and the amount of exploration a policy makes. The method described above is called the tabular method, due to the use of a table, to estimate the value of all of the states.

The tabular method works well when the state space is of a manageable size. However, when the state size becomes unmanageable and almost infinite, then exact state-value functions is no longer feasible. Instead, one needs to use an approximation method. A good example of an approximation method is the NN method, where functions are estimated through a complex structure of “neuron” activations and optimized with the gradient descent algorithm. For both the tabular and the randomized method [20] the RL paradigm uses Markov Decision Processes (MDPs) as an underlying model, in order to be able to reason about value estimation problems in a mathematical context. We describe the finite MDPs in the next section to give a short introduction to the model.

\section*{Evolution}

The value estimation techniques, are not the only techniques that can be used in a reinforcement learning context. Evolutionary learning is another method that can be used to optimize the policy of an agent. Evolutionary learning uses some encoding of the problem, usually called the “genome”, to create a population of individuals by performing random mutations on the genome. This population of indi-
individuals make up a generation, and each individual in the generation will be tested against the environment, in order to obtain the fitness of the individuals. The best individuals are then used to create a new population, constituting a new generation, and the bad ones are discarded. In the Tic-Tac-Toe example we can define the genome as the policy that decides whether to take some action, from a given state. The policy can then be mutated to create a population, by changing the probabilities. To select the individuals that will be allowed to survive, and be used in the next generation, we can let each individual play against an opponent a number of times. The fitness could then be measured as the number of wins the individual gets, and then the individual with the most wins moves on to the next generation.

Evolutionary learning, in the context of NN and RL, will be explained in more detail later on, in Chapter 9.

1.3.1 Markov Decision Process

If we look at a state signal (Definition 1) and the associated actions and rewards, it would be a reasonable assumption to say that the next state depends on all the experiences, states and actions, there have been up until the current experience. If this is true, then we can specify the joint probability distribution as:

\[
\text{Pr}\{s_{t+1} = s', r_{t+1} = r| (s_0, a_0, r_0), \ldots, (s_t, a_t, r_t)\}
\]

For all \(s', r\) and all past (state, action, reward) pairs \((s, a, r)\). This probability is difficult to work with, since it depends on all past experiences. However, if we were to assume that the response from the environment at time \(t + 1\), only depends on the state at time \(t\) instead, then we say that the state signal has the Markov Property and we can write the probability much nicer as [20]:

\[
p(s', r|s, a) = \text{Pr}\{s_{t+1} = s', r_{t+1} = r|s_t = s, a_t = a\}
\]

If we have a reinforcement task that satisfies the Markov Property, we call it a MDP, and it is these processes that we want to model. Looking at the Tic-Tac-Toe example, it is clear that this game would satisfy the Markov Property, as the state \(s_{t+1}\), is simply the state with the current board where a \(\times\) or \(\circ\) has been added to the selected space.

Given the probability from Equation 1.7, we are able to calculate all the probabilities that one would want.

Expected reward for state action pair: [20]

\[
r(s, a) = \mathbb{E}[r_{t+1}|s_t = s, a_t = a] = \sum_{r \in R} \sum_{s' \in S} p(s', r|s, a)
\]

State transition probability [20]:

\[
p(s'|s, a) = \text{Pr}\{s_{t+1} = s'|s_t = s, a_t = a\} = \sum_{r \in R} p(s', r|s, a)
\]
Expected reward for state-action-nextaction triplets [20]:

\[
    r(s, a, s') = \mathbb{E} \left[ r_{t+1} | s_t = s, a_t = a, s_{t+1} = s' \right] = \sum_{r \in \mathbb{R}} r \cdot p(s', r | s, a) \frac{p(s' | s, a)}{p(s' | s, a)}
\] (1.10)

1.3.2 Value Functions

To be able to handle the minimizing of the expected reward obtained during MDPs easier, we use value functions that is parameterized with the policy used and either only the state in state-value functions \( v_\pi(s) \) or both state and action in action-value functions \( q_\pi(s, a) \). In this report we will mostly consider the action-value functions because we have no way of knowing which state we end up with when we take an action, so we can’t figure out which states we can end up in and which one of those are the best. Instead, we can use the action-value function to estimate the value of the different actions from the current state. The internals of the value functions are almost the same since they both are expectations of future rewards as [20]:

\[
v_\pi(s) = \mathbb{E}_\pi [G_t | s_t = s] = \mathbb{E}_\pi \left[ \sum_{k=0}^{\infty} \gamma^k \cdot r_{t+k+1} | s_t = s \right]
\] (1.11)

\[
q_\pi(s, a) = \mathbb{E}_\pi [G_t | s_t = s, a_t = a] = \mathbb{E}_\pi \left[ \sum_{k=0}^{\infty} \gamma^k \cdot r_{t+k+1} | s_t = s, a_t = a \right]
\] (1.12)

Where \( r_{t+k+1} \) is the \( k' \)th future reward and \( \gamma \) is the discount factor. Another reason why we want to use these value functions, is that if we assume that we model a MDP, then the value functions must satisfy the Bellman Equation, i.e. an equation that relates current states and future states like [20]:

\[
v_\pi(s) = \mathbb{E}_\pi \left[ r_{t+1} + \gamma \cdot v_\pi(s') | s_t = s \right]
\] (1.13)

\[
q_\pi(s, a) = \mathbb{E}_\pi \left[ r_{t+1} + \gamma \cdot v_\pi(s') | s_t = s, a_t = a \right]
\] (1.14)

Here it is important to note that the state-value function is recursive while the action-value is not. Given that the \( v_\pi \) and \( q_\pi \) are optimal then we have a special relationship between them [20]:

\[
v_\pi(s) = \max_a q_\pi(s, a)
\] (1.15)

so we can change the Bellman Equation for the action-value function to also be recursive for the optimal strategy [20]:

\[
q_\pi(s, a) = \mathbb{E}_\pi \left[ r_{t+1} + \gamma \cdot \max_a q_\pi(s, a) | s_t = s, a_t = a \right]
\] (1.16)

Equipped with the Bellman Equation we can now try to find the optimal policy that satisfies the equation. A very popular method
to find policies, is the TD method. In the TD method, we update the action-value function as follows:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[ r_{t+1} + \gamma \cdot \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]$$  \hspace{1cm} (1.17)

Where $r_{t+1}$ is the reward obtained by taking action $a_t$ from state $s_t$ to $s_{t+1}$, $\alpha$ is the step-size and $\gamma$ is the discount factor. Here, we see that $r_{t+1} + \gamma \cdot \max_a Q(s_{t+1}, a)$ is the target value acquired from the Bellman Equation (Equation 1.16). Thus Equation 1.17 updates $Q$ to some new value, that is closer to satisfying the Bellman Equation. The TD method converges to the optimal action-value function for MDPs and given the optimal action-value function, it is easy to obtain a strategy to use, by simply choosing the action with the maximum value. [20]
In this chapter, we will explain some of the related work, that others have done. This includes papers that have a relation to the problem that we want to solve in Sections 2.1, 2.2, 2.3, as well as technologies that we use throughout the thesis in Sections 2.4, 2.4.1. In Chapter 3, we will analyze the related work and relate it to what we want to do in our thesis.

2.1 Playing Atari with Deep Reinforcement Learning

Mnih et al. [16] uses RL to train an AI to learn seven different games made by the company Atari which is known for its games and gaming consoles. Their goal is to compare the AI with expert humans, to see if it can compete with or even beat these expert humans.

The input that they use is a preprocessed version of the entire game view (210 × 160 RGB video at 60 Hz). They begin by converting the RGB values to gray-scale and then down-sample the view down to 110 × 84. This down-sampling of the input is done in order to reduce the dimension of the input, hence reducing the computational need. They crop the input further to 84 × 84, in order to utilize a GPU implementation of 2D convolution, which expects square inputs.

This down-scaled input is then fed to a deep convolutional NN consisting of two convolutional hidden layers: The first layer consists of 16 8 × 8 filters with a stride of 4, the second layer consisting of 32 4 × 4 filters with a stride of 2. After the two convolutional layers, the network consists of a fully connected hidden layer and a fully connected output layer, which outputs Q-values for each of the available actions.

They use off-policy Q-learning with experience replay for training the network. Experience replay is used to mitigate the inherent correlation in data, which is apparent in online learning algorithms. Furthermore, by using a batch of past experiences, they can learn from the same experience a number of times, instead of just once. Their algorithm can be seen at [16, p. 5].

Another technique they use, even though it is not a part of the algorithm, is to skip k frames. This means that they only choose an action every k’th frame and then replay that action in the skipped frames. This frame skipping method is used to further conserve the computational time used to predict actions.

Another thing worth noting is that this frame skipping is the only hyper parameter that is different in the different game experiments,
because some visuals would be missed if skipping is done over too many frames at a time.

The experiments they made, consisted of comparing their results with some of the current state-of-the-art RL methods and a human expert player. The results showed that their DQN out-performed the other RL methods in all of the seven games, and even did better than the expert human in three of the games.

2.2 DEEPMIND LAB

Observing a need for an easily-accessible platform for AI research, Google’s DeepMind team decided to develop freely and easily accessible AI-environment. Beattie et al. [2] presents the DeepMind Lab, an open-source engine that is tailored for ML, with an emphasis on 3D-vision from raw pixel inputs, first-person viewpoints, complex strategies and fully autonomous agents. Through this platform, it is easy to set up the foundation for writing a AI for e.g. traversing a map or master the game of laser-tag.

2.3 OPENAI GYM

Similar to DeepMind Lab described in Section 2.2, OpenAI Gym [19] aims to create a platform for AI researchers to cultivate their idea, and enable them to compare their results with the results of other projects working on the same environment. However, while DeepMind Lab solely used images as input to the agent, OpenAI Gym gives an input that is specific to the individual environment. OpenAI Gym can provide images, but it can provide other inputs as well, like joint-angles and velocities for a robot, or a board state for e.g. a board game.

2.4 FRAMEWORKS AND TECHNOLOGIES

2.4.1 TensorFlow

TensorFlow [21] is a framework originally developed by researchers and engineers working at the Google Brain team. Now, TensorFlow is open-source, freely available to everyone. TensorFlow has bindings for other languages, but it is primarily intended for Python development, and allows for constructing program “graphs” which can be executed in an optimized and much faster C-backend than what is possible in Python. It differs from a project like NumPy, in that it avoids the overhead of transmitting data to and from C more than necessary.

The TensorFlow framework is a general framework, which doesn’t necessarily need to be used for ML, but it is designed with ML in mind
and does it really well. With TensorFlow, it is easy to run code on a GPU, for newer GPUs, and even distribute it across several computers.
The subject of RL has certainly seen a lot of innovation in the last two decades. DeepMind produces a lot of new material, and the DQNs developed for the Atari games, described in Section 2.1, has been the foundation for most newer RL-AI initiatives. DeepMind has shown that, an AI taught using RL can learn to play Atari games in such a degree that it rivals the skills of human beings, without being provided with superior knowledge or possibilities. They even found that solutions where the AI is provided with the same information as the player, seems to work better than a solution with hand-picked information. AlphaGO [1] and TD-Gammon [22], are both examples that show RL-AIs can beat even the best of humans, in the games of GO and Backgammon respectively, by simply learning the game.

As the field matures, flourishing frameworks like TensorFlow (as described in Section 2.4.1) continue to ease the process of writing ML algorithms. The tools for developing RL implementations is evolving rapidly, and simultaneously the tools for developing the environments for the RL agents, are improving as well. DeepMind and OpenAI both strive to become the standard for RL efforts, they have both made it easy to pick up an environment and explore how to develop an AI for these environments.

Teaching AIs how to act in a three-dimensional world, such as the ones found in video games, is especially interesting since one might draw a parallel between learning how to move around in a simulated world and moving around in the real world. Furthermore, video-games makes it easy to visualize and measure the performance of the AI. The RL methods have yet to be used in a successful commercial game however, and the, current, practical uses of ML in games is fairly unexplored.

One of the usual approaches when it comes to developing RL-AI for video games, is to provide the AI with the same information as one would provide a human player. Thus the AI and the human compete on equal terms. When looking at initiatives like AlphaGO [1], which has successfully beaten even the best human players, the over-all goal seems to be creating an AI that can beat the human at a game. If the AI is better than any human at a game, then we deem it must be a “good” AI. Most initiatives try to create the perfect AI and provide it with the best information possible. However, if one would want to integrate RL into the video game AIs of today, then perfection is not the goal. An AI for an actual video game should be challenging, but for the most part it shouldn’t be invincible. Furthermore, an AI for
the games of today might not be able to receive the same view as the human player as rendering the screen and handling the resulting image is expensive, relative to the hardware we have presently.

Based on these findings, we will look at the less explored angle to RL, where the aim is to replace the AI in a video game and not the human. I.e. we are not looking to create a super-human AI given all possible information. Instead we will look at the results when we give it some hand-picked information, which is easily and quickly obtainable from the environment.
During this thesis, we have taken a very broad look at RL and a look at the practicality of RL to create AIs for video-games. We have investigated a specific example, in which we have developed a game and used the latest research, within the field of ML, to develop an AI for this game. In order to provide an overview of all the different parts of this thesis, and how they relate to each other, we can take a look at Figure 4.1. In this figure, different parts of the thesis are connected by edges, which illustrate the relationships between the different parts of the thesis. An edge going from part A to part B, means that A was used in the part of the thesis labelled B.

The first thing we did was to develop a game, which is described in Part i. We decided to develop our own game, as opposed to using OpenAI Gym [19] or DeepMind Lab [2] for two reasons. We wanted to have the perspective of a game developer, seeking to utilize RL in our game, rather than simply trying another combination of techniques on an established environment. Furthermore, we wanted to hand-pick and reason about the information which we provide to the AI, rather than relying on what the third party environment would provide us with.

Once we had developed the game, we wanted to have a selection of “black-box” AIs, which could be used for both training purposes as well as evaluating the performance of our AI. In order to obtain this selection, we arranged a hackathon\(^1\) for our fellow students at Aarhus University. The hackathon was a one-day event where groups of students worked together, in order to produce an AI which would compete in a tournament by the end of the day. The hackathon is described in further detail in Chapter 7.

After we had a game, and a selection of black-box AIs, we began the process of training an AI. The first thing we did, was to pitch two AIs against each other, in order to create a collection of “log files” which are recordings of what happened during the game and what the AIs observed during the game. Later, we used these log files to train a baseline RL-AI, such that it would be able to initially learn by observing the AI that were hand-made by humans. However, before we could begin training on these log files, we needed to create a NN, as explained in Part ii. When we had the NN architecture, and the baseline RL-AI trained using the backpropagation algorithm on the log files, we employed two techniques in order to further train the RL-AI.

\(^1\) An intensive coding event, with the purpose of working on some software.
Evolutionary learning is the first technique we utilized to improve the RL-AI. While the gradient descent algorithms utilized in backpropagation seeks to find the nearest local minimum, evolutionary learning searches broadly by making random mutations and picking the mutations that led to the best results.

Intuitively it makes sense to search broadly initially, in order to avoid ending up in a local minimum straight away. However, after a broad search we wanted to take the resulting mutation and go for depth again. Therefore, we would go back to backpropagation, after the evolutionary training, using the RL-AI directly instead of relying on the log files.

In the end, we evaluated the resulting RL-AI against the selection of black-box AIs, through a tournament, in order to have a measure of the performance of our RL-AI.
Part I

THE GAME

Before we began looking at the Reinforcement Learning (RL) aspects of this work, we had to develop a platform that we could train for.

In the following section, we describe the game that we developed. We had a group of students develop a collection of AI’s for the game, which would be used in the RL part of our thesis.
In this chapter, we will give our reasoning behind some of the choices that we have made during development of the game and how these relate to the Reinforcement Learning (RL) environment. In Section 5.1, we will describe some of the game-play mechanics which make up the game. I.e. the actions a player can take, and the information the player is provided with. Afterwards, we will describe the layout of the game in Section 5.2, by looking at the initialization of the game’s environment, the placement of the players within the world and the GUI provided in the game.

5.1 THE GAME MECHANICS

We named the top-down shooter that we developed #Shooter. In the game, two players are matched against each other inside an arena, where they will try to find and defeat the opposing player before the timer runs out. This arena is surrounded with walls, and littered with obstacles that the players must navigate through.

We decided to create a regular game, independent from the thought of Machine Learning (ML), because we wanted to work from the perspective of a game-developer, where the choice of Artificial Intelligence (AI) seldom influences the design of the game in any major way.

5.1.1 Actions

Deciding the actions that were available to the AI, we wanted to take the RL paradigm into account. Our primary goal was to have a simple and small set of actions, both because it narrows down the choices the AI has to make, and in order to simplify the game. To this end, we have created a set of actions that consists of:

- Movement in four directions.
- Rotation (left and right).
- Shooting.
- Preparation.
- A null action, which does nothing.

Most of these actions are quite straightforward, however the preparation action might need some explanation.
The prepare action saves exactly one action for the next turn, so the player will be able to make two actions next turn, note however that it is always exactly one action, so two preparations in one turn will only grant one extra action the next turn. Thus the prepare action brings a lot of tactical choices on the table, due to the fact that we do not supply the actual vision of the player (explained in Section 5.1.2), it can be used to both enhance the knowledge of the player or confuse the opposing player. However, we are also intrigued by what a RL-AI might do with this action. It will be interesting to see whether it will pick up on some of the more advanced strategies it brings or primarily avoid it.

5.1.2 Feature Vector

The second thing we needed to design, in order for the AI to act inside the game, was the nature of the information that we wanted to provide the AI with. Similar to the actions, we wanted to minimize the amount of information, in order to decrease the size of the problem the AI had to solve. Mnih et al. [16] noted that there is reason to believe that hand-picking information, in general, performs worse than simply presenting the AI with as much information as possible. However, we don’t have that luxury, as running-time is crucial to our application; there is some information, such as pixel-data from the Point Of View (POV) of the AI’s agent, which is simply too costly to obtain for our case. Furthermore, some information is not costly to obtain, but provides omniscient knowledge. We wanted to avoid providing the AI with information that would provide it with a benefit over the opponent (such as the exact position of the opponent). Therefore, we decided to design a vector of features that we deem useful, while not providing an omniscient knowledge.

In Table 5.1 we give a short description of what features the final feature-vector contains. However, some of the features are a little more complex, so we will give explain these features in greater detail.

Damage probability

Sight is a tricky thing to implement, give too little information and you have an AI with a lack of information about the world. Give too much and the AI could become omniscient, having an unfair advantage. Damage probability was our attempt at giving the AI some sort of vision that is limited and imperfect, while still meaningful. If we had given the AI the knowledge of the opponent’s exact position, then it would be able to compute the exact sequence of actions necessary to aim directly at the opponent, which would be undesirable.

Our approach requires the AIs to make a decision whether it should shoot immediately, or seek to improve its chances of hitting the opponent by e.g. adjusting the aim through rotation. One limitation of
this probability, is that it does not give any sense of direction, so the AI has to deduce to what side of its Line Of Sight (LOS) the opponent is located. This deduction can be done by turning left or right, and see if the damage probability increases or decreases.

Thus, the damage probability becomes an incomplete, but significant amount of information, which the AI can use in order to spot opponents within its Field Of View (FOV). An important detail of the damage probability, is that the AI cannot look through walls; if there is a wall between two AIs, the damage probability is set to 0, preventing them from shooting around corners.

The exact calculation of the damage probability is done as follows. Given the position \((X,Y)\) and Z-rotation \((\theta)\) of the player and the position \((U,V)\) for the opponent. We create the LOS from two points, note that the vector \((0,1)\) is the point-of-origin for the rotation.

\[
\text{point}_1 = (X,Y) \quad (5.1)
\]
\[
\text{point}_2 = (\sin(\theta), \cos(\theta)) \quad (5.2)
\]
\[
\text{LOS} = \text{line}(\text{point}_1, \text{point}_2) \quad (5.3)
\]

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change in Rotation</td>
<td>Change in rotation with respect to the Y-axis clockwise</td>
</tr>
<tr>
<td>Change in Position</td>
<td>Change in ((X,Y)) Coordinates</td>
</tr>
<tr>
<td>Current Velocity</td>
<td>Current velocity of the player</td>
</tr>
<tr>
<td>Damage probability</td>
<td>0 (\rightarrow) 1 probability of damaging the other player</td>
</tr>
<tr>
<td>Change in Damage probability</td>
<td>Change in damage probability since last tick</td>
</tr>
<tr>
<td>Distance to obstacle</td>
<td>Distance to the first obstacle in front of the players left and right boundaries</td>
</tr>
<tr>
<td>Health</td>
<td>Current health of the player</td>
</tr>
<tr>
<td>Opponent health</td>
<td>The current health of the Opponent</td>
</tr>
<tr>
<td>Time elapsed</td>
<td>Number of ticks since game start</td>
</tr>
<tr>
<td>Time since observed opponent</td>
<td>Number of ticks since Opponent was within field og view</td>
</tr>
<tr>
<td>Time since damage</td>
<td>Number of ticks since damage was taken</td>
</tr>
<tr>
<td>Shoot delay</td>
<td>number of ticks until you can shoot</td>
</tr>
</tbody>
</table>

Table 5.1: Descriptions of feature-vector Content
Then we calculate the smallest distance from the opponent to the \textit{LOS}. Given this distance, we calculate the damage probability by feeding it to a normal distribution ($\mu = 0, \sigma = 5$) that is scaled to 100\% at the median, such that if the player aims directly at the opponent then it has 100\% chance to damage the opponent.

\textit{Distance to Obstacle}

In order to allow the player to be able to move around, without bumping into obstacles constantly, we have supplied the player with the distance to obstacles directly in front of it. This information is split up into two distances, one corresponding to the distance from the right side of the player and one for the left side of the player. This will allow the player to determine whether it can move forward, even through narrow corridors, and will allow the player some sense of depth to its view. This “distance-to-obstacle” feature resembles the damage probability feature in the sense that, it serves to provide some notion of vision, of the obstacles, to the player without providing the player with omniscient knowledge.

5.2 Game Elements

Having described everything, with regard to the actions of the AI and how it selects these actions, we will now dive into the design of the environment in which the AI will interact, as well as the user interface provided for the viewer.

\textit{Game Map}

The game map is a square $X_{\text{size}} \times X_{\text{size}}$ grid, centered at $(0, 0)$ with walls surrounding it. We restrict the size of the game map, in order to reduce the amount of games that time-out and to make the games more intense, without making the map so small that they find each other straight away all the time. Internally, the map is represented by a two-dimensional array containing boolean values of whether that cell is occupied by an obstacle or not. To be able to easily convert from game coordinates to the coordinates of the internal array, we decided to always make the map size uneven such that we have a middle point in the array at $(\frac{X_{\text{size}}}{2}, \frac{X_{\text{size}}}{2})$, which corresponds to the center point $(0, 0)$ in game coordinates.

Given these specifications for the game map, we have chosen to add random obstacles on the map, such that every game has a very high probability of being different. These obstacles also serve to force the AI to learn the game generally, rather than just learning how to play a single map. Originally we just placed the obstacles throughout the map, but during the hackathon we observed there was a need
for more “fairness” during the matches between two AIs. This need prompted us to place the random obstacles onto the map, such that the right side is a mirror of the left side. In order to make the game as fair as possible for both players, and avoid giving one player an advantage over the other.

**Player Spawn**

After the creation of the map, and all of the obstacles that need to be inserted into the map, we have to spawn players inside the map. It is done by spawning the two players in mirroring top-right and bottom-left positions, in order to ensure the fairness mentioned in Section 5.2. If the initial spawn position is occupied by an obstacle, then we spiral out from the player until we find the cell closest to the original spawn point that is free. Due to a bug we first observed later on, we will spiral the same way, when looking for a free cell for both players. This can cause the players to be placed in places that are not mirrored properly, and this can cause some asymmetry. However, this is a rare issue and we estimate that it has minimal impact on our results, as the probability for the actual displacement of the player is somewhat small, and even if it happens, the advantage that one of the players get is that he has been moved slightly closer to the middle of the map.

**GUI**

We did not need a GUI when we trained the AIs, as we used a command-line interface. However, we wanted to have an interface for observing the AIs in action, as well as enticing people to participate in the hackathon (Chapter 7).

The GUI, shows the game itself, with the map and the players as well as their vision-frustum. Whenever a player is hit, there will be shown a small particle-effect at the respective player. Furthermore, we show the health and damage probabilities of the players on the screen with two-dimensional visual indicators. It can be seen in Figure 1.1.
In this chapter, we will give an explanation of how we implemented the design that we described in Chapter 5, as well as some of the tricks that we used to be able to experiment with the game more easily.

Due to earlier experiences, we decided to use the MIT-licensed game engine\footnote{A game engine is simply a foundation for building a game.} Torque3D \cite{torque}. There was no technical reasons for using this specific game engine, except for the ease of access and liberal license.

We based our implementation on a clean copy of the Torque3D game engine, however the engine only supports C++ and TorqueScript as programming languages. We did not want to require the hackathon attendants to learn a new and proprietary programming language in order to code their AIs. Therefore, we deemed it necessary to have a Java-like language, in order to successfully convince people to contribute to the hackathon. We implemented an unfinished interoperability project \cite{interop}, which lets Torque3D communicate with a scripting layer written in C#. The project was far from working, and we had to spend some time working on the back-end of the interoperability layer to make it stable. Generally, interoperability between managed and unmanaged languages is pretty sensitive, so we ran into a lot of corrupted memory and similar issues. We are, however, confident that we got it patched up to a state where any unsolved hidden errors would not impact our game.

In order for our RL-AI to be able to interact with the game, we had to make a communication channel between the learning algorithm and the game. In order to make it a simple plug-and-play solution, we decided to use the ZeroMQ message system \cite{zeromq} to send and receive messages between the game and the learner. Furthermore, we enabled a “render-free” mode, where the game would skip the rendering of the game and just work as a command-line interface. With the rendering turned off, we were able to turn up the speed of the game, such that it could run somewhere between 10 and 100 times as fast for training purposes.

The UI was written in TorqueScript and we wrote most of the game-logic in C#, building upon an existing TorqueScript template which had been partially rewritten in C#.

To make it easier for the hackathon attendees to program their AIs, we created a simple AI interface where the attendees only had to implement a function that returned one of the nine actions. They would
have to decide on which of the actions to return by using a “feature vector” supplied as an argument. This feature vector is the same as the one supplied to the RL-AI such that no one has an advantage. The programmers, at the hackathon, were allowed to do anything, in order to decide the actions to take in an attempt to win the game. The only exception, was that they were not allowed to directly manipulate the environment, except by telling their AI in the environment to take various actions. It would be unfair, if they used any of the globally exposed interface to e.g. set the health of the enemy to 1.
In this chapter, we describe the hackathon that we arranged, in order to collect adversaries for our RL-AIs to battle against and train with. We also describe and evaluate the tournament between the AIs created during the hackathon.

Having developed the game, which would provide the domain for our RL-AI, we wanted some adversaries we could train against, rather than starting from scratch for the RL part of our work. Since we don’t have the time to use human players for thousands of games, nor have we defined how a human player interacts with the environment, we decided that the adversaries should be some black-box AIs, with access to the same knowledge that a RL-AI would have. In order to achieve this, we arranged a hackathon. This chapter describes the event, in order to give an idea of the environment in which our adversaries was developed in and the conditions the participants, developing the adversary AIs, was under.

The hackathon was a single day event, which began at 12:00 on Saturday the 25th of February. We invited fellow students from the Computer Science education at Aarhus University, which had limited experience with both AI-development and the programming tools we used. 16 people showed up for the event, forming five groups in total. From 12:00 – 13:00 we introduced the game and explained how they could develop their AIs. Following that, they were given five hours (13:00 – 18:00) to develop their AIs. After dinner, we held a two-stage tournament to rank the AIs internally. All five groups, managed to make and submit an AI for the tournament. In the first stage, we did a best-of-one round-robin tournament, where each participant was pitched against each other one by one, until each combatant had fought one round against each other.

<table>
<thead>
<tr>
<th>NO.</th>
<th>AI NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BAI</td>
</tr>
<tr>
<td>2</td>
<td>Turing</td>
</tr>
<tr>
<td>3</td>
<td>ScottSteiner</td>
</tr>
<tr>
<td>4</td>
<td>FriskPige</td>
</tr>
<tr>
<td>5</td>
<td>FreksenThink</td>
</tr>
</tbody>
</table>

Table 7.1: Standings from the tournament.

1 An event where a group of people write a piece of code in a limited time.
The second stage of the tournament was a double-elimination \(^2\) round, where the results from the previous stage was used to seed \(^3\) the participants. The matches in the double-elimination bracket was run as best-of-five matches, except for the finals which was best-of-seven.

The resulting placements, can be seen in Table 7.1. The results from each individual match, in the round-robin stage and the double-elimination, stage can be seen in more detail in Appendix A.

### 7.1 Evaluation of the Tournament

During the final stage of the tournament, we found that there was an abnormal rate of wins for the AI which was given the “Player1” position. In fact, “Player1” only lost a single game in all the rounds excluding the finals. Therefore, we decided that, during the finals, the two combatants would have three rounds as “Player1” each and then a tie-breaker round. This seeming advantage for “Player1” could pose a potential problem in our next endeavours, so we decided that before we proceeded with training and evaluating our own AI, we would have to investigate this discrepancy.

Furthermore, there was various bugs in the game that were found during and after the hackathon event. For example, the “prepare” action would make the “shoot delay” count down twice the next tick, meaning players routinely using the “prepare” action could fire twice as often. This seeds doubt about whether the placements during the tournament at the hackathon are actually representative of how the bots perform against each other.

### 7.2 Continuing Efforts

The issues found during the hackathon, raised both the need to re-evaluate the adversaries internally, whenever we fix an error, and the need for new AIs that are developed to fit the newest version of the game, in order for us to justify comparing our RL-AI to these adversaries. Therefore, we created a community \(^4\) around the game, where the users could submit bug-reports and we could rank the AIs on a regular basis. This would serve to both grant us with better AI adversaries and improve the ranking of the adversaries.

We had two additional tournaments after the hackathon. During those two tournaments, there were several bug-fixes as well as changes to the rules of the game. Most notably was the symmetry of the game-

---

\(^2\) In double-elimination, the winners from each battle advance to the next round, where they will battle the other winners and the losers will move down to the lower-bracket or be removed from the tournament.

\(^3\) Seeding, in tournaments, means deciding what the initial placements in the tournament is.

\(^4\) Facilitated through Reddit at [https://www.reddit.com/r/SharpShooter/](https://www.reddit.com/r/SharpShooter/)
map, which we described in Section 5.2. Furthermore, during these tournaments, three new AIs were added: HundenBider, ScannerBot and Kurt. The new placements of the AIs, following the updates and the additions of new adversaries, can be seen in Table 7.2.

<table>
<thead>
<tr>
<th>NO.</th>
<th>AI NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ScannerBot</td>
</tr>
<tr>
<td>2</td>
<td>Kurt</td>
</tr>
<tr>
<td>3</td>
<td>HundenBider</td>
</tr>
<tr>
<td>4</td>
<td>Turing</td>
</tr>
<tr>
<td>5</td>
<td>BAI</td>
</tr>
<tr>
<td>6</td>
<td>ScottSteiner</td>
</tr>
<tr>
<td>7</td>
<td>FriskPige</td>
</tr>
<tr>
<td>8</td>
<td>FreksenThink</td>
</tr>
</tbody>
</table>

Table 7.2: Standings of the adversaries following the ongoing efforts.
In this chapter, we want to describe our experience with the game that we implemented. We will describe some of the difficulties that we have encountered and describe some suggestions for how one could improve the game, in order to accommodate some of its shortcomings.

8.1 Feature Vector

We chose a rather small feature vector, with just 15 pieces of information. The issue with small hand-picked features are that it might affect the possibilities for learning negatively, as described by Mnih et al. [16]. However, increasing the dimensionality of the feature vector, e.g. by extracting pixel-information such as it is done by Mnih et al., would introduce a lot of issues that makes it infeasible for our thesis. For one, it would negatively impact the performance, as extracting this information is expensive. Furthermore it would be a bigger challenge for the hand-written AIs to parse the information they are given, as the information is less explicit. The latter issue, could be solved by feeding the RL-AI and the hand-written AIs two different feature-vectors, however this would impact the equal footing that they would otherwise have.

As for the former issue, i.e. extracting pixel information being too costly, it could have made sense to provide the AI with a sort of low-density vision. Rather than rendering an image of some size, we could have extracted the information from some number of rays, making up the vision of the AI. Then rather than having a full window of vision, it would receive a small and manageable amount of samples, which could be extracted without significantly hurting performance. Extracting samples of the vision in this way, is similar to what Harvey is doing in his article [7], where he has a grid of “sensors” which provide information to the AI. However, we decided against this, in order to simplify the feature vector for the hand-written AIs.

8.2 AI Vision

AI-vision, is a whole field of research in itself, even outside of RL-contexts. Providing AIs with a sense of vision, without making them have an unfair, one might say inhuman, advantage. For example, imagine a shooter game like Counter Strike 1, if the AI inside that

1 http://counter-strike.net
game was able to see, and shoot the enemy as soon as just a single pixel shows, not even having to react and interpret the visuals it sees, it would be easy to write an AI that could beat any human player. It is important that the AI does not have this unfair advantage over the enemy but, as far as possible, it has to work under the same, or similar, conditions.

We did not have the luxury of time to investigate good solutions to this issue, instead we opted for the damage-probability and distance-to-obstacle features to impose such limitations on the AI. The damage-probability sought to impact the precision of the AI, forcing it to evaluate its aim and move to improve it, rather than being able to exactly calculate the actions needed to improve it. The distance-to-obstacle feature sought to limit its vision of the world, albeit a bigger limitation than the one imposed on humans which would be able to see all obstacles in their view-frustum. The distance-to-obstacle feature might very well be too limited to beat a human, but we were satisfied nonetheless as our primary goal in this thesis is not to beat humans, but rather to investigate the feasibility of RL when imposed with such a limited feature vector.

One solution that might have improved the notion of AI-vision significantly, have been the low-resolution sample extraction as described in Section 8.1. This extraction could have been a low-resolution approximation of the vision a human player would have been given.

8.3 Extensions

Mechanically, the game favours the player who gets the first shot in, since a straight-up fire-fight will usually cause the two players to lose health at the same rate. There are few come-back mechanics 2, we would have liked to improve the game by introducing power-ups, such as health-packs which would restore the player’s health. However, this would both take time to implement and increase the complexity of the game, so we decided against it.

It would also be interesting to have more players in the arena at any given time, as it would introduce another layer of decision-making. Rather than making a decision on whether to shoot or not to shoot, it would also be able to decide whether it should shoot this particular player, or find a weaker and more easily defeated prey.

In the end, we decided against all these extra features, in order to keep the game as simple as possible. Heavy time-constraints forced us to focus on working on the simplest case we could find.

2 Features that allow the losing player to turn the game around.
Now that we have described the overall structure of the game and acquired some opponents from the hackathon it is time to start the training session.

In this part we will describe how we will go about training an Artificial Intelligence (AI) that will be able to challenge the opponents from the hackathon. The training will be done through: Log Training, where our AI watches two of its opponents play against each other, Evolution training where our AI will adapt to the opponent based on random evolution and last but not least a deep learning session where a neural network will learn the details of the game.

Multiple of these reinforcement techniques, will be used to train a number of AIs. Whese AIs will participate in a final tournament, which will measure the performance of the AIs and indicate which of the reinforcement techniques is best, in the context of our game.
In this chapter, we will describe the choices made in the design of our Neural Network (NN) in Section 9.2, followed by a description of the training technique we used in Equation 9.2.2. We will discuss some possible extensions for our NN and training process in Section 9.3. Finally, in Section 9.4 we will outline and explain the different hyperparameters that need to be set before our network can be put into use.

9.1 Choice of Machine Learning Technique

Which training method one might use, when trying to train Artificial Intelligences (AIs), varies greatly depending on the Reinforcement Learning (RL) problem that one are attempting to solve. E.g. if you want to solve the Tic-Tac-Toe problem, i.e. creating an AI that plays Tic-Tac-Toe, then it might be overkill to create a Deep Recurrent Network (DRN) since the tic-tac-toe problem has a state space of limited size. A simpler and more effective way to handle that problem, would be to use a tabular method that has a lower overhead and much better convergence guarantees.

Our problem is much more difficult however. Given that the state space of our game is almost infinitely large, and that the computational power is limited, then computation of the exact Q-values for each possible state is not at all feasible. So instead we must turn to approximating the Q-value.

Approximating the Q-value can be done in a number of ways, but most of the current research [8, 10, 16, 26] is focused on NNs that approximate the Q-value using more or less complicated and novel techniques. In order to draw on the experience from this research, we have also chosen NNs as our approximator.

9.2 Architecture

The architecture of NNs can be quite a hassle, since there has not been a discovery of a one-size-fits-all architecture to use. In fact, given some specific problem, we can’t even tell what method or architecture would yield the best results for that problem. It is a mess of heuristics and empirical data for different kinds of problems. Therefore, we have looked at what various current researchers did, for problems slightly similar to ours and used some of the same network parameters that they found was useful.
We decided for a Dueling Double-Q Recurrent Network (DDQRN) architecture, illustrated in Figure 9.1, we will explain each part of this architecture in this section. Section 9.2.1 explains the recurrent-layer in our network, the forward-layer is a feed-forward layer with the details explained in Chapter 10. The output layer is built upon the double-Q and dueling-Q techniques, explained in Section 9.2.3 and Section 9.2.4 respectively.

![Figure 9.1: An overview of the network architecture](image)

### 9.2.1 Recurrent Network

The AI does not have access to a full stack of everything it has ever seen, nor is it omniscient. It can very well be that the AI needs to make sense of more than it can see in the moment, and also look at what it has seen so far. Recurrent Neural Networks (RNNs) achieve this, by letting the output of a hidden-layer in the NN act as input to that same layer, as seen in Figure 9.2. In order for backpropagation to work for a RNN, one would have to employ a work-around. Simply, unroll the recurrent parts of the network, into c copies of those parts, corresponding to time-steps $t, t + 1, \ldots, t + c$, as seen in Figure 9.3. Each copy $c_i$ takes the output of copy $c_{i-1}$ as input, except $c_0$ which receives some initial state.

![Figure 9.2: A RNN, figure from [18]](image)

The unrolling of RNNs is necessary, because the backpropagation algorithm used to train NNs doesn’t work with a graph that has cycles, as described in Section 1.2.
A simple RNN architecture does not solve all memory issues, it can handle short connections, so-called “short-term memory” (an illustration can be seen in Figure 9.4). However, as the distance between the inputs increases, it has difficulties retaining the important information, as it is simply lost over time. It is theoretically possible to learn longer connections, but in practice these architectures fail to learn connections, when they become too long. We call the longer connections “long-term memory”, and the issue that arises when the network needs to retain information, for an extended period of time, is illustrated in Figure 9.5.

Hochreiter and Schmidhuber [10] aimed to solve this issue by introducing a specific structure for RNN cells, called Long Short-Term Memory (LSTM). The LSTM cells aim to solve the aforementioned issue, by trying to learn what it should remember and what to forget, allowing it to retain the important information for an extended period of time. LSTM cells can be constructed in many different ways, however they are all based on the same basic structure. While a basic RNN cell, simply has a single hidden layer, as seen in Figure 9.6, a LSTM cell has four interacting layers. The basic LSTM cell can be seen in Figure 9.7, we won’t go into details here, but the idea is that there are three gates, a “forget”, “update” and “output” gate. These simple control-mechanisms helps the NN learn and select information to store for longer periods of time, in spite of the fact that we have increased the complexity of the network.
9.2.2 Deep Q

The state(s)-action (a) value function $Q$, under some policy $\pi$, is given by [8]:

$$Q_\pi(s, a) \equiv E \left[ R_1 + \gamma R_2 + \gamma^2 R_3 + \ldots | S_0 = s, A_0 = a \right] \quad (9.1)$$

Where $R_i$ is the reward at the $i$’th state action pair, and $\gamma$ is the discount factor. If we know this function, then it is easy to derive the optimal policy by choosing the action with the highest Q-value. Unfortunately, it is not feasible to compute the Q-values for all states in our game, since the state space is almost infinite. In order to alleviate this, we can parametrize the Q-function by the parameters of a neural network $Q(s, a; \theta_t)$, as described in Section 1.2. This gives us a Q-function that is an estimate of the correct Q-function and we can update the weights according to a stochastic gradient descent algorithm, as follows [8]:

$$\theta_{t+1} = \theta_t + \alpha \left( Y_t^Q - Q(S_t, A_t; \theta_t) \right) \nabla_{\theta_t} Q (S_t, A_t; \theta_t) \quad (9.2)$$

$$Y_t^Q = R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \theta_t) \quad (9.3)$$

Where $Y_t^Q$ is the target that we push the Q-value towards, and $\alpha$ is the step size parameter that defines how big steps we take towards the target. The network used to estimate the Q-function, is called a deep Q-network, characterized by the fact that it is Q-learning realized through a deep NN.
Deep Q Network (DQN), as described by Mnih et al. [16], is a technique that has proven to be much more stable than Q-learning when combined with deep networks. It works similar to Q-learning, by iteratively moving weights towards the Bellman target (see Section 1.3.2). However, it brings two additions. First of all, DQN uses a technique called Experience Replay, which will be explained later. Secondly, for a DQN, we maintain two networks. A primary network, $Q(s, a; \theta)$, and a target network $Q(s, a; \theta^-)$. For each training-step we will update the $\theta$ parameters for the primary network using the following target value:

$$Y_t^Q = R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \theta^-)$$  \hspace{1cm} (9.4)

In order to update the target network weights $\theta^-$, we will simply update the target network every $\tau$ training-steps, by setting it equal to the primary network $\theta^- \leftarrow \theta$, retaining its value at all other steps. Finally, actions are selected using an $\epsilon$-greedy behaviour with respect to the primary network $Q(s, a; \theta)$.

Using the target Q-network to predict the Q-values, decouples the selection of the actions from the evaluation of the actions. With this decoupling, it is not quite as likely that an overestimation in the selection implies an overestimation in the evaluation.

**Experience Replay**

Experience Replay is a simple method described by Lin [15], which simply stores the experiences the AI has and reuse them for learning. This seeks to reduce correlation between the data on which we train and smoothing out learning, preventing oscillations and unfortunate feedback-loops. Inspired by an example in [16], say we have an AI which can move in any direction. Now, imagine the maximizing action, at some point in time, was to move left, then the AI would mostly experience the left side of the environment, thus the training data available would be dominated by data-points from that side of the environment. If the maximizing action would then switch to moving right, then the AI would now experience data-points that are mostly from the right side of the map. Now, the AI would have learned a lot from the left side of the map which would be much less useful on the right side of the map. This could make the rewards received plummet and the AI would then begin from scratch, trying to learn this new side of the map and regain its former level of rewards, thus slowly beginning to forget the experiences from the left side of the map, in other words it keeps over-fitting to one side of the map. This could create a loop and the parameters could get stuck in a local minimum as it can only ever get as much reward as one of the sides can provide. Utilizing experience replay counter-acts that by providing an averaged distribution over many previous states, allowing it to train
on its experiences from the left side of the map while it experiences the right side of the map, mitigating this issue of over-fitting.

9.2.3 Double Q

In Section 9.2.2, we showed the DQN technique, which decoupled the Q-value of the selection from the Q-value of the evaluation. However, the DQN techniques, greedily selects the highest target value. A modification to this approach, as proposed by Hasselt, Guez, and Silver [8], is to use the action selected by the primary network to predict the target Q-value rather than the maximum target Q-value. Thus, the target value becomes [8]:

$$\gamma_{doubleQ}^t = R_{t+1} + \gamma Q(S_{t+1}, \text{argmax}_a Q(S_{t+1}, a; \theta_t)); \theta'_t)$$  \hspace{1cm} (9.5)

Here, it is important to notice that the roles of $\theta$ and $\theta'$ need to be swapped periodically, because both of them need to be updated.

We have decided to use the double Q technique because it is a fairly simple, yet very useful, extension of the DQN architecture. [8] Ideally, for double Q-learning, we would symmetrically update the target weights $\theta'$ by swapping the $\theta$ parameters with $\theta'$ periodically. However, [8] points out that one could use the same technique to update the target Q parameters, $\theta'$, as described in Section 9.2.2. Periodically setting $\theta' \leftarrow \theta$ will allow us to implement double Q with a small overhead, yet still benefit from the double Q technique. According to Hasselt, Guez, and Silver [8], this method will provide great improvements with little effort.

9.2.4 Dueling Q

Following the many successes of deep networks in RL, Wang, Freitas, and Lanctot [24] introduces a new NN-architecture for model-free RL. They call it the dueling Q network, and the difference between a dueling Q network and a regular DQN, lies in the output layer. Instead of simply outputting the Q-value directly, the output value should be split into two different streams, which learns an Advantage ($A$) function and a Value ($V$) function; the two streams are combined afterwards into a single Q stream. Recall that the Q-function is used to compute the value of taking an action $a$ when in some state $s$. The Value function describes the value of being in some state $s$, the Advantage function describes the value of taking some action $a$ when in state $s$, the Q-function is then simply the sum of these two functions:

$$Q(s, a) = V(s) + A(s, a)$$  \hspace{1cm} (9.6)

While it might seem futile to decompose a function simply to put it back together, this approach does indeed improve the networks ability to learn. The reason for this improvement, lies in the decoupling
of the state-value \( (V) \) function from the action-value function. In the single-stream method, the Q function represents both the state-value and the action-value, thus we learn the state-value for each action individually. With the dueling Q architecture, we will update the state-value function for all actions from this state, thus the network learns the \( V \)-function quicker.

9.3 Extensions

After we had experimented a bit with the network and did not see enough progress in the AI that was produced, we decided that we might need to extend the network with some novel techniques. This was done in an endeavor to enhance the learning, without changing the overall structure of the network.

In the following, we give a detailed description of the extensions that we have chosen to investigate. First, we have Adaptive Computation Time (ACT) where the main idea is to use the recurrent part of the network, in order to adapt the computation time to the parts of the problem which are hard to predict [6]. Secondly, we have “Network Evolution” where we, given a network, use an evolutionary technique to “mutate” the network [5, 25].

9.3.1 Adaptive Computation Time

When we investigated the methods used by DeepMind, we came across ACT, developed by Graves in [6], which is an attempt to enhance RNNs. ACT tries to adapt the computation time of RNNs by using their recurrent structure, by training multiple times on the same input. To see how this works we look at a simple \( \text{RNN} \) consisting of only one recurrent unit, specified by the input weight matrix \( W_i \), the output weight matrix \( W_o \) and the output biases \( b_o \). Together with the state transition function \( S \) the \( \text{RNN} \) can compute the state signal \( s_1, s_2, \ldots, s_t \) and output \( y_1, y_2, \ldots, y_t \) using the input signal \( x_1, x_2, \ldots, x_t \) and iterating the following [6]:

\[
\begin{align*}
  s_t &= S(s_{t-1}, W_i x_t) \\
  y_t &= W_o s_t + b_o
\end{align*}
\]

ACT changes this setup slightly, by allowing a varying number of state transitions in a single time-step. Using this setup, we define the maximum number of updates in one time-step \( N(t) \). Given this we
can define the intermediate state and output signals \( s^1_t, s^2_t, \ldots, s^N(t)_t \) and \( y^1_t, y^2_t, \ldots, y^N(t)_t \) as [6]:

\[
s^n_t = \begin{cases} 
S(s_{t-1}, x^n) & \text{if } n = 1 \\
S(s^{n-1}_t, x^n) & \text{otherwise}
\end{cases} 
\]

\[
y^n_t = W_0 s^n_t + b_0 
\]

where \( x^n_t = x_t + \delta_{n,1} \) is the original input augmented with a binary flag that indicates whether the input step as just been incremented such that the network can differentiate between the two.

To be able to define how many intermediate states is needed a halting unit is supplied at each intermediate calculation to decide whether or not to stop the intermediate computation. The halting unit is defined as [6]:

\[
h^n_t = \sigma(W_h s^n_t + b_h) 
\]

This halting unit can then be used to define the probability of ending the intermediate computation [6].

\[
p^n_t = \begin{cases} 
R(t) & \text{if } n = N(t) \\
h^n_t & \text{otherwise}
\end{cases} 
\]

where

\[
N(t) = \min\{n': \sum_{n=1}^{n'} h^n_t \geq 1 - \epsilon\} 
\]

\[
R(t) = 1 - \sum_{n=1}^{N(t)-1} h^n_t 
\]

and \( \epsilon \) is a small constant such that it is possible to only use one computation. Using \( p^n_t \) we can compute the mean-field value for both the state and output as [6]:

\[
s_t = \sum_{n=1}^{N(t)} s^n_t \cdot p^n_t 
\]

\[
y_t = \sum_{n=1}^{N(t)} y^n_t \cdot p^n_t 
\]

It is not obvious that this is correct but if we assume that the states and outputs are linear then Equation 9.13 is correct and we can now use variable computation time at each time-step.

However if we look closely then it might be the fact that an intermediate computation need too much computation time. This would be detrimental to the algorithm causing it to be quite slow. Graves [6] alleviate this problem by defining the “pondering” signal \( \rho_1, \rho_2, \ldots, \rho_t \) as:

\[
\rho_t = N(t) + R(t) 
\]
and the ponder cost as [6]:

$$\mathcal{P}(x) = \sum_{t=1}^{T} \rho_t$$  \hspace{1cm} (9.15)

using the ponder cost we can encourage the network to keep this parameter low by adding it to the loss function [6]:

$$\mathcal{L}'(x, y) = \mathcal{L}(x, y) + \tau \cdot \mathcal{P}(x)$$  \hspace{1cm} (9.16)

this way the optimizer makes sure that the ponder is not way too high.

### 9.3.2 Network Evolution

Gradient descent algorithms only search the function space in the direction of the gradients [20]. Due to this, the gradient based algorithms, such as the Adam algorithm used in our training algorithm [13], only lead to a local minimum in the vicinity of the starting location. Therefore, we wanted to investigate evolutionary programming techniques. These evolutionary techniques uses randomness, in order to search the function space. To see the details of the implementation, see Section 10.3. We will give a short general description of the evolutionary techniques in this section.

Given a problem instance, the “genome” of an individual, is defined in such a way that it is easy to create random changes, referred to as mutations. The genome could be defined as a binary string representation or as a real numbered vector, since these are good candidates for mutations. [25].

The evolution is done by maintaining a population $P$ of individuals. Given $P$ one would generate some larger temporary population $Q$ by mutating the “genomes” of $P$’s individuals. The individuals of $Q$ is then evaluated to obtain their fitness $^1$. Given the individuals’ fitness, we select individuals based on a tournament selection procedure. A tournament selection procedure matches a few individuals, of the population $Q$ up, against each other to see which of the individuals has the highest fitness. This is done a number of times to get the new population $P'$ [25].

**Weight Evolution** Given this template for evolutionary programming, we will describe how one would be able to evolve a NN. There are numerous approaches, so in order to make sense of them all, a great review of these methods [25] has been written by Yao. He describes three different major methods for NN evolution. The first method is “Weight Evolution”, where only the weights of the network are mutated. This is a quite simple method, because you do not have

---

$^1$ A value that indicate how well the individual is solving the problem
to change the whole structure of the NN, but only change the weights to the mutated weights. To define the fitness of a weight-evolved network, we can simply look at the results the new approximation grants us. Later, we will see that this is not always the case for other methods.

**Architecture Evolution** The second method that Yao [25] describes is “Architecture Evolution”. This method does not change the weights of the network, but instead it tries to change the structure of the network by e.g. adding/removing nodes or layers. Another part of the architecture one might change could be individual activation functions, which could be changed between different techniques, like sigmoid or softmax, when evolving the architecture. Using this method, one has more control over the fitness function, since the function does not have to be continuous, so we can use parameters such as the size of the network or the time complexity when designing fitness functions.

**Evolution of Learning Rules** The last major method, revolves around evolution of learning rules. This method changes the hyper-parameters, learning rule and weight update function. We could use this method to evolve our network’s hyper-parameters, but as Yao writes, [25, p. 1436], it is much more interesting to look at the learning rules. This is, however, quite difficult because we need to have an encoding of the learning rule, and since we are using built-in functions of TensorFlow™ this would force us to implement our own optimizers which would be out-of-scope for this thesis.

Now, having these three different methods of evolving NNs, we have chosen to simply use the “Weight Evolution” method. We found that it was a simple, yet attractive, candidate. Furthermore, the implementation complexity of the other methods, seemed to have an extent that would give ground for their own projects entirely. Even though we only chose to use the simplest and easiest method of evolutionary learning, we do believe that random exploration of the function space is a viable method for learning in and of itself. However, examining all the other techniques would be out-of-scope for this report, as they are much more involved and time-consuming.

9.3.3 **Hybrid Training**

Since the evolutionary approach is random at its core, the results that we get from it have a very small chance to be optimal, as we search in the entire function space. One can picture the search in the function space as a broad search, that finds various functions that are increasingly closer to an optimal value, but is very unlikely to actually find
a global, not even a local, optimum. However, the gradient descent algorithm, that we exchanged with the evolutionary algorithm, has exactly the property that we are searching for. I.e. gradient descent seeks to find the local optimum in a close neighbourhood. We could use the evolutionary algorithm, to find good candidates for the gradient descent algorithm to optimize from, finding the local optimum from that starting point. Yao also suggest this approach, calling it “Hybrid Training” [25, p. 1428].

In hybrid training we get the best of both worlds, because we alternate the techniques by first searching wide, with evolutionary learning, finding the best individual. Then a deep search, using gradient descent to find a local minimum in the vicinity of the best individual. After finding this local minimum, we can then proceed to another evolution and if we assume that we are close to a local minimum, then there is a high probability that a new better individual will have the potential for an even lower local minimum. Using this method we can repeatedly alternate between the two techniques to optimize the search for optimal solutions.

9.4 Hyper-parameters

Even though we have decided on an architecture for our RL-setup, we still have to make a series of design-choices. Many of the algorithms that we use, contain some free variables that need to be assigned. These free variables, that need to be tuned according to the problem, are called “hyper-parameters“. In this section, we will give an overview of the different hyper-parameters and what values we have chosen for them.

9.4.1 Parameters from the theories of Reinforcement Learning

From the theories of RL, we get two hyper-parameters: the discount factor ($\gamma$) from Equation 1.11 and the step size from Equation 1.17. The discount factor should be set to some number between zero and one; it controls how much the future rewards play a role in the value-estimation. Step size (aka. learning rate) should be set to a value that is higher than zero (and usually lower than one), and it controls how much we move the network weights towards the gradients during the gradient descent step.

In the RL paradigm, we have to design some reward function as well, in order to provide the agent with rewards. This reward function could be seen as a hyper-parameter as well, since it is something that we need to design, and provide to the algorithm. The reward function is a very important part of RL, since it is the way we incentivize the agent to achieve our goals for it.
9.4.2 Neural Network parameters

There are hyper-parameters that is tied to the NN as well. First of all, we have the size of the network, i.e. how many nodes we should have in each hidden layer and the amount of such layers. Network size controls the complexity of the function the network is able to learn, and how fast it is able to learn. Larger network sizes makes it possible to learn more complex functions, but the time it takes to learn that function increases significantly as well, so there is a trade-off to be made.

In order to optimize the gradient descent algorithm, we decided to use a variation of the gradient descent technique, called batch gradient descent and the size of these batches is yet another hyper-parameter.

experience replay Our NN has a recurrent layer, which needs a continuous stream of events to be able to learn the connections between different observations. We can extract these streams of events from the experience replay buffer, which we are using during the training process. The length of these streams, we call it the “trace length”, has to be chosen and thus it constitutes a hyper-parameter.

Furthermore, the experience replay buffer is initially empty, in order to get it filled with some experiences to learn from, we need to start learning by taking a number of “pre-train-steps” before we begin the training process. The number of pre-train-steps can define the quality of the initial experience-buffer, and the number of experiences in the experience replay buffer that exist before the training process began. Finally, the experience replay buffer needs a size-limitation, as mentioned in Equation 9.2.2, in order to limit the memory-usage of the buffer.

target network Section 9.2.3 explains the need for a target network, in order to provide a target Q-value. This target network needs to be updated periodically, in order to accommodate changes in the main network. This is done gradually, by moving the target network towards the main network, using an update rate (τ), which defines the rate at which the target-network is moved towards the primary network.

9.4.3 Evolutionary learning

When we use evolutionary learning, instead of gradient descent, as a training method, we need to decide on the number of offspring each of the individuals gets in the evolution phase, as well as the number of games played in order to estimate the fitness (as explained in Section 10.3). A greater number of offspring results in a larger com-
putation time for each generation, but the diversity of generations increases as well which leads to better results.

To conclude this section about the hyper-parameters we show a table of all the parameters and what values we have chosen for them.

<table>
<thead>
<tr>
<th>HYPER PARAMETER</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\gamma$</td>
<td>0.99</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>0.0001</td>
</tr>
<tr>
<td>$\tau$</td>
<td>0.001</td>
</tr>
<tr>
<td>Batch size</td>
<td>100</td>
</tr>
<tr>
<td>Trace length</td>
<td>79</td>
</tr>
<tr>
<td>Network size</td>
<td>See Chapter 10</td>
</tr>
<tr>
<td>Pre-train-steps</td>
<td>200,000</td>
</tr>
<tr>
<td>Number of offspring</td>
<td>5</td>
</tr>
<tr>
<td>Experience buffer size</td>
<td>1000</td>
</tr>
<tr>
<td>Population size</td>
<td>See Section 11.2</td>
</tr>
<tr>
<td>Games to estimate fitness</td>
<td>See Section 11.2</td>
</tr>
</tbody>
</table>

Table 9.1: hyper-parameters and their corresponding values
IMPLEMENTATION

In this chapter, we will describe how we implemented the design described in Chapter 9. In Section 10.1, we explain our choice of technology for implementing our NN and the reinforcement learning algorithms. Section 10.2, explains the choices we made in the implementation of the network itself. We go through the implementation of each layer and specify the size and shape of the network. In Section 10.2.1 we will describe the ACT extension of the network. Finally, we will describe the implementation of the evolutionary learning technique we will utilize during training, in addition to the more common back-propagation techniques, in Section 10.3.

The implementation of our learning algorithm is available at https://github.com/lukaspj/Shooter-Learner. The DDQN is defined in the file “src/simple_ddqnn.py”, the training of the network is implemented in the file “src/ddqnn_trainer.py” and the rest of the files are the different training techniques and how the network communicates with the game.

Because of instabilities in the game, it would break down about every 5000 games. In order to avoid data-loss, we regularly saved the whole state of the network, experience-buffer and the evolutionary population. This would let us pick up where we left off in case anything went wrong.

10.1 PYTHON AND TENSORFLOW

Google™, being one of the major players in the area of RL, provides and supports the TensorFlow™ framework [21]. Therefore, it was a sound option and easy for us to pick up. TensorFlow™ has a very good binding for the Python language, which provides high-level access to the framework while preserving the control and power that TensorFlow™ provides. These are the reasons why we decided to use Python as our programming language when we implemented the learner, rather than continue working in C#.

10.2 IMPLEMENTATION OF THE NETWORK

The implementation of our DDQN, closely follows the figure shown in Figure 9.1. However, instead of receiving a single training sample, the network receives a batch of training samples. The reason for giving the network the full batch of training samples, instead of one at a time, is one of optimization. The GPU has special hardware for matrix
operations, so batching all the samples into one run like this, allows the GPU to treat the batch as one huge matrix rather than several smaller matrices and thus utilize the fast specialized hardware it has.

The first part of our network is the recurrent layer, the purpose of this layer is to facilitate memory and allowing the NN to make decisions in conjunction with past experiences. We utilized standard LSTM-cells for this part of the network. Implementing such a RNN layer is very straight-forward in TensorFlow™, so this does not merit further details. The output of the recurrent layer is fed into the densely-connected feed-forward layer of our network. This layer’s purpose is to take the memorized input and learn what output it should give such that the output layer can find the final Q-values. We use the term “layer” loosely here, since the forward-layer actually consists of two feed-forward hidden-layers, the first being of size 30 and the second being of size 20. According to Heaton [9, p. 159] there are many different rule-of-thumb methods to determine the size of the hidden-layers. Heaton [9] highlights the following:

- The number of hidden neurons should be between the size of the input layer and the size of the output layer.
- The number of hidden neurons should be \( \frac{2}{3} \) the size of the input layer, plus the size of the output layer.
- The number of hidden neurons should be less than twice the size of the input layer.

In the end, it comes down to the following trade-off: increasing the number of hidden neurons allows the network to learn more complex functions, but can drastically increase the time it takes to learn. Based on these observations, we decided for the size of our hidden layers, where the size of the first layer is twice the size of the input layer and the size of the second layer is equal to exactly \( \frac{2}{3} \) the size of the input layer \( (15 \cdot \frac{2}{3} = 10) \) plus the size of the output layer \((10)\).

After our forward-layer, we have the output layer which takes the output from the forward-layer, splits it into the advantage and value streams as described in Section 9.2.4 and combines the two streams into a single Q-value.

Having our Q-value, we can find the “error” value for our network by \((Q_{\text{target}} - Q)^2\), as described in [8], i.e. we want to minimize the difference between the network and the target network. The target network is as described in Section 9.2.3 but instead of using periodical copies of the main network we update the target network every learning step but only update it towards the main network with a factor of \(\tau\). Given the error of the network, we will make another small optimization here. Lample and Chaplot [14] observed that when using experience replay, where batches of experiences are used to train,
the first experiences are no good for training. This is because of the fact that our network has not been able to observe what happened up until that experience. Therefore, the memory stored in the recurrent layer will be incomplete. Lample and Chaplot [14] discarded some of the gradients, in order to mitigate this issue. Since they discard some of the gradients, the network will be able to make a series of observations before it starts to learn. We decided to mask half of the gradients, thus only learning from the last half.

Now having our loss value, we were able to optimize our network using gradient descent. However, rather than using a simple gradient-descent optimizer, we decided to go for the widely used Adam optimizer [13]. The Adam optimizer is an adaptive gradient-descent algorithm that trades a minor increase in computation time and memory usage, with better performance. The major benefit for us, is that it typically requires very little tuning. RL is plagued by long training times, so utilizing an algorithm that minimizes the impact our choice of hyper-parameters have on the outcome is very useful. It gives us the ability to complete experiments and allows us to focus on other experiments than the choice of hyper-parameters.

10.2.1 Adaptive Computation Time

We based the implementation on the code written by Neumann [17], which we upgraded to work in TensorFlow 1.0, with the author’s help. After implementation, we found that the learning process slowed down a lot. One of the issues with RL currently, is the tremendous amount of training data that the NN has to go through. At the same time, one of the interesting things about RL is the availability of almost unlimited data, since you can simply generate new data any time you need. When you are doing supervised learning, and the amount of samples are limited, you want to make the most of the samples you have, as that is the only way to improve training process. However, in our case, we ended up spending a lot of time on individual samples using ACT, in fact 24 hours for just 28 samples, and we decided that we would not be able to pursue this path to the extent that we wanted, because of time-constraints. We decided that we would rather be able to train on a larger quantity of training data, than improve the quality of the training data using this technique. We believe the reason for this abysmal speed, is down to implementation and not the theoretical limitations of the method. In fact, there were many issues with swapping information to and from the GPU’s memory which is a major bottleneck for GPU computations.
10.3 Network Evolution

In Section 9.3.2 we discussed why we chose to use “weight evolution” as a learning mechanism. Here we will give a description of how we have implemented it.

First and foremost, we need to have some network that will act as a host for the whole population. We do this by collecting all the “trainable”\(^2\) variables \(W_i, i = 1 \ldots n\) from the network model in a list. During evolution, we only need to be able to use the network to predict actions based on some weights, as we will be doing weight evolution. Therefore, we will not need to use the “target network”; we only use the target network for the Double-Q learning technique.

According to [25, pp. 1427] we need to have a Gaussian mutator \(\eta_i\) for each of the \(n\) variables. Using a mutator for each of the variables, allows each variable to mutate independently of the other variables. These Gaussian mutators are drawn from a normal distribution with \(\mu = 0\) and \(\sigma^2 = 1\). Individuals in the population can now be defined by the variables and the mutators for those variables as: \((W, \eta)\) and each individual is able to produce randomly generated offspring \((W', \eta')\) from these vectors as follows:

\[
\eta_i' = \eta_i \cdot \exp (\tau' \cdot N(0, 1) + \tau \cdot N_i(0, 1)) \tag{10.1}
\]

\[
W_i' = W_i + \eta_i' \cdot N_i(0, 1) \tag{10.2}
\]

Here \(N(0, 1)\) is the standard normal distribution and \(N_i(0, 1)\) ensures that the random value is recomputed for each \(i\), resulting in different changes to the variables in the network weights \(W_i\). We have adopted the \(\tau, \tau'\) values from [25] setting them to be:

\[
\tau = \left(\sqrt{2 \cdot \sqrt{n}}\right)^{-1} \tag{10.3}
\]

\[
\tau' = \left(\sqrt{2 \cdot n}\right)^{-1} \tag{10.4}
\]

Now, we have defined how to generate a population from an existing network, by mutating the variables randomly using the \(\eta_i\) mutators. We will now describe a technique for thinning out the population. We define a tournament selection procedure, in order to select the best individuals from each generation. Given the initial population size \(M\), the number of offspring \(K\) and the current population size \(N = M \cdot K + M\), we split the population up into \(M\) groups with \(K + 1\) individuals in each group. For each group we allow the individual with the highest fitness to move on to the next generation.

\(2\) All the variables that are updated during the process of learning.
The fitness of an individual is evaluated by running the network, against some AI, a number of times in order to estimate the reward density $r_d$:

$$\text{fitness} = r_d = \frac{\sum_{i=1}^{\text{Games}} r_i}{\text{Games}}$$ (10.5)

Where $r_i$ is the reward obtained in the $i$’th game. In practice, this will equal the win-rate, as the size of the reward for a win is greater than any other rewards by a magnitude, other rewards will only really serve as tie-breakers.

In order to evaluate the fitness, we need to decide the number of games played; it is important to remember that $r_d$ is only an estimate of the actual reward density. We need to decide on a trade-off between the accuracy, and precision, of the estimate and the time it takes to obtain the estimate. Furthermore, we will need to decide the size of each generation population. Larger populations allow for better evolutionary progress, exploring more options; smaller populations allow for quicker iterations through generations but of a lower quality, and less exploring. We have chosen to vary the trade-off on population size and games played dynamically, based on the fitness of the current best individual. We will describe how we determine the trade-off in the experiments section: Section 11.2.
In this chapter, we describe the experiments that we have conducted, in order to evaluate the performance of the network described in Chapter 9.

An overview of the experiments we will look at can be seen in Figure 11.1. First of all, we will look at selecting some of the hyper-parameters described in Section 9.4. In order to make a call on the trade-off between running time and amount of learning per episode, we will look at the running time training on an episode using different batch-sizes.

![Figure 11.1: An overview of the different experiments made in this thesis](image)

After the hyper-parameters have been established, we will look at different training-techniques, which we will use to create different AIs in Section 11.2. The first training technique we look at is the “Three step learning” used in the “Main experiment”, which is the experiment we invested the most time in. During this experiment we combine different techniques and use different adversaries to achieve four different AIs.
The second training technique we used, was the “All in Evo” technique, where we used evolutionary learning solely, over a long time period, to train an AI with BAI as the adversary. We explore this technique in Section 11.3.

During the “Main experiment” we discovered some issues which we investigate in Section 11.4 and Section 11.5. Which corresponds to the two steps at the bottom of Figure 11.1.

Finally, we will take all the resulting AIs from the previous experiments and the adversaries described in Chapter 7, and determine the performance of them by running a tournament and ranking the AIs relative to each other based on the tournament. This final tournament, described in Section 11.6 will conclude our experiments.

To be able to easily reference the AIs from the experiments, we prefixed them with “RL” and named them semantically after how they were trained. RLFreksenThink, RLBAI, RLScottSteiner and RLMix (a mix of FreksenThink, BAI and ScottSteiner), are all named after the adversaries they trained against during the main experiment, described in Section 11.2. RLBAIEvo is the AI that were trained by doing continuous evolutionary learning (see Section 11.3), and RLCopy is the AI that were trained by trying to copy the actions the BAI AI took during a match against Turing (Section 11.5.1). Last, we have the RLPunisher AI which was training with expert-knowledge injected by providing it with punishments in certain cases, as described in Section 11.4. For future reference, we provide Table 11.1, for an overview of the different names and which experiments they refer to.

<table>
<thead>
<tr>
<th>NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>RLPunisher</td>
<td>Three step learning using punishing meta rewards against trained against ScottSteiner AI</td>
</tr>
<tr>
<td>RLBAIEvo</td>
<td>Continuing evolutionary learning against BAI</td>
</tr>
<tr>
<td>RLBAI</td>
<td>Three step learning against BAI AI</td>
</tr>
<tr>
<td>RLScottSteiner</td>
<td>Three step learning against ScottSteiner AI</td>
</tr>
<tr>
<td>RLMix</td>
<td>Three step learning against the Mix of the three AIs</td>
</tr>
<tr>
<td>RLFreksenThink</td>
<td>Three step learning against FreksenThink AI</td>
</tr>
<tr>
<td>RLCopy</td>
<td>Log training with rewards trying to copy the actions of BAI AI</td>
</tr>
</tbody>
</table>

Table 11.1: Naming convention of the different RL-AIs

11.1 Finding the Hyper-parameters

Overall, we based most of our selection of hyper-parameters on what others have done for their projects and rules-of-thumb as was the
11.1 Finding the Hyper-Parameters

However, we did find that the numbers we had used for the “batch size” and “trace length” parameters seemed a bit in the lower end. We first looked at trace length which was set to 8 initially, based on the number from an online tutorial [12]. Having a trace length of 8 is equivalent of training on strings of events that occurred within a period of $\frac{1}{4}$ second. This seemed a bit on the lower end, we wanted the trace length to be at least 20 so the AI would be able to experience the full time from when a shot is fired and until it was able to fire again. We ended up setting the number to 76 which is about 2.3 seconds, in order to make sure the strings of experiences was long enough. It needs to be so long that the AI would be able to learn patterns, but we wanted to avoid increasing the time it would take to train on each string too much.

The next hyper-parameter we wanted to tune was the batch size. The batch size will increase the time for each training step but in return, it will also increase how much we train on the experiences we have encountered. It was originally set to 4; we wanted to increase the batch size, but avoid increasing it to a point where it increases the training time too radically. Therefore, we decided to measure the impact of the batch size on the time it took to perform a training step. The result of this experiment can be seen in Figure 11.2.

Now, the reason that there is a lot of noise in the data, is because we ran each batch size on one episode each and computed how long it took. Episodes have varying length, from 100 experiences and up to 3200 experiences, so some of the training sessions could have a lot of training steps it needed to go through. This varying length was on purpose however, as we want to know how the batch sizes compare...
to each other as well as how these different lengths effect the running time. From eye-measure, it seems like there is some lower-bound and upper-bound on the time it takes to train. We plotted two lines, that seems to approximately follow the lower- and upper bounds. The line that approximates the upper-bound, is defined as $1.6x + 80$ and the lower-bound line is $0.15x + 10$. The upper-bound increases faster than the lower-bound but they are both linear, which means that we don’t risk unmanageable increases in running time on long episodes. We decided that we would set the batch size to 100, which would mean that the upper-bound running time for an episode would be $\sim 200$s but it would be more likely that each episode would take $\sim 40$s on average. This seemed like a good trade-off between increasing the batch size and keeping the training time down.

11.2 THE MAIN EXPERIMENT

We designed a primary method to train our AIs. We wanted to combine all the techniques we have described: log-training for learning from the experts, evolutionary training for a broad search of the function-space and deep learning for locating the best minimum when searching from the result we got from evolutionary training. This technique is what we will call “Three step learning” for future reference. Using these methods, we wanted to examine how our network learns when exposed to different adversaries. We know that the adversaries, we acquired from the hackathon (see Chapter 7), have different skill levels based on a tournament ranking: Table 7.1. Therefore, we want to examine how different skill levels of adversaries affects the overall quality of the developed RL-AI.

The first thing we did, was to create an initial model by collecting 2100 experience logs, from games between the two best adversaries BAI and Turing. We used this initial model as a base-line for all the different AIs we would train. We decided on 2100 logs, as we did not see any remarkable improvement in how the AI performed so it seemed futile to continue. However, we did a manual observation of the base-line AI, and found that it had learned some “states”. I.e. it had learned that it should act differently when it would get shot, and try to react to being shot. Therefore, we concluded that even though the AI did not improve significantly in performance, we suspect training on the logs had an effect as it had learned to react when it was exposed to danger. We explored the effect that initializing the AI with these logs have on the performance in Section 11.2.1.

The log training can be thought of as observing expert players compete, in order to learn how to copy the actions that led to a victory. Intuitively, it seemed like a good idea to utilize log training to get a base-line AI up and running. It does have a few limitations, e.g. the
network is not able to decide on which actions it can take, so it cannot explore how other policies work. It can only explore the policy taken by the expert it observes; log training still seems like a useful technique to get an initialized AI, which we could use for the next steps, i.e. the evolutionary learning step.

We wanted to be able to develop an RL-AI for each of the adversaries obtained from the hackathon. As mentioned earlier, we would base them on an AI that had trained on 2100 logs, and then train them with evolutionary learning where the fitness is measured against the respective adversary, followed by deep-learning with the adversary. However, each AI should preferably be able to train against a couple of thousand games and training is a long rigorous process. Therefore, we decided to scale down the breadth of the experiments a bit so we could focus on making sure we made the experiments properly. We decided we should select three AIs to perform the primary experiment on, and we thought the most interesting results would come if we chose the best, the worst and an average AI so we could compare the results of training on adversaries with different levels of skill. Based on the tournament standings after the hackathon, we decided to choose FreksenThink, ScottSteiner and BAI1 as the adversaries we would use for the primary experiment. Additionally, we chose to run the primary experiment against a mix of the former three AIs, picking a random AI for each episode.

11.2.1 The impact of the log-training on the evolutionary learning

The first step of our three-step process, was the log-training step. Its primary purpose, was to initialize the weights for the evolutionary learning, to allow for a quicker evolutionary learning process, in the next step. The intuition being that, even if it doesn’t learn how to play the game well, it will give it a quick notion of how the game works. Informally, we trained it against the 2100 logs and observed it play; we saw the AI performing poorly, but reacting to getting shot. I.e. it did not learn how it should react to getting shot in order to win, but it did learn that it should react to getting shot.

In order to reassure us that initializing the weights, using the logs, in this way improved the learning process of the evolutionary learning, we made an experiment with evolutionary learning against FreksenThink without initializing the weights using the log training. Due to data-loss, we did not manage to run the experiment for the full 11 generations, only the first 9 generations, however we can clearly see the difference in the progress it made in Figure 11.3.

It starts off at a lower estimated win-rate, of just 5% against the 9% that RLFreksenThink gets, in the first generation. During the evolutionary learning, RLFreksenThink sees a much bigger rise in es-

---

1 FreksenThink, ScottSteiner and BAI are AIs from the hackathon (See Chapter 7)
Figure 11.3: The fitness of RLFreksenThink’s evolutionary learning process, compared to doing evolutionary learning without initializing the weights using log training first. We see that, the log training seems to have a huge impact on the learning ability of the AI, as it performs much better in estimated win-rate.

Estimated win-rate, culminating in a win-rate of 30% in generation 9 against the estimated win-rate of the AI that was not initialized on logs, which lies as just 7% estimated win-rate in the 9th generation.

11.2.2 Designing a reward function

Knowing that we did not have enough time to investigate everything, we had to decide what rewards the AI should receive, we made some decisions based on intuition. First of all, we decided that winning a game should give a reward of 1, this made sense as the only real credits distributed by the game is the “win/loss” signal at the end of each game. However, we feared the reward function might be a bit too binary, so we decided to provide a very small additional reward for reducing the health of the enemy. The hope was that this would help guide the AI towards the win-condition, we decided the reward for reducing the health of the enemy would be \( \frac{1}{100000} \) such that it was vanishing small compared to actually winning. It was crucial to avoid this change in reward to allow a policy which provides as much reward for reliably hitting the enemy, e.g. 10 times each game, as opposed to actually winning the game at a lower frequency. If one such policy exists, it could create a local minimum, where the AI would not care about winning a game but just getting a few shots in every time. While we cannot say for certain that this is an issue in practice, we decided we wanted to avoid such an issue by making sure that a win will almost always overshadow hitting the enemy.
When we moved on from evolutionary learning and into the deep learning, we found that the rewards our AI received was dropping quickly. It seemed to be deteriorating rapidly by the deep learning. In fact it was so abnormally quick (see Figure 11.4) that we had to investigate the issue. We started by looking at the final Q-values the AI observed, i.e. the rewards it predicted during the last training batch. From Figure 11.5, we observed that, its Q-values (which is the rewards it expects to receive) was both very unstable and very wrong, as it did not converge to 0 (which it should if it never wins), so we investigated our code and found that we only updated the target network of the times we intended. Fixing this issue made our network converge to 0 (see Figure 11.6), which makes sense considering that it was still losing and forgetting everything it had learned from the evolutionary learning steps.

![Figure 11.4: Rewards received when we only update $\frac{1}{1000}$ of the times. It quickly flat-lines at 0, seemingly not using the experience it got from the evolutionary learning.](image)

We assumed the network training was done properly for this debugging step, and tried to tweak the reward function in order to fix the issue. The first thing we tried was modifying the number of “pre-train steps”. Modifying the number of pre-train steps, allows the AI to build a bigger experience buffer with episodes it has learnt from the evolution step. However, there was no difference in results if we changed it to 100,000 steps or even 200,000 steps (see Figure 11.7).

We analysed our code over and over, but we were unable to figure out a solution to the issue. In the end, as a “Hail Mary” attempt, we tried simply multiplying the rewards given (which was 1 for a win and 0.00001 for hitting the enemy with a shot) by 100,000, changing it to 100,000 for a win and 1 for hitting the enemy. This drastically
Figure 11.5: Q-values and target Q-values when we only update $\frac{1}{1000}$ of the times. In spite of losing all the time, it does not converge to 0 and seems to be very unstable.

Figure 11.6: The Q-values when we update the target-network after each training step. In contrast to Figure 11.5, it now quickly converges to 0, this does not improve its win-rate however.
changed our result, and the AI would no longer lose all the games as soon as it started the deep learning step, as we will see in Section 11.2.

We do not really know why changing the rewards gave us such a drastic change in the Q-values. If we would have to guess we think that it might have been a floating point issue, which would not affect evolutionary learning as it doesn’t do any training, it simply adds the rewards up and picks the best individuals. However, this would also mean that the initial model we have used for our evolutionary learning steps, has been trained with a reward function with some possible critical errors. We investigate these issues further in Section 11.5.

11.2.3 The results of the primary experiment

In Section 11.2 we described how we executed the first part of our training, by creating a base-line initialization of our AI based on log-training. The next step was the evolutionary learning step, which starts off from the baseline AI produced by the log-training. The evolutionary learning step could potentially run forever, continually looking for a change in the AI that would make some improvement, so we wanted to decide on some set number of generations it would run for. We chose to run the evolutionary learning for the baseline AI with FreksenThink as an adversary, and ran it until we could see diminishing returns. At 11 generations, it seemed to lose its momentum, so we decided that all the AIs should run 11 generations of evolutionary learning.

The populations started of with a size of 10 individuals, and for each generation those individuals would produce 5 offspring each.
This would result in 60 individuals which would run 100 games against their adversaries, to obtain an estimation of their fitness. Running 100 games against the adversary gives us a fairly stable estimation of their win-loss ratio against that adversary. It goes without saying that we could simply play more games between the two, to get a better estimate as there are some variance even with 100 games. It comes down to a trade-off between how many individuals we want to evaluate and how precise the evaluation should be. We wanted to evaluate many different individuals in the beginning, not caring much for small variations in the result, but when we had measured a great quantity of AIs, the quality, and thereby the accuracy of the estimations, would matter more. In other words, the more individuals we have in our population, the more samples of the function space we have and thus the chances that we have found an improving sample increases, but so does the time to compute the fitness for all the samples. Similarly, the more games we use to estimate the fitness, the more precise the estimate is but again the precision comes at a cost of computation time. We decided that we would interpolate the number of individuals in each generation, as well as the number of games played in order to evaluate the fitness. We defined Equation 11.1 to measure these two numbers based on the last observed fitness.

\[
|P| = \left\lfloor 10 \cdot \left( e^{\text{fitness}} + \frac{1}{4} \cdot (1 - 5e) \right) \cdot \left( \frac{-4}{5 \cdot (e - 1)} \right) \right\rfloor 
\]

\[
\#\text{Games} = \left\lfloor 100 \cdot \left( e^{-\text{fitness}} + \frac{1 - 5e}{4e} \right) \cdot \left( \frac{-4e}{e - 1} \right) \right\rfloor 
\]

This is an exponential interpolation, it will interpolate from 10 to 2 for the population size and from 100 to 500 for the number of games played in order to estimate the fitness.

11.2.4 Three step training against different adversaries

In this section, we describe our results from utilizing the hybrid-training technique (See Section 9.3.3). We use the hybrid training to define the three step learning method described in Section 11.2. Using the three step learning method, we want so test how well it fairs when used on the different AIs from the hackathon (See Chapter 7).

Unfortunately, we did not have the luxury of running the experiment, for the different AIs, in parallel, thus we decided to let them run as long as we could as opposed to a set amount of training steps. The result being, that the different AIs did not have equally many "training steps", i.e. updates of the weights in the network. In Table 11.2, we get an overview of the amount of training steps each AI, in this main expirement, were given. RLSteiner were allowed to train twice as much as RLMix and RLBAI, which goes to show the discrepancy between the number of training steps provided for each AI.
Table 11.2: A comparison on the different processing times relative to the longest processing time. We see that the RLSteiner AI has a lot more processing steps than the other and that the RLMix AI has the least of them all.

<table>
<thead>
<tr>
<th>AI</th>
<th>Training Steps</th>
<th>Relative to RLSteiner</th>
</tr>
</thead>
<tbody>
<tr>
<td>RLMix</td>
<td>~2,760,000</td>
<td>39%</td>
</tr>
<tr>
<td>RLBAI</td>
<td>~2,962,000</td>
<td>42%</td>
</tr>
<tr>
<td>RLFreksenThink</td>
<td>~5,869,000</td>
<td>83%</td>
</tr>
<tr>
<td>RLScottSteiner</td>
<td>~7,093,000</td>
<td>100%</td>
</tr>
</tbody>
</table>

Figure 11.8: The fitness during 11 generations of evolutionary training, for different AIs. As one would expect, the easier AIs provide a higher fitness for the learning AI.
The weakest adversary

The first results we look at is the training process with the AI called FreksenThink. First and foremost, we look at the fitness for each generation in the evolutionary learning step as seen in Figure 11.8.

After the evolution step, we can see how we reach a win-rate of 30% at the 9th generation using FreksenThink as an adversary. Keep in mind that at 30% fitness, they only played around 260 games to achieve that fitness for that specific individual, so it’s only an estimate. We observe that the AI consistently improves its fitness over the generations, which strengthens our belief that the AI is learning and that this problem is possible to solve. It is very possible that we could have continued improving the AI using evolutionary learning, but we decided to move on to deep learning at this point. We explore the results of continued evolutionary learning in Section 11.3.

Having the AI that won 30% of the games against FreksenThink, we began the deep-learning process – utilizing back-propagation to update the network. Unfortunately, we immediately saw a drop in win-rate. When we look at Figure 11.9, we see that it has dropped to 10%, which is quite unfortunate. We will discuss this drop, and what could be done to prevent it, in Section 11.4. However, in spite of the drop in win-rate, we can see a slight increase in win-rate over-all, from ~10% to ~15%. This leads us to believe that our AI is also learning and improving during this step of the process, however the learning seems to have stagnated during the last half of learning.

![Figure 11.9: Estimated win-rate for RLFreksenThink. Starting at 10% in spite of the evolutionary learning producing results of > 20% win-rate. It does slightly increase from the 10% to ~15% however.](image)

Earlier, in Section 11.2.2, we decided to look at the final Q-values observed for each episode, when we compared it to the given rewards. In that context, we use the final Q-values as an indicator of the over-all
Q-values for the training batches during each game. In order to boost our confidence in the validity of this decision, we can take a look at Figure 11.10 which shows the average Q-values, observed during each episode compared to the final Q-values. We see that, apart from a constant difference in scale, the two graphs are highly correlated. In retrospect the average-Q values better reflects the progress of the AI, as they are more stable, therefore we decided to use the average Q-values in this section.

![Graph showing average Q-values and final Q-values](image)

Figure 11.10: Average Q-values over each episode compared to the final Q-values observed in each episode, for RLFreksenThink. In spite of a difference in scale, they seem to be highly correlated.

Looking at Figure 11.11, we can see the rewards received and the average Q-values observed for each episode. The Q-values are very unstable in the beginning, but, at around the 6500th episode, they begin to have a steadily increasing pattern. The more interesting trend in this graph, is that there seems to be some correlation between the rewards and the Q-values; when there is an increase in the rewards, then the Q-values seems to be increasing shortly after as well. The reaction in Q-values will not be immediate, because of the experience replay extension.
Figure 11.11: Rewards vs. Q-values. Rewards are 100,000 for a win, therefore there will be many spikes. The Q-values seem to follow the rewards, just slightly delayed. Rewards have a bigger influence on the Q-values in the beginning, since the experience buffer is growing.

The average adversary

ScottSteiner was the chosen adversary which came in around the middle in the standings. We can see the fitness during the evolutionary learning in Figure 11.8.

Against ScottSteiner, we only reach ~5% win-rate at the end of the 11 generations of evolutionary learning. Under-way, it peaked at around 6%, this is significantly less than the results against FreksenThink. Unlike the FreksenThink-AI, which seems to increase in fitness steadily across the 11 generations, the ScottSteiner-AI seems to halt in progress after the 4th generation. This could indicate that it’s easier to train against FreksenThink than ScottSteiner, at least in the very short run.

Similar to what happened with the AI that trained against FreksenThink, the AI that trained against ScottSteiner experienced a drop in win-rate when deep-learning began. As with the previous adversary, we will discuss this drop in Section 11.4. The win-rate (seen in Figure 11.12) dropped from the aforementioned ~5% to ~0.5%, increasing to ~1% over the deep-training period. Interestingly, we observe that the AI seems to increase steadily in estimated win-rate, even though the average Q-values (Figure 11.13) doesn’t seem to increase. Albeit, the Q-values are very noisy, so the increase might just be hidden by the noise. Again, the deep-learning results does not seem promising so far.
Figure 11.12: Estimated win-rate for RLScottSteiner. It slightly increases during the training, but it starts at 0% in spite of the results achieved during the evolutionary learning.

Figure 11.13: Average Q-values for RLScottSteiner.
The best adversary

The most challenging adversary, is the BAI-AI. As the winner of the original tournament from the hackathon, it would certainly prove a formidable opponent for our AI. This level of difficulty certainly shows in the evolutionary fitness, where the fitness against the BAI adversary is the lowest of the three, peaking at \( \sim 3\% \) win-rate and only improving slightly over the 11 generations. We explore whether this slight improvement continues in Section 11.3.

Not surprising, the estimated win-rate of the BAI AI takes a dive from \( \sim 3\% \) to 0% when we make the change from evolutionary learning to deep learning. Meaning, the AI went from winning occasionally, to never winning. In fact, as seen in Figure 11.14, the AI floors completely at an estimated win-rate of 0% all the way. In Figure 11.15 we see that the Q-values don’t change much during the duration of the training process, so our AI seems to be at a complete halt.

![Figure 11.14: Estimated win-rate for RLBAI.](image-url)

Figure 11.14: Estimated win-rate for RLBAI. Completely flat-lines at 0%, which is not too upsetting since BAI is the most challenging adversary, however it is a disappointment compared to the results from the evolutionary learning, in Figure 11.8.
The mix of adversaries

As one might expect, the AI training against the mix of the three chosen AIs achieved a fitness somewhere between the other three AIs, during evolutionary learning. Achieving a peak of \( \sim 18\% \) and ending up at \( \sim 14\% \), this is not surprising when one looks at the final fitness of the other three AIs and take the average of them:

\[
\frac{30 + 5 + 3}{3} \approx 13
\]  

(11.3)

However, it is worth noting that it performs slightly better than the aforementioned average.

When we look at the estimated win-rate of the AI (Figure 11.16), we see the same pattern as with the other AIs: a sharp drop from the evolutionary learning step, down to 6%. For this AI, there is a slight decrease in the estimated win-rate overall, in contrast to the slight increase we saw in the other AIs. However, it is important to keep in mind that this AI has had less time to train than the others, so it might achieve an increase if it were allowed to train for a month or so, as this AI has only trained for 11 days, which is less than half of the time we spent on RLFreksenThink.

Figure 11.17 shows the average Q-values for the AI with mixed adversaries, we can see how the AI fell down a pit, which correlates with the estimated win-rate we see. It comes back to some more reasonable results after the 5000th episode, which indicates that it is becoming more optimistic. Although it seems to be dropping at the end, making it difficult to evaluate the performance of this AI.
Figure 11.16: Estimated win-rate for RLMix. In contrast to the other AIs, this AI deteriorates instead of improving slightly over the course of training. However, it did not have the same amount of time to train in.

Figure 11.17: Average Q-values for each episode, for RLMix. It takes a sharp dive in its expected performance, and doesn’t quite seem to recover and get on a track that indicates continual improvement.
11.3 GOING ALL IN ON EVOLUTIONARY LEARNING

Discouraged by the results we got in our main experiment (Section 11.2), we sought different approaches to train our AI. One of our first ideas was to simply commit to the simple evolutionary learning we used to as the second step in our main experiment. I.e. we wanted to simply skip the deep-learning step and see what would happen if we continued the evolutionary learning. In contrast to our deep-learning results, the evolutionary learning showed a consistent, albeit slow, increase in fitness. Therefore it would make sense to continue down this path as long as we observe this rise in fitness.

We chose to run many generations of evolutionary learning against the BAI adversary. We decided to do it against BAI, because we had rated it as the strongest adversary, and we did not have enough time to experiment with all the AIs, so we decided that if it succeeded against BAI, then it would probably also succeed against the weaker AIs. In Figure 11.18 we can see the results of this experiment. We had time to run it for 93 generations, and we clearly observed that the steady increase in fitness continues with just a few set-backs. It increases all the way up to a win-rate of approximately 17%, which is significantly better than the deep-learning results. However, this improvement did not come cheap, it took more than one and a half months to achieve this result so it is a very slow process. In spite of the few set-backs observed in the data, it does seem like there was still plenty of room for improvement to the fitness. Unfortunately, we were not able to explore just how big this room was and what the limit of this technique is. We expect that its rate of improvement would decrease over time, as the difference between winning and losing a match increasingly depends on finer detailed decisions. E.g. winning the first game is about figuring out that it should shoot, while winning half of the games likely depends on optimizing when to shoot and how to dodge.
11.4 THE DIP IN PERFORMANCE AFTER EVOLUTIONARY LEARNING

Section 11.2 showed a sharp dive in performance, when we began the deep-learning step after the evolutionary learning. We were initially very puzzled, as we had imagined that the performance of the AI, at the very least, should stay the same. When we investigated the issue, however, a problem became very clear. Figure 11.19 shows a histogram through time over the actions selected, by the AI that trained against FreksenThink. There is a large spike at 7, which is the “shoot” action, this clearly indicates that the AI is pressing the “shoot” action trigger almost constantly. Occasionally, the AI tries to mix it up with different actions, but other than that it seems to be over-fitting towards shooting. It does make some sense, considering that it only gets rewards if it shoots and defeats the enemy so the immediate logical action would be to always shoot. However, there are many situations where it never makes sense to shoot, e.g. if the enemy is not in sight or if the “shoot delay”-feature is not currently 0, as it would not accomplish anything rather than waste time.

Based on these observations, we decided to construct a new experiment, where we added some punishments for taking actions that we, as the experts, deem to be futile. We decided that taking the “none”-action was always a bad idea, as well as shooting while the “shoot delay”-feature was not 0 and taking a “prepare”-action twice in the same tick (following a “prepare”-action in order to take two actions in one tick). We gave the AI a reward of −1 as punishment for making any of these mistakes. Additionally, we set the reward for hitting the
enemy to 100, as to offset the reward for shooting from the punishment for making a mistake.

Due to limited time-constraints, we chose to perform this experiment against ScottSteiner. FreksenThink often has very long games, as it is not very efficient, at finding and eliminating its opponent; BAI is a significantly bigger challenge than ScottSteiner, thus it is more difficult to learn how to play against it.

Unfortunately, we did not have enough time to gather enough data to enable us to state anything about the win-rate estimations, seen in Figure 11.20, with surety. We only had time to train it for ~978,000 training steps, which is significantly less than the AIs from the main experiment (see Table 11.2). It does seem like adding the punishments could have helped, in the long run, but it could just as likely be a coincidence. Note that the two graphs in Figure 11.20 have a little discrepancy; when we evaluated the AI without punishments, we used 200 evaluation rounds rather than 100 and thus, the data without punishments is a bit less coarse.

However, if we compare the Q-values of the two different experiments (with and without punishments), as seen in Figure 11.21, we see a huge difference. The graph with punishments is increasing and shows a lot more progress and promise, whereas the Q-values for the AI without punishments just fluctuates around 0.

It is impossible to say anything for certain, however there seems to be a lot more promise in the Q-values when punishments is turned on. Essentially, when we turn these punishments on, they constitute a kind of “expert knowledge” where the AI is told instantly, for some actions, that we have decided that the action is took was a bad ac-
Figure 11.20: The estimated win-rate for RLScottSteiner and RLPunisher. Unfortunately, there is not enough data here to show any meaningful difference between the two with any form of certainty.

Figure 11.21: Q-values for RLScottSteiner and RLPunisher. The Q-values, when punishments is enabled, is a lot more promising and is increasing steadily.
tion. Therefore, we expected to see some improvement in the results. Although these improvements do not clearly show in the estimated win-rates (Figure 11.20), we believe that the improvement in Q-values would, eventually, result in improved win-rates as well.

However, it did not solve the issue with the dive in performance as a result of an over-fitting to shooting. It still drops from 5% estimated win-rate to the 1% we see here. While over-fitting to shooting is a problem, it is not apparent that it is the sole issue that causes the over-fitting. We can see in Figure 11.22, that there still seems to be a focus on shooting, but it is not as imbalanced.

Figure 11.22: A histogram through time over selected actions, by RLPunisher. While the “shoot action” (represented by the number 7) is still dominant, the selection of actions is a lot more nuanced, compared to the AI that trained against FreksenThink, Figure 11.19.

11.5 CAN LOG TRAINING STAND ON ITS OWN?

We used a static baseline of 2100 logs when we were experimenting with the learning model. We stopped the log training because we did not see any more progress, as described in Section 11.2.

However, during the experiments we discovered a couple of errors in our code, that might have had an impact on how well the training of the AI, using logs, performed. We only used the logs as an initializer for the more sophisticated training algorithms, so we did not find it necessary to re-start the process, by creating a new log baseline.

Theoretically, we should be able to train the AI using log-training alone. The primary constraint would be that the AI would not be able to learn anything that the logs did not show it. We decided to write
this section to explore the potential performance of training on logs, rather than letting the AI explore the world for itself.

First, we wanted to look into how the changes, that we had made to the code, would affect the log training. We collected 200 logs from matches between BAI and Turing\(^2\), we used these logs continually by iterating the 200 logs over and over again. For performance reasons, we only used 200 logs; the limited set of logs serves to reduce the time used to load the logs. We assumed that the 200 logs would be sufficient, to give a substantial amount of different states.

Looking at Figure 11.23, we can see how the trained network has predicted actions according to the first 100 logs. It is clear that the training algorithm has grossly over-fitted to the shoot action. If we look at Figure 11.24, we see that the predicted Q-values (expected reward of the predicted actions) converge to a value around the 300 mark. Knowing that the AI has over-fitted to an almost-always-shoot policy, this Q-value would be the predicted value of a final shoot action.

![Figure 11.23: Histogram over actions predicted by the AI during observations in 100 logs. It is almost exclusively choosing the “shoot”-action, with some “move backward”-actions. Even if we see that the Q-values converge this histogram gives us a we see that it might not be an optimal strategy](image)

The Q-values observed in Section 11.2 are much smaller than the ones we observe here, in order to reassure us that the Q-values make sense, we will undergo a little thought experiment: If we would assume that the AI wins every time, then we know that it receives a reward of 100 000 at the final state (the reward of 1 for hitting is negligible). In this case, a Q-value of 300 gives us that the AI has shot and missed at least \( \frac{100000}{300} \approx 333 \). In order for this to be true, the games

\(^2\) BAI and Turing are AI’s created at the hackathon, see Chapter 7
11.5 CAN LOG TRAINING STAND ON ITS OWN?

has to have a length of \( \geq 333 \) frames. The AI does not always win, and some games are much longer than 333 frames, therefore the actual win-rate for a Q-value of 300 will be much smaller than the 100% we assumed in the thought experiment. Another overestimation, used in the thought experiment, is the fact that we assume that the player always shoots. There are a few overestimations in this thought experiment, so it is not an exact explanation of what we see happening, however it does show that the scale of the Q-values doesn’t seem out of proportions.

As mentioned, we see a gross over-fit towards shooting as well as a convergence in Q-values, which indicates that the network learning is stalling. This leads us to believe that it is very hard for the network, using log training, to learn the intricate connections, between the different features, like health and damage probability, in order to master the game.

11.5.1 Adversary Copy Using Logs

Using simple log training did not give results that was satisfactory, therefore we turned our attention to another idea. Instead of learning directly from the actions of the AI, we would instead try to copy the actions of the AI throughout the game. We used the same approach as in Section 11.5, taking 200 logs and using them to train repeatedly. The only change that we made to the learning algorithm was to give a reward of: 1 if the network predicted the same action as BAI and \(-1\) if not. Using this reward function, we know that the network is optimizing its ability to copy the actions of BAI, and that the Q-
value must be a value between $\pm 3600$ because one only can take 3600 actions per game. If we look at Figure 11.25, we see that RLCopy predicts a reward of $-4$. According to the reward function this should give a correctness-rate of somewhere under 50% because we know that, if it makes 50% correct predictions it would correspond to a Q-value of zero. Now, looking at Figure 11.26 we see that this is a quite good estimate. As we see that the correctness of the predicted action lies around 30%. Looking at Figure 11.26, it might not seem as if the AI is able to learn any more because the correctness seems to have stagnated around 30%. However, if we look at Figure 11.25, we see that the Q-value is still increasing, so we have some hope that the performance might improve, given more time.

![Figure 11.25: Final Q-value for RLCopy. We see that the value is rising which could indicate that there is more to learn](image)

Having looked at the correctness and the Q-value of RLCopy, we wanted to compare the actions of this AI to the simple log training and the AI that it is supposed to copy. Looking at Figure 11.27, we see how RLCopy evolves over 52 evaluations where we evaluate at log 0, 50, 100, 150 and 200 in the loop. Given that we have 200 logs in the loop, we evaluate two times in a row after log 200 and after log 0 when the loop starts over. Each of the evaluations is evaluated using the same 100 logs.

We see that the network predicts the action to be “moveForward” most of the time. This is good since we also see that this is the most used action by the BAI, however it still overestimates the frequency of the “moveForward” action, which in turn causes the other actions to be underestimated.

We do not really see a pattern in the evolution of RLCopy, other than we see that it chooses some of the actions that have a high frequency for the BAI. The choice of the high frequency actions is due
Figure 11.26: Correctness of RLCopy in terms of correct actions predicted. The Correctness seems to be stagnated around 30%.

Figure 11.27: Development of actions taken by RLCopy together with the actions taken by the AI that it tries to copy. RLCopy correctly identifies the most frequent action, but also overestimates that same action.
to the nature of our reward function. Given that the BAI AI decides on a lot of “moveForward” actions then the network will estimate that it is best to also take a “moveForward” action most of the time.

Even though RLCopy has far from copied the BAI AI at the moment, we see from Figure 11.25 that the predicted value is growing. As mentioned before, this gives us an optimistic hope that this technique could produce something close to a copy of BAI. However, just producing a copy of an existing AI does not really serve a meaningful purpose, from a practical aspect; at best one would end up with something that is as good as the existing solution. This copy-AI could, however, be used in the same manner that we use the evolutionary learning. I.e. as an initialization technique for more sophisticated tools of learning, as it has, rather quickly, been able to learn from an established “expert”.

To get an overview of the differences between the “copy” log-training and the “simple” log-training, we have plotted the last evaluation steps from the two methods, together with a sample from the BAI AI, in Figure 11.28. We see that there is a big difference in the actions that the two AIs chose, RLCopy is much closer to choosing the actions of the BAI AI than the “simple” AI. We are not able to compare the actual quality of the two methods through this figure, as we would need to match them up against each other to know which one is better (which can be seen in Section 11.6). However, it is obvious that RLCopy, would act in a way that looks more similar to what an AI might do, as the “simple” AI would mostly stand still and shoot.

![Figure 11.28: Actions taken by the simple log training, RLCopy and the BAI AI. The BAI and Copy log are much more similar than the simple log](image-url)
While the estimated win-rates and Q-values can give a rough idea of the development in the performance of the AIs, they do not show how the AIs perform relative to each other. Establishing the performance of the AIs relative to each other, and establishing whether one AI is better than the other, is not as straightforward as it may sound. In order to establish which AI is best, one might evaluate their performance against one specific opponent in order to find their win-rate against said opponent. Another way of establishing the best AI would be to let every AI fight against each other in a round-robin style tournament. Either criteria is equivalently valid, with its own merits and there are many other options for deciding which AI is best. In the end, it comes down to a matter of choice and preference. We decided for the latter option, the round-robin style tournament where each AI fights each of the other AIs, in order to measure points gained as well as rounds won and tied.

We opted for this style of tournament, from the intuition that the best AI would be the one that could defeat a wide variety of opponents, as opposed to doing well against a single opponent. Through this round-robin style tournament, every AI would have to test its skills against every other AI.

We used both the hand-written adversaries from the tournaments, described in Section 7.2, as well as the AIs produced in Chapter 11. We matched them against each other one-by-one until every AI had fought every possible combination. For every match, the two opponents would fight 200 rounds, each win gaining the victor 1 point and 0 for a draw. Given that there are 15 AIs, each AI will have to play 14 matches, resulting in the total number of games played (or maximum amount of points gained) to be $200 \times 14 = 2800$. The results of this tournament, can be seen in Table 11.3.

A couple of quick things worth noting: the hand-written AIs has the same rankings relative to each other compared to Table 7.2, except HundenBider which is now at the top. Secondly, three RL-AIs perform better than the weakest hand-written AI, but every RL-AI perform worse than FriskPige, which is the second-weakest AI.

Taking a more in-depth look at the standings, the AIs have two different measures here. The “Win-Loss-Tie” (W-L-T) statistics and the points they got, where 1 points is equivalent to winning 1 round. The actual placement of the AIs, is a bit more complex than that, evidenced by Turing and BAI which has the same W-L-T statistics, but still BAI (with 2035 points) is below Turing (with 1755 points). In order to place them, we did not just measure how many wins they got, but who they won against as well, which was a major factor
for tie-breaking. We used Challonge\(^3\) to handle the tournaments and placements of the AIs.

The hand-written FreksenThink AI, with 4 wins and 10 losses, performed worse than: RLPunisher, RLBAIEvo and RLBAI. RLPunisher and RLBAIEvo had two more wins than FreksenThink, in spite of FreksenThink leading in points. Interestingly HundenBider, at 1st place had just 1974 points compared to BAI which had 2035 points at 5th place. This shows that an AI can perform better over-all, according to these metrics, without actually having the most wins. Winning is not all that matters, an AI can excel by gaining more points than the opponent, even if it wins with 1 point while the opponent gets 0. Being able to consistently out-perform the adversary it is matched against, is more important than doing really well against a few adversaries.

---

\(^3\) A website for managing online tournaments.
DISCUSSION

In this section, we will discuss the results we saw in Chapter 11. A discussion of the results we saw in the final tournament can be seen in Section 12.1, along with a discussion of the impact of training against different adversaries and how we have strayed from our original main focus. In Section 12.2, we will have a look at the results from the continued evolution experiment (RLBAIEvo), and look at how it helps us explain the dip in performance observed in Section 11.2. Section 12.3 discusses the RLPunisher, the results we saw and how “designing” our AIs affect their performance. Finally we will discuss the computational power we had available and how it affected us in Section 12.4.

12.1 TOURNAMENT STANDINGS

If we look at the tournament standings we see that two of our RL-AIs fared better than the others, that being RLBAI and RLPunisher.

RLBAI performed remarkably well, in fact it performed better than RLScottSteiner, RLMix and RLFreksenThink, even though it flat-lined at an estimated reward of 0% (see Section 11.2). However, at 81 points gained out of 2800 possible, it begs the question of whether this placement was down to random luck or whether it is a better AI. However, we see that RLBAI was better than RLScottSteiner which in turn was better than RLMix and RLFreksenThink; we observe that the RL-AI that trained against the strongest opponents also got the best results. This leads us to believe that the results from RLBAI is more than sheer luck and actually has some merit. These observations shows us that even if progress cannot be seen through the measurements we make, that does not mean it is not there. We believe the progress made by RLBAI can be contributed to the meta-rewards granted upon shooting the enemy, allowing RLBAI to get a notion of what the “correct” decisions are, even if it never experiences a win.

Another interesting result is the RLPunisher, which performed best compared to all the other RL-AI. The RLPunisher was only allowed to train for one week but still it achieved much better results than RLFreksenThink and even RLBAIEvo which trained for over one and a half month. More than anything, this demonstrates the impact of the choice of reward function used during training.
**Discussion**

**Varying results against different adversaries**

In the main experiment, we used the same training technique on four different adversaries. These four AIs, were picked from different points of the ladder in the tournament standings from the hackathon. Table 11.3 shows us the difference in performance of these four AIs, and we can see that the harder the adversary used for training, the better it performed at the tournament. Of course, this could be random coincidence but we assume there is some truth behind these results. One of the very interesting parts of these results, is the fact that RLBAI performed fairly well, it even beat the AI FreksenThink. In spite of the fact that, RLBAI never reached more than 0% estimated win-rate. We believe that the meta-rewards has been necessary for RLBAI in order for it to know when it made progress, as it would have been its only indication when it never experienced the rewards from a win. Looking back, estimating the win-rate of RLBAI against a weaker adversary would have helped us measure its performance during training in order to compare how it improved relative to the other AIs. In fact, if we had estimated the win-rate of all the different RL-AIs against the same adversary during training, we would have been able to compare their progress directly.

We began this thesis with stating that our primary focus was on the impact of the adversary on the training process. I. e. whether the training process yield different results if the adversary, the RL-AI trains against, is harder and whether it is better to train against the best, the worst or something in between. Our results indicates that the strongest AI provided the best learning for the RL-AI. However, it is not possible to say anything for certain, as our RL-AIs had issues learning the game, and the amount of wins they gained in the tournament is too low to draw any conclusions.

Due to these complications, which we predicted early on, we decided to switched our focus a bit during the thesis. Knowing that we would not be able to modify the training process or improve the results from the main experiment, since training took such a long time, we decided to change our focus a bit. While our primary focus, and the majority of our time was spent on the main experiment, which compares different adversaries, this thesis has concerned investigation and discussion of both different learning techniques and the impact of the adversary equally.

**12.2 THE PERFORMANCE OF RLBAIEVO**

Early in the main experiment (Section 11.2), we observed that we didn’t see the progress we had hoped for. Discouraged by our initial results, we decided to do an experiment that was significantly
simpler than the 3-step approach used during the main experiment. We decided to do an experiment where we would train an AI, using evolutionary learning only, which is described in Section 11.3. During this experiment we achieved win-rates against BAI, the best AI, of around 18%, which was higher than any other RL-AI could ever hope for. Therefore, we assumed it would easily come out on top during the grand tournament in the end. However, it turns out that RLBAIEvo would come in second, compared to the rest of the RL-AI, and ninth overall.

We were disappointed and wanted to investigate the issue. We found that, looking at the results from the tournament, Appendix B, RLBAIEvo only won 4.5% of the games against BAI, which was significantly lower than the aforementioned 18%. According to Equation 11.1, the number of games the RL-AI played at the last generation in order to establish its fitness of 18%, was around 200 games, which was the same as during the tournament. Since the number of games played was almost equal, the win-rate of 4.5% in the tournament could be regarded as an estimate of the fitness which is just as valid as the estimate of 18%.

To explain that we have two different estimates, which are equally valid, that shows so radically different pictures, we have to look at how the estimate of 18% was measured. During an evolutionary generation, we perform mutations on some individuals, generating x offspring. For simplicity, let’s assume we generate 50 offspring for each generation. What we are then essentially doing is this: we take some individual, perform small mutations and compute the estimate of the win-rate for each of the resulting offspring. As a result we compute 50 estimates for different individuals. Given that there are some variance to the win-rates, we have 50 different estimates that could potentially be over-estimated, and just a single over-estimation would pollute our measurements of the fitness. As a result, we saw the estimated fitness of each generation as the largest estimation, which would likely be an over-estimation. We see that the over-estimations are increasing, so we don’t believe it would hurt the training process itself. However, in retrospect we should have either measured the average fitness as an estimate of a generation’s win-rate, rather than just the highest fitness or we should have improved the quality of the estimate for each individual.

During the experiment in Section 11.2, we saw a dip in the estimated performance when we switched from evolutionary learning to backpropagation. At the time we were puzzled, and we tried to avoid the dip in performance through an experiment in Section 11.4. However we know now, that we were misguided and that the dip in performance is more likely to be a result of this over-estimation we have seen.
Another issue which could affect the performance of the AI, when switching to deep Q learning, could be related to a change of scale in Q-values. In evolutionary learning, the AI is changing randomly, and the specimen that performs best is selected for further breeding. Thus, the expected values for each action an evolutionarily-trained AI will predict, can be of any dimension as long as they are scaled correctly relative to each other. On the other hand, deep learning teaches the AI to predict some value based on a reward function. Therefore, the values predicted by the deep-learning AI will try to predict the returns from this reward function. As a result, the Q-value that the evolutionary network predicts might be very different from the Q-value predicted by a deep network using some reward function. This difference in Q-values between the two networks could cause the dip in win-rate from the switch between evolution and deep learning. Note though that if we switch from deep to evolution then the weights of the deep network is already initialized to accommodate the reward function and as such we would expect that going from deep learning to evolutionary learning would not suffer from this potential issue.

We used the evolutionary learning as a way to search broadly, before going all-in on deep-learning. In retrospect, it might have made sense to switch back and forth between the two techniques, in order to take advantage of the evolutionary learning technique while avoiding this discrepancy in Q-values to arise.

12.3 THE PERFORMANCE OF THE RLPUNISHER AI

When we initially observed the dip following the evolutionary learning step, we thought that it was caused by the AI over-fitting towards the shoot-action which the deep learning tended to do. In an effort to alleviate this issue, we created the RLPunisher AI which would punish the AI when it did something that was objectively bad. We have now found that the over-fitting was not the only problem tied to the dip, however we see from Table 11.3 that the RLPunisher outperforms every other RL-AI, even though it had a very limited amount of time to train. This indicates that directly injecting this expert-knowledge improved the AI’s learning process. In general, one should be wary of designing the behaviour of the AI in this way, as we mention in Section 8.1. However, we believe that the reason the RLPunisher is doing well, and why we feel safe making this intervention, is that we are exclusively limiting it from making pointless actions. E.g. shooting while the shoot-delay is above 0, accomplishes nothing. The only limitation that we have made, that might prevent it from an optimal strategy, is the fact that it is always punished for doing nothing, even if doing nothing is the best course of action.

In general the AIs were subject to a lot of “design” choices made by the developers. E.g. we designed the feature vector provided to
the AIs (which is explicitly advised against by [16]), the reward function, the architecture of the network etc. In Section 8.1, we discussed how hand-picking the features, and introducing expert-knowledge, generally does not work very well, since it inhibits the learning process of the AI, furthermore, we discussed why we need to make these kinds of design choices. Even though we have our best results from the RLPunisher, which had a lot of “design” put into it, one might question whether we generally designed our AIs too much and thus robbed the AIs of their freedom to learn on their own. However, it is hard to escape the necessity of designing e.g. the feature vector to some degree. Part of the problem, we set out to solve, was that we did not have the luxury of being able to extract the pixel-information that one would have provided a human player with.

In retrospect, we could have changed the design of the feature vector to a pseudo-designed vector, as described in Section 8.1, where the AI is provided with an array of sensors, rather than specific information. This array of sensors can be regarded as a low-density pixel-image and thus we expect it to be closer to the degree of freedom for learning that the rendered pixel-information, one would provide a player, with would. However, for the feature vector that we worked with, it seems, based on the results, that it is too hard for the AI to learn the game without providing it with a fair amount of expert-knowledge.

12.4 THE FEASIBILITY OF OUR TECHNIQUES IN LIMITED TIME

Every one of our experiments required a substantial amount of computational time to finish and we think that many of our experiments show promise in the sense that if we gave them more time then they might have learned more. From Section 2.1, we see that Google™ only uses 50 hours to evaluate ~5 000 000 training steps [16]. Given that we have used over one and a half month to evaluate around the same number of training steps, we see that the monstrous power that Google™ is able to muster exceeds the power of our home computers. Very recently Google™ announced an opportunity to get access to some of their Machine Learning (ML) optimized CPU’s which they call TPU’s (Tensor Processing Unit). A single TPU can deliver up to 180 teraflops, and they can be combined into a “pod” which can provide up to 11.5 petaflops. In comparison, our personal GPU’s, e.g. the Geforce GTX 970, provides a measly 4 teraflops. In the future, it will be possible to use the power of these TPU’s, which could mean the training time for a project, like this thesis, would be reduced significantly. Allowing hundreds, or even thousands, times as much training to be performed in the same time-frame. With this kind of training-power, it is possible that our techniques could provide much

1 1 teraflop is equivalent to $1000^4$ floating-point operations per second.
better results. However, since we managed to train some of our AIs on a comparable amount of steps, even though it was over the course of over one and a half months, DeepMind Technologies still experienced more success than our best AI, measured by the performance of the AIs.

We suggest that the primary reasons for this discrepancy is two-fold, the first reason being the difference in input features. As mentioned in Section 12.3, we have designed the input to the AI by hand. While the information it was provided with was not incomplete, as the hand-written AIs did not have any issues, we still limited the information it had available to a small set of features. It can be argued that, providing it with this, very specific, information, forces it to learn how to use each piece of information and just a slight misunderstanding could be catastrophic. Meanwhile, DeepMind Technologies provide their AIs with every piece of information a human player would have available. Secondly, a reason for the discrepancy could be down to the scoring system in each game. In the game used in this thesis, the only way a player could get a point would be if it won the game. However, the Atari games have a scoring system which is a lot more nuanced. Rather than a binary win/loss scoring, they can measure how much of each level they clear, and how many goals they reach. It might be that, having a nuanced scoring system helps the AI recognize progress, which is something we tried to achieve with the meta-rewards for hitting the enemy. We saw with RLPunisher that the AI could quickly adapt to the points it received when they were delivered almost continuously.
CONCLUSION

We set out to evaluate the practical use of Reinforcement Learning (RL) for computer games, through the eyes of a developer. We created a game from scratch, arranged a hackathon where black-box Artificial Intelligences (AIs) were hand-written by our fellow students and challenged these black-box AIs, with AIs of our own which were based on the RL-paradigm of Machine Learning (ML).

Our RL-AIs went through rigorous training processes, using different state-of-the-art techniques. They did not manage to out-perform the hand-written AIs, however they did manage to learn enough about the game for them to have different skill-levels. The resulting AIs showed promise, and especially one RL-AI performed well, namely the one called RLPunisher. The RLPunisher outperformed all the other RL-AIs, in spite of having significantly less time to train (see Section 11.4). However, even for those – very – limited results, we had to let it train for over a week and a half. Going in to this thesis, we had the expectation of creating an AI that would challenge or even beat the hand-written AIs. We are far from that goal, we did not foresee the amount of time it would take to train a single AI, to reach into several months.

Given the possible computing power, presented by Google’s Cloud TPUs which we described in Section 12.4, it is very possible that reasonable results could be obtained using the techniques we have shown in this thesis. However, we are not convinced that the approach we took is the optimal one. Especially the choice of feature vector, i.e. the information we provide to the AIs in each step of the episode, causes uncertainty of whether we have restricted the freedom of the AIs too much.

This thesis should be taken as an example of the immaturity that RL suffers from. While there may be established techniques and strategies for some use-cases of RL in computer games, trying to apply said techniques to a different, less optimal, use-case quickly becomes a tedious and difficult task. While not impossible, the game and the issue that we showed in this thesis, proved a difficult task for learning. We have observed an importance in selecting a “good” reward function to ease the learning for the AI, which suddenly becomes a sort of highly complex hyper-parameter. In general we experienced very limited success.
13.1 FUTURE WORK

Going forward, an investigation of the importance and usefulness of reward functions is necessary. Continuing the experiment from Section 11.4, in order to establish the progress it makes given more time, would provide a very useful insight in this regard.

In general, utilizing stronger hardware would provide a lot better insight into the usefulness of these techniques, as it is very hard to say, at this point, whether the techniques are not feasible or just haven’t been given enough computation-time to flourish.

Furthermore, a study into the use of a sensor-array, as described in Section 8.1, could provide a more useful approach to providing the AI with information. We expect that the use of a sensor-array could provide a middle-ground between these two approaches, by giving the AI information that better resembles the view a human gets, rather than having it base its decisions on e.g. probabilities and distances to obstacles.
Part III

APPENDIX
BRACKETS FROM THE HACKATHON

Figure A.1: Brackets from the round-robin stage, courtesy of http://challonge.com/

Figure A.2: Brackets from the double-elimination stage, courtesy of http://challonge.com/
### THE MATCHES FROM THE FINAL TOURNAMENT

<table>
<thead>
<tr>
<th>Round</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RLMix</td>
<td>12</td>
<td>RLBAI</td>
<td>2</td>
<td>BAI</td>
<td>194</td>
<td>RLPunisher</td>
<td>12</td>
<td>RLPunisher</td>
<td>0</td>
<td>RLPunisher</td>
<td>1</td>
<td>BAI</td>
<td>1</td>
<td>FriskPige</td>
</tr>
<tr>
<td></td>
<td>HundenBider</td>
<td>97</td>
<td>FreksenThink</td>
<td>0</td>
<td>BAI</td>
<td>1</td>
<td>RLPunisher</td>
<td>15</td>
<td>RLPunisher</td>
<td>25</td>
<td>RLPunisher</td>
<td>54</td>
<td>RLPunisher</td>
<td>25</td>
<td>RLPunisher</td>
</tr>
<tr>
<td></td>
<td>Kurt</td>
<td>7</td>
<td>ScottSteiner</td>
<td>154</td>
<td>Kurt</td>
<td>7</td>
<td>Turing</td>
<td>143</td>
<td>Turing</td>
<td>0</td>
<td>Turing</td>
<td>0</td>
<td>Turing</td>
<td>0</td>
<td>Turing</td>
</tr>
<tr>
<td></td>
<td>RLMix</td>
<td>12</td>
<td>RLBAI</td>
<td>2</td>
<td>BAI</td>
<td>194</td>
<td>RLPunisher</td>
<td>12</td>
<td>RLPunisher</td>
<td>0</td>
<td>RLPunisher</td>
<td>1</td>
<td>BAI</td>
<td>1</td>
<td>FriskPige</td>
</tr>
<tr>
<td></td>
<td>HundenBider</td>
<td>97</td>
<td>FreksenThink</td>
<td>0</td>
<td>BAI</td>
<td>1</td>
<td>RLPunisher</td>
<td>15</td>
<td>RLPunisher</td>
<td>25</td>
<td>RLPunisher</td>
<td>54</td>
<td>RLPunisher</td>
<td>54</td>
<td>RLPunisher</td>
</tr>
<tr>
<td></td>
<td>Kurt</td>
<td>7</td>
<td>ScottSteiner</td>
<td>154</td>
<td>Kurt</td>
<td>7</td>
<td>Turing</td>
<td>143</td>
<td>Turing</td>
<td>0</td>
<td>Turing</td>
<td>0</td>
<td>Turing</td>
<td>0</td>
<td>Turing</td>
</tr>
</tbody>
</table>

Figure B.1: Final tournament match results, rounds 1 – 7, courtesy of [http://challonge.com/](http://challonge.com/)
Figure B.2: Final tournament match results, rounds 8 – 15, courtesy of http://challonge.com/


DELEGATION OF WORK

This work is a shared effort, and most of the ideas as well as the implementations have been a shared effort.

However the primary presentation of various sections in this report has been delegated as follows:

**Simon Jensen**

- Section 1.2 Neural Networks
- Section 1.3 Reinforcement Learning
- Section 2.1 Playing Atari with Deep Reinforcement Learning
- Chapter 5 Design
- Chapter 8 Evaluation
- Section 9.2.3 Double Q
- Section 9.3 Extensions
- Section 10.3 Network Evolution

**Lukas Jørgensen**

- Section 2.2 DeepMind Lab
- Section 2.3 OpenAI Gym
- Section 2.4.1 TensorFlow
- Chapter 3 Analysis
- Chapter 4 Thesis Outline
- Chapter 6 Implementation
- Chapter 7 Adversaries
- Section 9.2.1 Recurrent Network
- Section 9.2.4 Dueling Q
- Section 9.2.2 Deep Q
- Section 10.1 Python and TensorFlow
- Section 10.2 Implementation of the network
- Chapter 11 Experiments and training. Except Section 11.5.
This document was typeset using the typographical look-and-feel classicthesis developed by André Miede. The style was inspired by Robert Bringhurst’s seminal book on typography “The Elements of Typographic Style”. classicthesis is available for both \LaTeX{} and LyX:

https://bitbucket.org/amiede/classicthesis/

Happy users of classicthesis usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/

Final Version as of June 15, 2017 (classicthesis).