Learning to play MOBA Games using Neural Networks and Reinforcement Learning

Janine Weber(210990)

A thesis presented for the degree of
Master of Science in Computer Science

Department of Mathematics and Computer Science
University of Southern Denmark
January 2017, Denmark
Supervisors: Rolf Fagerberg, Marco Chiarandini

Abstract

Games have long since proven a valid testing bed for new concepts, that later can be applied to real-life situations. However, most studies focus on developing agents for clear defined domains as provided by games such as Go, Chess and Checkers. Video Games tend to be overlooked in this regard, as their emphasis generally lies on graphics and user satisfaction. This, however, often results in games with unsatisfying AI agents, that either cannot compete with the level of human players or use unfair means to get ahead. The aim of this thesis is to develop a learning agent that plays the MOBA game Heroes of Newerth with little to no previous game-knowledge. The agent uses a neural network to evaluate the state of the game, and trains the network's weights using the temporal difference algorithm TD(λ). The results obtained during the experiments show a promising start into the development of learning agents for MOBA games. Thus, reinforcement learning in combination with neural networks can very-well be used to efficiently teach computer programs to play fast-paced action games, even without prior game knowledge. However, whether this translates into being competitive at a human's level of play is, as of now, unclear.

Contents

1 Introduction

1.1 MOBA Games

Video games are have long since replaced board games in popularity. They are electronic games that can be found on as good as any system with a processor and a screen; from mobile phones to video consoles to desktop computers.

There exist a diverse number of genres for video games, such as first-person shooters, role-playing games, real-time strategies and massively multiplayers. Another genre, which includes League of Legends[1] one of the most played games in the world, is the multiplayer online battle arena (MOBA). MOBA games combine real-time strategy with fast paced action. Within them, two teams play against each other until one of them is defeated. To achieve this, players must not only be able to act in a highly skill dependent environment, but also have to learn to strategically work together with their team mates to triumph over the opponent.

We will discuss at detail the concepts and mechanisms behind MOBA games in Chapter 2.

1.2 Machine Learning

Machine learning is a field within the area of the artificial intelligence, that ultimately seeks to teach computers to learn behaviors without explicitly having to program them. The three most recognized sub-categories of machine learning are supervised learning, unsupervised learning and reinforcement learning.

Supervised learning teaches a computer to recognize patterns by providing a set of example inputs and corresponding desired target outputs. Typically, an external human teacher attempts to share their knowledge with the computer system in the hopes that it can generate a general enough function to be able to apply it to unseen situations. Example applications are speech and handwriting recognition.

In unsupervised learning methods, computer programs are not provided with labeled data to learn from as in supervised learning, but instead have to infer functions on their own in an attempt to model the patterns of a given

[1] http://www.riotgames.com/

data set. Unsupervised learning can for example be used within the area of morphology, that is, to discover regularities behind word forming.

When using reinforcement learning, one seeks to teach computers how to act within a dynamic environment to achieve a specific goal. There is no external teacher to tell the program how to behave, and often even the goal is hidden or unknown. The computer learns by receiving reinforcement signals from the environment. Examples of reinforcement learning are self-driving cars, or AI agents for games.

In relation to supervised learning, we will look at neural networks in Chapter 5. Reinforcement learning will be detailed in Chapter 6.

1.3 Motivation

Nowadays, video games are huge within the entertainment industry. According to games market research done by Newzoo[2], alone in 2014 the estimate of people playing video games was 1.7 billion, which is comparable to the number of people actively participating in traditional sports (1.6 billion). As the gaming industry grows, Artificial Intelligence in advanced game environments is a continually advancing topic that is in constant need of improvement.

Considering the recent successes of computer programs such as the AI agent AlphaGo (Silver et al. (2016)) that learned to play the game Go at a world-class level, neural networks and reinforcement learning have never been more popular. What makes these techniques so fascinating, is that they seek to model biological and psychological concepts of humans. It is beyond interesting to develop computer systems that attempt to model brain functions, or the way living beings learn from experience.

Thus, for the scope of this thesis we seek to make one of the first steps in the development of learning agents for the MOBA game Heroes of Newerth[3], and hope to engage more researchers to follow down the same path.

1.4 Research Goals and Methodology

The research goal for this thesis is summarized in the following statement.

> *Investigate if and to what degree agents can be taught to play the game Heroes of Newerth using reinforcement learning and neural*

[2]https://newzoo.com/

[3]http://www.heroesofnewerth.com/

networks.

The research started by diving into the various areas of artificial intelligence. The previous successes done within the fields of reinforcement learning and artificial neural networks directed the thesis in this direction. The objective became to develop a learning agent for close-combat fighting in Heroes of Newerth.

In order to reach the research goal, a learning agent was implemented that uses a neural networks to evaluate, which action to take, and temporal difference learning to train and improve the network's evaluation capabilities. Learning was achieved through letting the agent play against itself. The produced AI was tested against the original game agents. Modifications were made accordingly to the achieved results.

1.5 Structure

In Chapter 2 the reader will be introduced to the game Heroes of Newerth. Related works regarding reinforcement learning is presented in Chapter 3. We afterwards discuss the necessary background material for this thesis: general agent systems in Chapter 4, neural networks in Chapter 5, and reinforcement learning in Chapter 6. Chapter 7 presents the model of the learning agent, and Chapter 8 evaluates its significance.

2 Introduction to Heroes of Newerth

Heroes of Newerth is a game of the Multiplayer Online Battle Arena (MOBA) genre.

2.1 Game Breakdown

All MOBA games are essentially the same; they may differ in various aspects of the game, but ultimately adhere to the same set of ground rules. Furthermore, they are highly skill-based games, however, players that have before played another MOBA game can more easily adapt to others as well, meaning they generally don't have to go through the same learning curve as new players.

While most MOBA games offer their players various maps to play, Figure 2.1 conceptually demonstrates the map that is most popular and generally associated with MOBA games.[4]

There are two opposing teams, which in the case of HoN are called *Legion* and *Hellbourne*, of five players each. A team's ultimate goal is to destroy the main building in the enemy's *base*; whichever team achieves this the fastest wins. There are no draws. While this may sound simple, the players need to overcome a couple of defenses before they can break down the opponent's base.

Every player controls a single in-game character, also called the *hero* or *champion*, which are a lot stronger than (almost all) minions. Most MOBA games often offer a large collection of different heroes[5], no two of which are ever the same. Every hero tends to fall into one of several roles, such as *tank*, *support* or *damage-dealer*, each of which can typically be sub-categorized even further. Every hero has access to four actions, generally referred to as *abilities* or *skills*. Depending on the ability definition, they can be used to attack opponents or help teammates.

Typically, there exist three main roads that lead from one base to another, also called the *lanes*. There is a *top lane*, a *middle lane* and a *bottom lane*. The remaining territory is referred to as the *jungle*, which generally is a forest. Every thirty seconds (starting at 1 minute and 30 seconds) three sets

[4] In Heroes of Newerth this map is called "Forest of Caldavar".

[5] Currently, Dota2, Heroes of Newerth and League of Legends each offer over 100 heroes, and continuously keep adding more.

of non-player controlled units, known as *creeps* or *minions* spawn, in the base of each team. Each group (also called a *creep wave*) travels slowly along one of the three lanes towards the opposing team's base, and will attack all opposing heroes, buildings or minions in their path. Furthermore, additional creeps, that belong to neither team, spawn at predefined positions in the jungle. They will only attack heroes when provoked.

Heroes can obtain *experience points*, whenever they kill unfriendly units or are within the vicinity of such an event. These experience points *level* the hero, which allows them to grow stronger. At each level, the hero can decide to immediately learn or empower one of its four abilities. Heroes will also periodically gain *gold* during the duration of a match. Gold can be used as a credit buy *items* for the hero, which enhance the hero and make them stronger. Further gold can be acquired by killing enemy (or neutral) creeps (also known as *CSing*, short for *creep-scoring*), or killing opponent heroes or structures.

At every lane two defensive structures are located, which are known as *towers* or *turrets*. These will attack any unit of the opposing team that comes within their immediate vicinity. Towers are generally seen as *objectives* to overcome for the enemy team, and as *safe locations* for the own team. In order for teams to be able to attack the enemy's base, they first have to destroy the structures leading up to it. Typically, one refers to defensive structures by the *tier* at which they are located. Therefore, in the start of a match, teams can only attack the towers that are closest to be map center, the *tier 1* towers. After such a tower has been destroyed, the *tier 2* tower in the same lane (and only in that lane) can be attacked. Thus, in order for a team to destroy the opponents base, they first must destroy the tier 1, 2 and 3 towers in at least one lane.

All structures as well as units that are friendly to a team provide *vision* about the nearby area to all team members. Everything beyond this vision becomes a dark area, also called the *fog of war*. Players still can see the map in such areas, but cannot observe what is current happening there.

A match can generally be divided into two phases; one of which is guided by *laning*, and the other which is related to *teamfighting*. As mentioned, the five heroes generally fulfill different roles within the team. In the start of a match, the game flow is guided by the laning phase, where, for each team, one hero moves along the mid lane; another one along the toplane, a third hero enters the jungle and two heroes move together along the bottom lane. Once the lane heroes enter the center of their respective lanes, they

will encounter the opposing team's hero(s) as well as begin to start killing enemy lane creeps to gain gold and experience. The last hero has a more special role; they will move through the jungle and kill the creeps within to level up. Furthermore, they typically move all across the map to the various lanes in an attempt to assist their teammates there. Once a team feels they have grown strong enough, they will often start to group together to take down the opponents defensive structures.

Games within the MOBA genre are thus highly complex games, that combine strategic thinking and real-time action, each of which takes an equally important part in a match. However, for the scope of this thesis we will only concern ourselves with the one-vs.-one combat of the laning phase.

As the terminology used is very game specific and can be confusing for people new to games of the MOBA genre, Appendix A provides a summary of the most commonly used terms and phrases. In the remaining sections of this chapter we will examine why Heroes of Newerth was chosen as well as the existing AI code.

2.2 Heroes of Newerth as Testing Grounds

At the start of the thesis a very important decision was that of finding appropriate testing grounds for the development of a learning agent for MOBA games. There were several criteria that made Heroes of Newerth a good choice.

The arguably five most popular MOBA games are League of Legends (LoL), Dota 2, Heroes of Newerth (HoN), Heroes of the Storm (HotS) and Smite. While all of the games provide AI agents to some degree, the only game that opened their code base to developers was Heroes of Newerth. Not only can developers create their own agent code for the various heroes, the game provides access to the game console, allowing the creating of a sandbox-like testing environment.

However, two obvious drawbacks could be immediately determined. For one, the documentation for both the code and the game console is non-existent and most information has to be obtained through posts done by other developers on the game's official forums. Secondly, the game would still have to be run with a graphical interface, giving concerns as to the speed of tests and experiments.

Ultimately, the choice came down to either creating a custom MOBA game that would operate on a console and text-only basis, or Heroes of

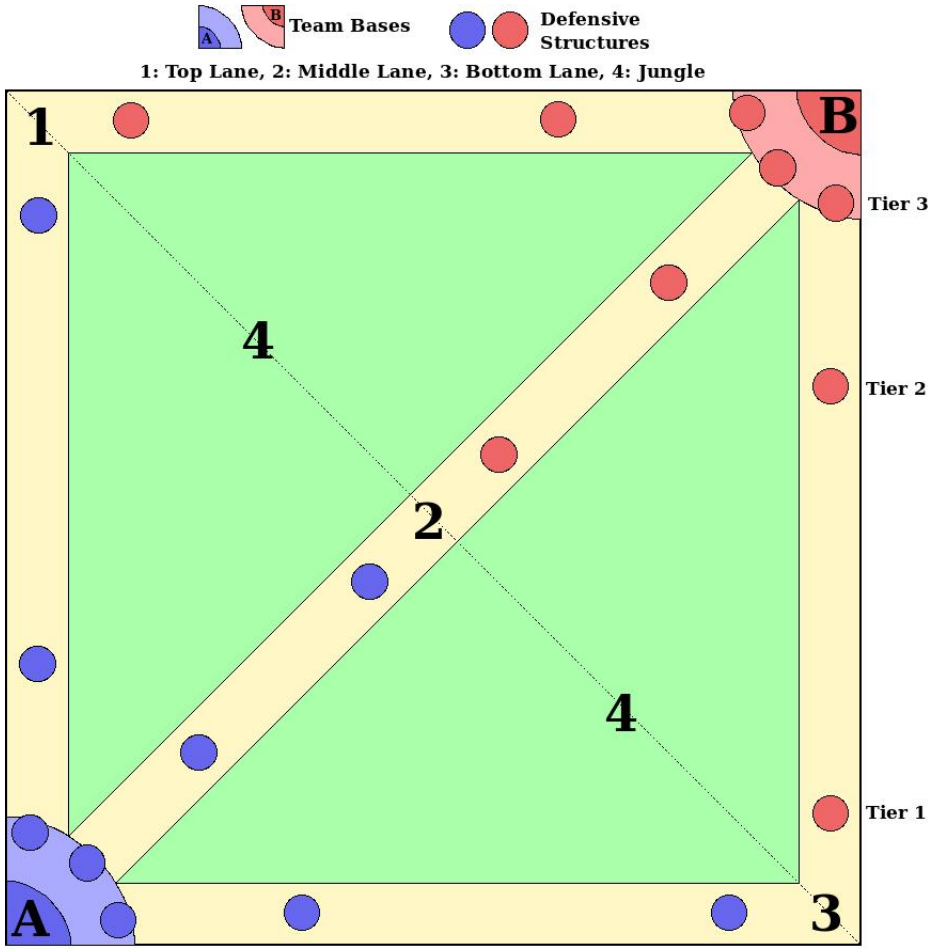


Figure 2.1: Common map setup found in most commercial MOBA games.

Newerth. In the previous section the reader may have gained an insight of how complex MOBA games actually can become. MOBA games are not as simply defined as games such as chess; even a "watered-down" version would still have to implement a large number of game mechanisms, from hero design, to logic, to physics. As this thesis had as basis the development of a learning agent for MOBA games, the question arose whether the custom game would have represented MOBA games appropriately and accurately. Thus Heroes of Newerth was chosen.

In Chapter 8 we will expand the discussion of Heroes of Newerth as a testing environment.

2.3 Existing AI

Heroes of Newerth already offers AI agents (also known as *bots*) for a variety of heroes. As of now this set consists of 42 bots, whereas the game includes 134 heroes in total. We will now examine the methods in which these bots are currently implemented.

In Heroes of Newerth bots are hard-coded, and operate by a rule-based decision making process. More specifically, agents work by switching between a set of predefined behaviors. Every 0.25 seconds, the game asks the bot what action it wants to take. The bot then assesses which of the behaviors that it has access to is the most appropriate to use in the current situation. It determines this by calculating a utility score for each behavior. The score ultimately determines the priority that a behavior is given. Thus, the bot selects the behavior that has the highest utility score, and attempts to execute it. If, for whatever reason, the behavior cannot be executed, the bot will choose the behavior with the next highest utility, and so on. This procedure is summarized in Algorithm 2.1. The default behaviors as well as their utility ranges are listed in Table 2.1. It is possible for developers to override default behaviors or add additional ones, such that they, for instance, become more hero specific and less general.

Lastly, there also exists a team-controller, which decides how and when agents of the same team should work together. The team-controller can override any single bot behavior, that is, they it can tell the bots of a team to "drop everything" and instead do something else.

Table 2.2 lists the files of the code framework which are relevant for the scope of this thesis. While these files can be modified, they generally aren't subject to change. Instead, developers can create two additional files specific to a certain hero; a lua script, where all AI behavior can be overrode, and a XML file which essentially instructs the game how to load the new bot into the game.

Algorithm 2.1 Pseudocode Heroes of Newerth AI.

```
1: procedure HoN-AI
2:     repeat every 0.25s
3:         get state of the world                    ▷ Environment & own state
4:         assess behaviors                          ▷ Rule-based utility calculation
5:         execute behavior with highest utility     ▷ If not possible, next highest,...
6:     until match end
```

Behavior	**Utility**	**Explanation**	**Depends on**
HarassHero	0-100	Deal damage to an enemy hero.	distance to enemy, relative health percent, range of abilities and attacks, proximity of enemy tower, momentum
RetreatFrom Threat	0-100	Flee from enemy.	damage taken over the last 1.5 seconds, agent is targeted by enemy creeps or tower
HealAtWell	0-100	Heal at base.	current health percent, proximity to base
DontBreak Channel	0 or 100	Don't interrupt non-instantaneous actions.	only if agent is channeling an ability
Shop	0 or 99	Buying items.	only if agent is still "shopping"
PreGame	0 or 98	Pre-match preparations.	only if match hasn't started yet
HitBuilding(1)	0, 36 or 40	Attack opponent's defensive structure.	type of building, range to building, building vulnerability
TeamGroup	0 or 35	Group with teammates to attack.	*team controlling code*

HitBuilding(2)	0, 23 or 25	Attack opponent's defensive structure.	whether attacking will put agent into danger
AttackCreeps(1)	0 or 24	Kill enemy creep.	only if creep can be killed by single attack
TeamDefend	0 or 23	Group with teammates to defend.	*team controlling code*
PushBehavior	0 or 22	Attack enemy creeps to advance forward.	opponent's status (alive or dead), agent's attack damage
AttackCreeps(2)	0 or 21	Kill a friendly creep.	only if creep can be killed by single attack
UseHealthRegen	20-40	Use a heal on self.	rune availability, health potion availability
PositionSelf	20	Agent adjusts positioning.	building proximity, creep proximity, enemy position

Table 2.1: General behaviors of Heroes of Newerth bots, ordered by priority from highest to lowest.

File Name	Function
behaviorlib.lua	Contains the functions related to the default bot behavior as seen in Table 2.1. This includes methods for the utility calculation, as well as behavior execution.
botbraincore.lua	This file controls the bot. It processes environment and game state, and ultimately assesses and executes the bot's behaviors.
core.lua	Contains various supporting functions, such as mathematical functions and game data.
eventslib.lua	Game events, such information about when two opposing players engaging in a combat scenario, are processed here.
metadata.lua	Loads and processes the metadata related to the various game maps, as well as the waypoints used to guide in-game pathing of bots.

Table 2.2: Overview of Heroes of Newerth AI code files.

3 Related Work

One of the earliest and most successful game-related applications that used reinforcement learning to train an artificial neural network was created by Tesauro (1995). The author's program, TD-Gammon, was designed to teach an agent to play the game backgammon with little knowledge about gameplay and without an external teacher to guide its actions. This work was so popular, because the agent learned to play near the level of the world's best human backgammon players, and in fact, developed entirely new winning strategies that impacted the whole backgammon community. Furthermore, learning was achieved entirely from having the agent play against itself. In principle, TD-Gammon works like a lot of other computer controlled board-game agents. It chooses its actions by first examining all possible legal moves as well as the opponents possible responses, and then used an evaluation function to estimate these potential board positions. The board position that proved to be most beneficial to the agent then indicated the action to take. However, TD-Gammon exceeded most other computer programs in the way it learned the evaluation function. The neural network's parameters were adjusted after every move within a match, using the TD(λ) algorithm of the area of temporal difference learning. It's objective was to reduce the difference between the evaluations of past and present board positions, which in turn would be reflected in the predictions of the final board positions.

A more recent example is that of Silver et al. (2016), where the authors developed a learning agent, AlphaGo, for the game Go. In the past, Go has proven to be an exceptionally challenging game to be mastered by computer programs because of its huge space of possible board positions and moves. AlphaGo, however, is the first computer program to ever defeat a human professional player in a best of five. The program combines deep neural networks with Monte-Carlo simulation. Similarly to TD-Gammon, a value network is used to evaluate a given board position, and thus, to estimate the chances of winning. A policy network is used to determine the appropriate order and continuation of moves. AlphaGo was trained partly through supervised learning provided by plays made by human experts, as well as by playing against itself and other computer programs.

The study by Mnih et al. (2013) combined deep neural networks with reinforcement learning in order to master a set of Atari 2600 games. This was a very significant milestone as game agents were only capable of mastering a

single specific domain up until this point. This program, however, managed to master 49 different game scenarios. Furthermore, the input to the network was composed of nothing but pixels and scores, and it was trained without any guidance by human experts. The program was able to beat the overall performance of professional human players as well as previous computer agents.

In the Master thesis by Bagge and Grigo (2016), a reinforcement-learning agent was developed to play the game Connect Four. It was based upon approximating an evaluation function for board positions using neural networks. The network was trained using the TD(λ) algorithm, the same which was used in TD-Gammon. Initial learning was produced through self-play. The agent was extended with various search methods, such as minimax and Monte-Carlo tree search.

Up until this point, no research has been made in regards of applying reinforcement learning and neural networks to teaching agents to play games of the MOBA genre. The most closely related study was done by Kvanli and Hammerstad (2014) in their Master Thesis, in which the authors investigate the performance of case-based reasoning agents and multi-agent cooperation for the game Heroes of Newerth. The results showed that, while these methods improve the existing agent logic, MOBA games are still a very unexplored area.

4 Agent Systems

This section provides an overview of what can be understood of agents and agent systems according to the textbook by Russel and Norvig (2010).

An *agent* is an entity that perceives and observes its environment through *sensory inputs* and acts upon it through *actuators*. Human agents gather information about their environment using their eyes, ears and other organs, and can execute actions via, for instance, their hands, arms and legs. Similarly, a robotic agent may perceive its surroundings though cameras and infrared sensors, and have various motors for its actuators.

The implementation of computerized agents depends largely on the characteristics of the environment for which they are designed. This environment is also called the agent's *task environment* and can be specified by the properties and characteristics listed in Table 4.1.

Agents can be classified into different categories that represent the underlying architectures present in almost all intelligent systems. These agent types are summarized in Table 4.2. Independent from the underlying type of agent architecture, all agents can be improved through learning. Learning agents are able to evolve their behavior, which allows them to become more skilled with time, even if their initial knowledge of the world was largely defined by incompetence. The general architecture of such an agent is depicted in Figure 4.1.

For the scope of this thesis we will only be considering utility-based learning agents, whose discussion we continue in Chapter 6.

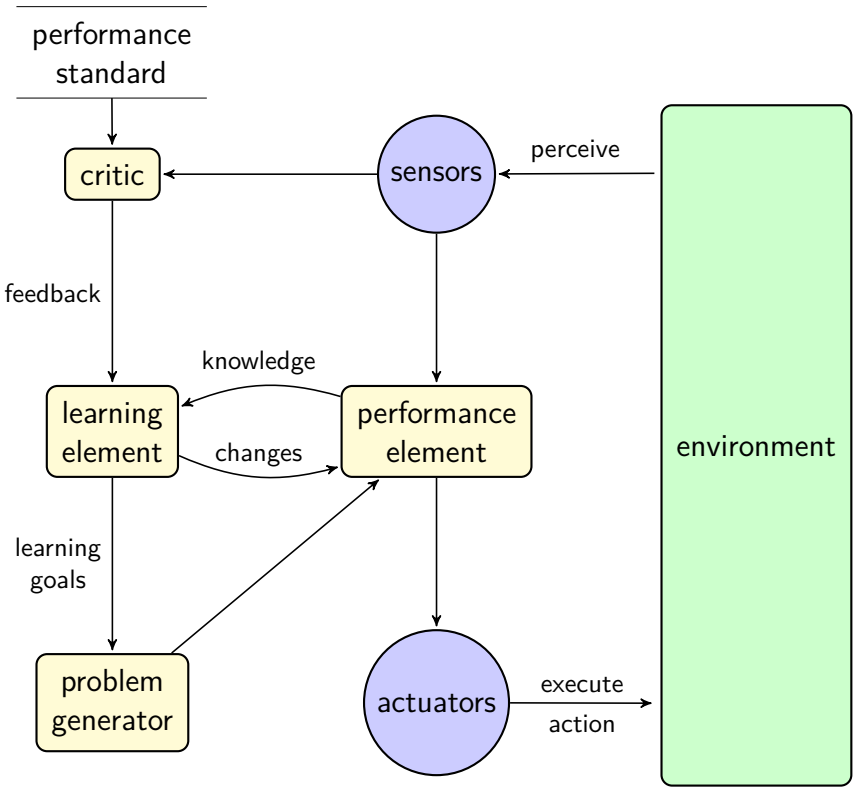


Figure 4.1: A general learning agent.

Task Environment	Characteristics	Description
Observability	Fully observable, partially observable, unobservable	Describes if and to what degree the agent's sensors are able to perceive the environment.
Agents	single agent, multiagent	Defines the number of agents that are present in the environment, and thereby the environment's competitiveness.
Determinism	deterministic, stochastic	Depicts whether the next state of the environment completely depends on the current state and the agent's action executed, or whether the environment possesses some inherent randomness that can impact the next state.
Episodic	episodic, sequential	Expresses whether an agent's actions are considered as atomic episodes, or if the agent's past decisions and actions have an impact on the next state.
Stasis	static, dynamic, semidynamic	Specifies if and to what degree the environment is subject to change while the agent is deliberating.
Discreteness	discrete, continuous	Characterizes how time is related to the percepts and actions of the agent.
Knowledge	known, unknown	States if the outcomes or outcome probabilities for all actions are given.

Table 4.1: Task environments and their possible characteristics.

Category	Description
Simple reflex agent	The agent acts in accordance with a specified rule whose condition matches the current state of the environment, which is defined by the percept.
Model-based reflex agent	The agent monitors the current state of the world by modeling how the environment evolves and how the agent's actions impact it.
Goal-based agent	The agent keeps track of the state of the world and maintains a set of goals it is trying to achieve. It chooses actions that will, immediately or in the future, lead to the accomplishment of its objectives.
Utility-based agent	The agent keeps a model of the world as well as a utility function that measures how fitting a state is to the agent's goals. The agent chooses the next action that will lead to the best expected utility among the utilities for all possible outcome states.

Table 4.2: Types of agent architectures.

5 Artificial Neural Networks

The human brain is made up of a densely interconnected network of billions of brain cells, also called neurons. Activity of neurons is generally excited or inhibited though connections to other neurons. This intricate structure allows us humans to make complex decisions in a short amount of time.

Artificial Neural Networks are computer systems that attempt to model and simulate these various brain functions. This chapter will seek to explain how such a network can be constructed. Section 5.1 looks at the architecture of artificial neurons, Section 5.2 discusses the overall network structure, and Section 5.3 dives into the specifics of teaching a neural network to recognize patterns.

5.1 Artificial Neurons

As described in the textbook by Russel and Norvig (2010) neural networks represent neurons by *nodes*, or *units*. These are connected through *directed edges*, or *links*. Imagine an artificial neuron as an entity that makes decisions based on weighting and evaluating the information it receives through its input links.

The simplest type of neurons are called *perceptrons*. Perceptrons work by accepting multiple input values, calculating the *weighted sum of inputs* and producing a single binary output, which is computed by the unit's *activation function*. In the case of perceptron neurons, the activation function is a *threshold function*, such that neuron's output takes the value 1 if the weighted sum is above some specified threshold and 0 otherwise.

More formally speaking, a link from unit j to unit k propagates the activation a_j from j to k. Every edge has a real-valued weight w_{kj} associated with it, that regulates the contribution of the input pattern to the output. Such a neuron is depicted in Figure 5.1.

Accordingly, the weighted sum of inputs for unit k can be computed by

$$z_k = \sum_{j=0}^{n} w_{kj} a_j, \tag{5.1}$$

where k is the index of any unit in the network that produces an output, j is the index of all units with links directed into unit k, $w_{kj} \in \mathbb{R}$ is the weight of input j into unit k, and a_j is the value provided by unit j. Note, that

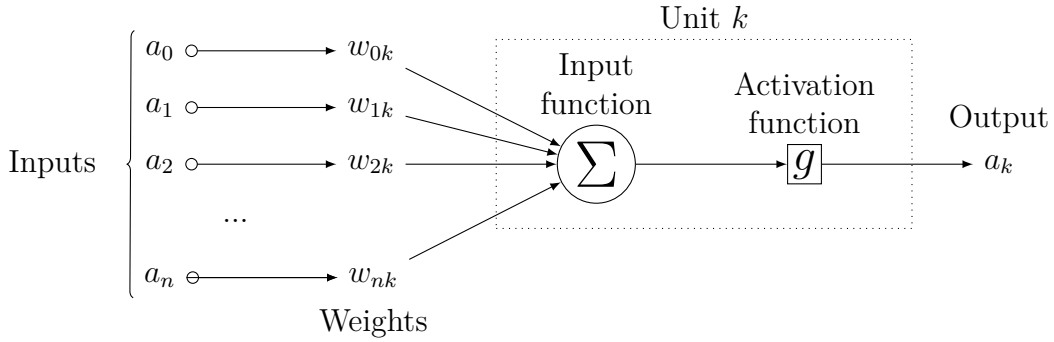


Figure 5.1: A neuron. The unit's output activation is $a_k = g(\sum_{j=0}^{n} w_{kj} a_j)$, where a_j is the output activation of unit j, $a_0 = 1$ is the bias, and w_{kj} is the weight on the link from unit j to this unit k.

$j = 0$ indicates an additional input, called the *bias*. The bias always takes the value 1, and defines how easy it is to activate the neuron.

Then, the activation function g of the unit k can be written as

$$a_k = g(z_k) = g(\sum_{j=0}^{n} w_{kj} a_j). \tag{5.2}$$

Mitchell (1997) states, that the kind of activation function used, determines the type of unit within the neural network, as well as the mapping that is possible within it. In case of perceptrons, g is a stepwise threshold function and therefore neural networks using exclusively perceptrons can only classify data that is linearly separable. However, if g is a logistic function, the output of a unit can be any nonlinear function of its inputs. Units in such networks are often called *sigmoid perceptrons*, or *sigmoid units*. In case of such a sigmoid unit, the output of unit k is defined by

$$a_k = g(z_k) = \frac{1}{1 + e^{-z_k}} = \frac{1}{1 + e^{\sum_{j=0}^{n} w_{kj} a_j}}. \tag{5.3}$$

Both types of activation functions can be observed in Figure 5.2.

Now that we have examined the building blocks of neural networks, we can dedicate the next section to taking a look at the overall structure itself.

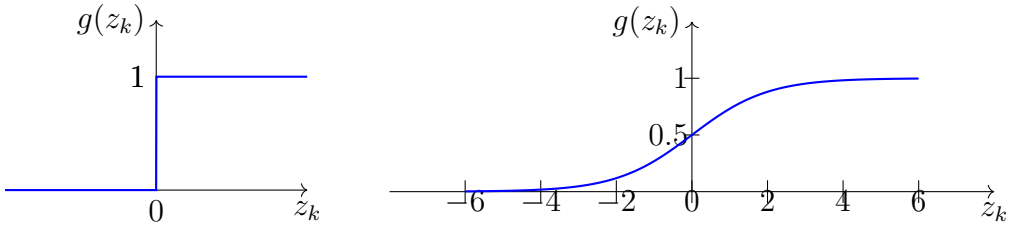
5.2 Network Structure

In the previous section we discussed the details of the neurons that make up a neural network. Now, we will look into how they can be arranged in order to simulate the intricate network of synaptic connections of the human brain.

Mitchell (1997) describes two fundamentally distinct ways to connect the units together to create a network. One such way refers *feed-forward networks*, in which connections are only in one direction such that a directed acyclic graph is formed. The other way is to link the units in the fashion of a *recurrent network*, such that they are allowed to form a directed cycle.

The main difference between those two approaches lies in the preservation of the network's state. Recurrent networks enable earlier input to be fed back into the network, which allows them to incorporate a memory over the history of the network's states. Feed-forward neural networks don't support this, unless they are specifically modeled to incorporate historical values. In practice, recurrent networks are powerful tools but they are said to be more difficult to train due to their complexity. On the other hand, feed-forward neural networks are ideally suited for identifying and modeling relations between input and output values. Generally speaking, recurrent networks have shown to be excellent in problems involving sequential data such as speech and text (LeCun et al. (2015)), whereas feed-forward neural networks are extremely popular and successful in areas such as pattern recognition (Bishop (1995)).

The structure of a network is additionally influenced by its number of *layers*. In *single-layer networks*, input and output units are directly connected, whereas *multi-layer networks* add one or more layers of nodes called *hidden units* in between the input and output neurons. A multi-layer feed-forward



(a) Threshold activation function (b) Sigmoid activation function

Figure 5.2: Different types of activation functions.

neural network is depicted in Figure 5.3.

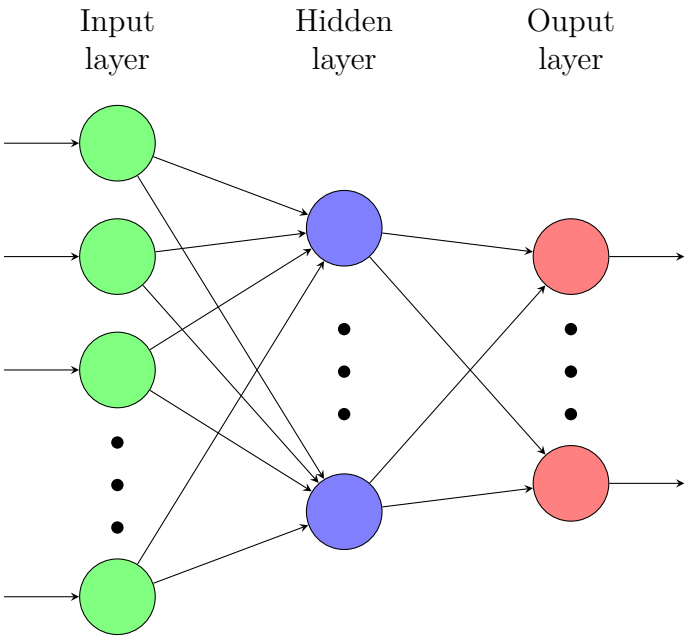


Figure 5.3: Structure of a feed-forward neural network with one hidden layer.

Even though the architectures of the different multi-layer neural networks differ in their specifics, they all share a common foundation. There is always one input layer, at least one hidden layer and one output layer. The number of neurons present in the input and output layers depend heavily on the context for which the neural network is designed.

The input units receive external information about the situation that we want the neural network to evaluate. They don't quite follow the design of the neuron we depicted in Figure 5.1; instead, each input receives one data point from an outside source and merely communicates it (as its activation) to the connected hidden units without making any changes.[6] Specifically, they have no bias nor an activation function[7]. The number of input units is generally influenced by the information that is available as well as data

[6]However, the reader should be made aware that the data may well have been preprocessed in order adhere to certain standards or normalization. Nonetheless, this is not the task of an input unit.

[7]Or, if the reader prefers, the activation function can be seen as the identity function.

selection and structuring. Output units, on the other hand, represent what we seek the network to produce for us.

In essence, the input layer represents a question we want the network to answer, and the output represents the answer itself. A famous early example in the field of artificial neural networks is that of LeCun et al. (1989), where a neural network was trained to recognize handwritten zip code digits. There, the network is presented with a 16×16 normalized image containing a handwritten number, which translates into 256 input units (one for every pixel) taking values between 1 and -1 (dependent on the pixel's gray level). The output is a vector of 10 units (one for each digit class), taking either value 1, when the pattern in the image belongs to the class, or -1, when it doesn't.

In between the input and output layers lie the hidden units, which are organized into one or more hidden layers. Let take a step back and think about what the layers are actually used for. Every layer in a neural network can apply any arbitrary function to the activations of the previous layer to produce another output. For instance, let the activations of the input layer be represented by the vector $\mathbf{a}^1$, the activations of the hidden layer by the vector $\mathbf{a}^2$ and the activations of the output layer by the vector $\mathbf{a}^3$. Furthermore, let the activation function that maps $\mathbf{a}^1$ to $\mathbf{a}^2$ be denoted by f, and let the activation function that maps $\mathbf{a}^2$ to $\mathbf{a}^3$ be g. Following, by applying what we discussed in Section 5.1,

$$\mathbf{a}^2 = f(\mathbf{a}^1)$$

and

$$\mathbf{a}^3 = g(\mathbf{a}^2) = g(f(\mathbf{a}^1)).$$

Hence, extending the number of layers by adding one (or more) hidden layers allows us to execute computations that neither f nor g could achieve on their own.

Following this line of thinking, the network's complexity potential can be increased by either increasing the number of hidden neurons or the number of hidden layers, or both. However, as explained in the paper by Sarle (1994), this also increases the number of weights that have to be trained. For this reason, designer's of neural networks need to consider the trade-off between choosing enough hidden neurons (and layers) to be able to model one's problem efficiently, while at the same time recognizing the associated increasing time performance.

Neural networks with more than one hidden layer are referred to as *deep*. Deep learning has become increasingly popular and brought about some amazing breakthroughs within various fields. A very recent example of such is Google's DeepMind *AlphaGo* AI, which mastered the game of Go with deep neural networks and defeated the European Go champion by 5 games to 0, thereby being the first computer program to ever do so (Silver et al. (2016)).

Nonetheless, it has been proven by Hornik et al. (1989), that standard multi-layer feed-forward networks using sigmoid units are capable of approximating any measurable function to arbitrary accuracy. Therefore, and because this architecture is the simplest and most used type of neural network (Kröse and van der Smagt (1993)), from this point onward we will only consider multi-layer feed-forward neural networks using sigmoid units.

So far we have neglected the discussion of the weights within neural networks. All input units are connected to all units in the first hidden layer, which in turn are connected to either the units in next hidden layer or, if there are no further hidden layers, the output layer. There are no connections between units of the same layer. Every such connection has a weight associated with it, which regulates the relation that exits between the input and output values. Values for the weights are determined by iteratively training the network on various data sets, that demonstrate how to correctly identify particular patterns. In the remainder of this chapter we will explore how this learning process can be achieved.

5.3 Learning

In order for a neural network to produce the desired set of outputs given the application of a set of inputs, the strengths of the connections have to be adjusted. As described in the textbook by Kröse and van der Smagt (1993), there are various methods to achieve this. For one, it is possible to set the weights by the means of *a priori* knowledge.

Another way is to train the neural network by feeding it *training examples*. These training examples serve to teach the network to recognize the correct answer to a given input pattern. This is a popular method called *supervised learning*. The set of training examples is often supplied by an *external teacher*, but can also be provided by the network itself, in which case the learning paradigm is referred to as *self-supervised*.

The most common practice of training multi-layer feed-forward networks

is called *back-propagation*. To understand the concept behind the back-propagation algorithm, lets go back to the example of recognizing handwritten zip codes (LeCun et al. (1989)). Imagine a training example being a handwritten 9, which was broken down to 256 values, each of which represents the gray scale of the image's corresponding pixel. Furthermore, alongside this data the corresponding target output is supplied as a vector of size 10, where every cell is set to -1 except for the 9th, which is 1. Now, the network's actual output, which is computed in the *forward pass* by propagating the activation values in a forward-flow from the input to the output units, may vary from the desired one. We can specify how incorrect the network's answer was by calculating a *training error*. This training error can then be propagated backwards in a *backward pass* through the network using a *learning rule* after which the weights are adapted so as to get the actual output to approximate the desired output.

Before we dive further into the details of the back-propagation algorithm, we discuss and clarify the neural network relevant notation that is going to be used from now on throughout this thesis.

Let I, H and O be the number of input, hidden and output units, respectively, and let L be the number of layers of the network. In addition, let the sigmoid function be denoted by σ, and T be the number of training steps, each of which consists of a training example that has an input pattern and a corresponding target output. Following, let the set of training inputs X be given by

$$X = \{\mathbf{x}^1, \mathbf{x}^2, ..., \mathbf{x}^T\},$$

where any one training input $\mathbf{x}^t$ is a vector containing one data point for each input unit

$$\mathbf{x}^t = [x_0^t{}^8 \quad x_1^t \quad x_2^t \quad \cdots \quad x_I^t]. \qquad \forall t \in \{1, 2, .., T\}$$

Similarly, let the set of correct outputs corresponding to each input pattern be denoted by

$$Y = \{\mathbf{y}(\mathbf{x}^1), \mathbf{y}(\mathbf{x}^2), ..., \mathbf{y}(\mathbf{x}^T)\},$$

where any one target output $\mathbf{y}(\mathbf{x}^t)$ is a vector containing one data point per output unit

$$\mathbf{y}(\mathbf{x}^t) = [y_1(\mathbf{x}^t) \quad y_2(\mathbf{x}^t) \quad \cdots \quad y_O(\mathbf{x}^t)]. \qquad \forall t \in \{1, 2, .., T\}$$

[8] where $x_0^t = 1$ is the bias

Hence, we can write the set of training examples D as

$$D = \{(\mathbf{x}^1, \mathbf{y}(\mathbf{x}^1)), (\mathbf{x}^2, \mathbf{y}(\mathbf{x}^2)), ..., (\mathbf{x}^T, \mathbf{y}(\mathbf{x}^T))\}.$$

We can now expand on Equation 5.3 by defining the weighted sum and activation value for a given training input $\mathbf{x}^t$ by

$$a_j^l(\mathbf{x}^t) = \sigma(z_j^l(\mathbf{x}^t)) = \sigma(\sum_i w_{ji}^l a_i^{l-1}(\mathbf{x}^t)), \tag{5.4}$$

where $l \in \{2, 3, ..., L\}$ specifies the layer of the network, j takes the index of any unit within layer l, i takes the index of any unit with an edge into unit j, w_{ji}^l is the weight associated with said edge, and $a_i^{l-1}(\mathbf{x}^t)$ is the activation produced by unit i.

A commonly used performance scale is the *mean squared error* (MSE), which is

$$MSE = \frac{1}{2T}\sum_{t=1}^{T}(\mathbf{y}(\mathbf{x}^t) - \mathbf{a}^L(\mathbf{x}^t))^2. \tag{5.5}$$

This cost function reflects the network's output activation. When

$$\mathbf{a}^L(\mathbf{x}^t) \approx \mathbf{y}(\mathbf{x}^t), \qquad \forall t \in \{1, 2, ..., T\}$$

that is, when the network's outputs approximate the correct outputs for all training examples, then the training error will be low. When this isn't the case, then the cost function will become increasingly large, dependent on the degree and the number of cases where the network's outputs differ from the correct outputs.

5.3.1 Gradient Descent

The basis for the back-propagation algorithm is the *gradient descent*, which is a method of searching the space of possible weights such that the network's cost function is minimized.

In accordance to the mean squared error of Equation 5.5, we can define the training error E^t for the tth training example $\mathbf{x}^t$ by

$$\begin{aligned} E^t &= \frac{1}{2}(\mathbf{y}(\mathbf{x}^t) - \mathbf{a}^L(\mathbf{x}^t))^2 \\ &= \frac{1}{2}\sum_{k=1}^{O}(y_k(\mathbf{x}^t) - a_k^L(\mathbf{x}^t))^2. \end{aligned} \tag{5.6}$$

Thus, the network's error function E can be computed by summing over the training errors for all training examples

$$\begin{aligned} E &= \sum_{t=1}^{T} E^t \\ &= \frac{1}{2} \sum_{t=1}^{T} \sum_{k=1}^{O} (y_k(\mathbf{x}^t) - a_k^L(\mathbf{x}^t))^2. \end{aligned} \tag{5.7}$$

In oder to minimize the network's cost function, let us first find the *steepest increase* along the error function's surface. To accomplish this, one must compute the derivative of E, also called the *gradient of* E, which is

$$\nabla_{\mathbf{w}} E = (\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, ..., \frac{\partial E}{\partial w_\ell}), \tag{5.8}$$

where $\mathbf{w} = w_1, w_2, ..., w_\ell$ is the vector of all connections within the network. Accordingly, by negating the gradient we can find the direction of the *steepest decrease* and thereby the cost function's minimum, namely, where $\nabla_{\mathbf{w}} E = 0$.

Now that we have established that E is a continuous and differentiable function of the network's weights $w_1, w_2, ..., w_\ell$, we can examine the means by which the network's weights are updated. The traditional gradient descent method requires calculating the error function for the entire data set, as demonstrated in Equation 5.8. However, this becomes computationally expensive for large training sets. Therefore, one variation of this process is the *stochastic gradient descent*. Instead, one now randomly selects a training example from the set, and makes a small adjustment to the weights in the direction of the gradient with respect to the training error for that specific example. Thus, the *training rule* guiding the update of the network's weights is

$$w_i = w_i + \Delta w_i \qquad \forall i \in \{1, 2, ..., \ell\} \tag{5.9}$$

where the Δw_i denotes the change in weight, which is defined by

$$\Delta w_i = -\alpha \frac{\partial E^t}{\partial w_i}, \qquad \forall i \in \{1, 2, ..., \ell\} \tag{5.10}$$

with α being the *learning rate*. The learning rate is a positive constant, which defines the step size of each iteration in the gradient descent search. The smaller α, the smaller the adjustments to the weights at each training step.

5.3.2 Back-Propagation

The stochastic gradient descent rule defined in Equation 5.10 can be computed by means of the *back-propagation* algorithm. Back-propagation is nothing more than a method that seeks to compute $\frac{\partial E^t}{\partial w_i}$ for any weight in the network. For the purpose of this discussion, and to simply the notation that will be used, assume a fixed training example $\mathbf{x}^t$.

Any weight w_{jk}^l can influence the entire network through the corresponding weighted sum of inputs z_j^l. Thus, we can expand the derivative of the error function with regard to $\mathbf{x}^t$ by applying the chain rule, such that

$$\begin{aligned}\frac{\partial E^t}{\partial w_{jk}^l} &= \frac{\partial E^t}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{jk}^l} \\ &\overset{(*)}{=} \frac{\partial E^t}{\partial z_j^l}\frac{\partial}{\partial w_{jk}^l}(w_{jk}^l a_k^{l-1}) \\ &= \frac{\partial E^t}{\partial z_j^l}a_k^{l-1},\end{aligned} \tag{5.11}$$

where (*) stems from the definition of z_j^l (Equation 5.4) and the fact that the only relevant term with respect to w_{jk}^l is $w_{jk}^l a_k^{l-1}$.

All that remains to be done is to calculate $\frac{\partial E^t}{\partial z_j^l}$. Thus, let

$$\delta_j^l = -\frac{\partial E^t}{\partial z_j^l} \tag{5.12}$$

denote the error term associated with unit j of layer $l \in \{L, L-1, ..., 2\}$. We will derive this computation by first considering the case where j takes the index of an output unit, and afterwards by examining the case where j represents an internal unit.

In the first case, we compute the training error for all units k at the output layer, namely, where $l = L$. Similarly to how any w_{jk}^l can influence the network through z_j^l, z_j^l can only impact the network by means of the unit's activation a_j^l. Then, by again applying the chain rule we have

$$\begin{aligned}\delta_k^L &= -\frac{\partial E^t}{\partial z_k^L} \\ &= -\frac{\partial E^t}{\partial a_k^L}\frac{\partial a_k^L}{\partial z_k^L}\end{aligned} \tag{5.13}$$

We can rewrite the first term using Equation 5.6, such that

$$\frac{\partial E^t}{\partial a_k^L} = \frac{\partial}{\partial a_k^L}\frac{1}{2}\sum_j (y_j - a_j^L)$$

where j takes the index of any output unit. Apart from when $j = k$, all derivatives of the sum will be zero. Hence,

$$\begin{aligned}\frac{\partial E^t}{\partial a_k^L} &= \frac{\partial}{\partial a_k^L}\frac{1}{2}(y_k - a_k^L) \\ &= \frac{1}{2}2(y_k - a_k^L)\frac{\partial (y_k - a_k^L)}{\partial a_k^L} \\ &= -(y_k - a_k^L).\end{aligned} \tag{5.14}$$

Let's examine the second term of Equation 5.13 next. By definition, $a_k^l = \sigma(z_k^l)$, and thus it's derivative $\frac{\partial a_k^L}{\partial z_k^L}$ is simply the derivative of the sigmoid function, which we know to be

$$\sigma'(s) = \frac{\partial \sigma(s)}{\partial s} = \sigma(s)(1 - \sigma(s)).$$

Following, we have

$$\begin{aligned}\frac{\partial a_k^L}{\partial z_k^L} &= \sigma'(z_k^L) \\ &= \sigma(z_k^L)(1 - \sigma(z_k^L)) \\ &= a_k^L(1 - a_k^L).\end{aligned} \tag{5.15}$$

Substituting the derived expressions 5.14 and 5.15 into Equation 5.13 yields

$$\delta_k^L = (y_k - a_k^L)a_k^L(1 - a_k^L), \tag{5.16}$$

after which we can combine Equations 5.10 and 5.13 into a definition for the stochastic gradient descent rule for output units, which is

$$\Delta w_{kj}^L = \alpha(y_k - a_k^L)a_k^L(1 - a_k^L)a_j^{L-1}. \tag{5.17}$$

Last but not least, we need to consider the second case, namely, how to define the training rule for hidden units. This case is more complicated, as updating the weights w_{ji}^l, where j takes the index of any unit in layer $l \in \{L-1, L-2, ..., 2\}$ and i represents any unit of layer $l-1$ that has a

direct input into j, depends additionally on how w_{ji}^l can indirectly impact the network. Thus, we must consider all units k of layer $l+1$ that are have as input the activation output of unit j. Therefore, we can write

$$\begin{aligned}
\frac{\partial E^t}{\partial z_j^l} &= \sum_k \frac{\partial E^t}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} \\
&\overset{(1)}{=} \sum_k -\delta_k^{l+1} \frac{\partial z_k^{l+1}}{\partial z_j^l} \\
&\overset{(2)}{=} \sum_k -\delta_k^{l+1} \frac{\partial z_k^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial z_j^l} \\
&\overset{(3)}{=} \sum_k -\delta_k^{l+1} w_{kj}^{l+1} \frac{\partial a_j^l}{\partial z_j^l} \\
&\overset{(4)}{=} \sum_k -\delta_k^{l+1} w_{kj}^{l+1} a_j^l (1 - a_j^l),
\end{aligned} \tag{5.18}$$

where (1) follows from Equation 5.12, (2) by applying the chain rule, (3) because we can write

$$z_k^{l+1} = \sum_i w_{kj}^{l+1} a_j^l \tag{5.19}$$

in accordance to Equation 5.4, and thereby

$$\begin{aligned}
\frac{\partial z_k^{l+1}}{\partial a_j^l} &= \frac{\partial}{\partial a_j^l} \sum_i w_{ki}^{l+1} a_i^l \\
&\overset{(*)}{=} w_{kj}^{l+1}
\end{aligned} \tag{5.20}$$

where (*) follows because the only term that involves a_j^l is $w_{kj}^{l+1} a_j^l$; and finally (4) by substituting expressions from Equation 5.15.

By Equation 5.12, we can rewrite what we derived in Equation 5.18, such that

$$\delta_j^l = a_j^l (1 - a_j^l) \sum_k \delta_k^{l+1} w_{kj}^{l+1}. \qquad \forall l \in \{L-1, L-2, ..., 2\} \tag{5.21}$$

Lastly, we conclude the proof by defining the stochastic gradient descent rule for internal units as

$$\Delta w_{ji}^l = \alpha \; a_j^l (1 - a_j^l) \sum_k \delta_k^{l+1} w_{kj}^{l+1} a_i^{l-1}. \tag{5.22}$$

The back-propagation method is summarized in its entirety in Algorithm 5.1, where the input is the set of training examples and the termination condition can be any arbitrary rule such as a specified number of iterations or minimum threshold for the error of all training examples.

Algorithm 5.1 Pseudocode for the back-propagation algorithm.

```
1:  procedure BP(D)
2:      repeat
3:          for all (x, y) ∈ D do
4:              Forward Pass:
5:                  (1) Compute the vectors of the weighted inputs and the
                    activations for all units in every layer by recursively ap-
                    plying Equation 5.4.
6:              Backward Pass:
7:                  (2) For all units in the output layer, calculate the training
                    errors by using Equation 5.16.
8:                  (2) For all units in the hidden layers, calculate the training
                    errors by repeatedly using Equation 5.21.
9:              Update the Network:
10:                 (3) Using the corresponding training errors where appro-
                    priate, update all network weights in accordance to Equa-
                    tions 5.17 and 5.22.
11:     until termination condition is met
```

6 Reinforcement Learning

In the last chapter we examined a supervised learning technique, namely, artificial neural networks, where an external teacher provides what the network's correct response should be given a specific input pattern. Thereby, the network is able to adopt the desired behavior. However, this type of learning may not always be suited to (or even applicable in) every problem instance. Consider, for example, a situation for which the correct behavior is unavailable or even unknown. Supervised learning cannot be applied to such uncertain territory, but instead the agent must somehow learn from its own experience.

Expectation takes a great part in the way humans learn from experience. Whenever we take an action or make a decision, we immediately establish an associated *expected outcome*. When the *actual outcome* that follows a decision varies from what we predicted, we learn to adopt our future expectations for similar situations. More specifically, as described in the study by Schultz (2007), our brain calculates a *prediction error*, that represents how our expectation varied from the real outcome. The *neurotransmitter dopamine* regulates the feelings of reward and pleasure. In case of a *positive prediction error*, because things turned out better than expected, more dopamine is released. In the opposite case, where things are worse than predicted and the brain calculated a *negative prediction error*, the release of dopamine is decreased. Thereby, humans can adapt their expectations, and make decisions appropriately.

Accordingly, within the field of machine learning, teaching an artificial agent how to behave in an environment by the way of providing *rewards* and *punishments* is referred to as *reinforcement learning*. Section 6.1 details agents and environments, Sections 6.2 and 6.3 introduces the reader to the core theory of reinforcement learning and we conclude the chapter with *temporal difference learning* in Section 6.5.

6.1 Agents and Environments

In Section 4 we were already introduced to the concepts of agents and environments. Now, we will redefine and specify these terms in relation to reinforcement learning.

As described in the textbook by Sutton and Barto (2005), the applications

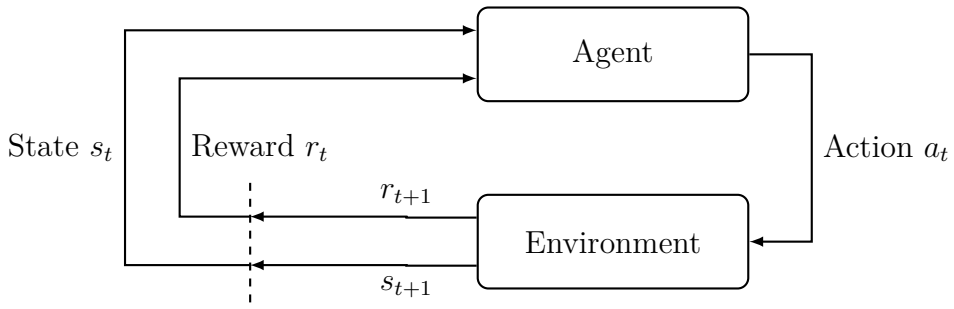


Figure 6.1: The agent-environment interaction.

of reinforcement learning often revolve around interactive environments, in which the agent and the environment act reciprocally with each other over a specified time horizon. Such an interaction between agent and environment is depicted in Figure 6.1. At each discrete time step $t \in \mathbb{N}$, the agent receives some representation of the current state of the environment $s_t \in \mathcal{S}$, where $\mathcal{S}$ is the set of all possible environment states. On this basis, the agent selects an action $a_t \in \mathcal{A}(s_t)$, where $\mathcal{A}(s_t)$ is the set of all actions that the agent can carry out from this state. Once the agent has executed an action, the world is directly impacted by it. Thus, the agent finds itself in a new state s_{t+1}. Furthermore, the agent receives a reward $r_{t+1} \in \mathcal{R}$, where $\mathcal{R}$ is the set of all receivable rewards, that represents how beneficial the agent's behavior was in the pursuit of achieving its goals; that is, whether the situation has improved or worsened from the agent's point of view. This process can be seen in Figure 6.2. Such a sequence of interactions can be of *episodic* or *continuous* nature. The former divides the reinforcement learning task into a sequence of *episodes* (also called *episodic tasks*) each of which starts in an initial state and ends in a terminal state, and the latter keeps the task as one, which starts in an initial state and has the potential to never terminate.

Generally, the agent's objective is to develop a strategy that tells it which action is the best to take from a given state, such that its goals are achieved as fast as possible. This strategy is most often called the agent's *policy*.

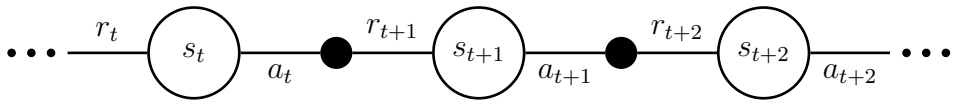


Figure 6.2: A sequence of actions with intermediate rewards and states.

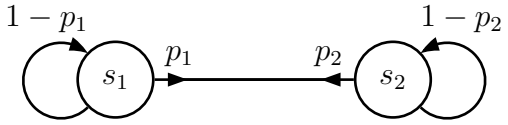


Figure 6.3: A simple Markov Chain with two states and the related transition probabilities.

Reinforcement learning methods teach the agent how to adapt their policy to learn from its experiences; that is, to maximize the rewards obtained over time by the agent. We will discuss policies in Section 6.3.

Let us look at the various aspects of how reinforcement learning differs from other methods that can be used to determine agent policies, such as supervised learning. For one, the feedback given to the agent is in the form of a sequence of intermediate rewards, and not a pair of input-output values. Another important difference is the trade-off introduced by the distribution of training examples, which are determined by the sequence of actions the agent chooses. Therefore, the question arises whether it's more beneficiary to let the agent *explore* unknown states and actions, or whether one should favor the *exploitation* of states and actions that have already proven to achieve high rewards. This trade-off will be examined in Section 6.6.

6.2 Markov Decision Processes

Before we dive further into the theory behind reinforcement learning, we have to discuss how problems within the category are usually are described as detailed.

A *Markov chain* is a process that consists of a finite number of states with stochastic transition functions, where the next state only depends on the current state and not on the past sequence of actions. This allows for the states to be treated independently. A stochastic process that adheres to these rules is said to have the *Markov property*. Figure 6.3 displays such a Markov chain.

A simple example of a *Markovian process*, that is, a Markov chain that has the Markov property, is a game of chess. Each player has perfect information about the state of the game, and the probability of winning the game by applying a specific move is not impacted by the actions that were done in the past. On the other hand, card games such as poker are a great example

of non-Markovian processes. Not only does each player not know what cards their opponents have, but additionally factors such as betting and bluffing as well as a player's preferred play-style need to be considered. For instance, knowing that a player likes to play aggressive rather than passive has a great impact on the winning conditions.

Markov decision processes (MDPs) are an extension of Markov chains. In addition to the properties of Markov chains, they allow for the extension of *actions* that enable choices, and *rewards*, which provide motivation for learning. Given any state $s \in \mathcal{S}$ and action $a \in \mathcal{A}(s)$, the probability of landing in each possible successor state $s' \in \mathcal{S}$, also called the *transition probability*, is defined as

$$\mathcal{P}^a_{ss'} = \Pr\{s_{t+1} = s' | s_t = s, a_t = a\}. \tag{6.1}$$

Accordingly, the *expected reward* associated with starting in state s, taking action a and landing in state s' is given by

$$\mathcal{R}^a_{ss'} = \mathbb{E}\{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\}. \tag{6.2}$$

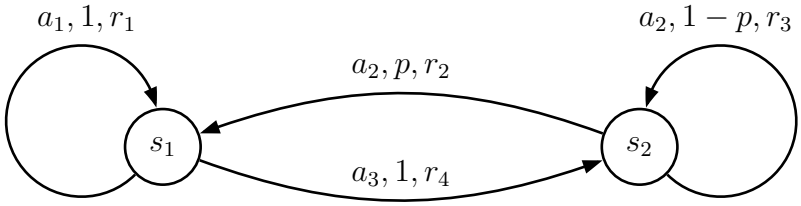
An example of a MDP can be seen in Figure 6.4.

6.3 Value Functions and Policies

By means of a policy, denoted by π, the agent determines its next best move. Policies can be either of *deterministic* or *stochastic* nature. In case of the former, a policy $\pi(s)$ deterministically select an action $a \in \mathcal{A}(s)$ based on the current state $s \in \mathcal{S}$, whereas the latter describes a policy $\pi(s, a)$ that models the probability of taking action $a \in \mathcal{A}(s)$ from state $s \in \mathcal{S}$ for all actions and states.

A policy's quality can be measured by the expected rewards the agent can obtain by following this policy. Consider any episodic task of a reinforcement learning problem. We can define the cumulative function of the rewards at any time step of the episode as the sum of the discounted rewards it receives over the future. Thus, let the *discounted return* for any time step t be

$$\begin{aligned} \mathcal{G}_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} ... \\ &= \sum_{k=0}^{T} \gamma^k r_{t+k+1}, \end{aligned} \tag{6.3}$$



$s = s_t$	$s' = s_{t+1}$	$a = a_t$	$\mathcal{P}^a_{ss'}$	$\mathcal{R}^a_{ss'}$
s_1	s_1	a_1	1	r_1
s_1	s_1	a_3	0	0
s_1	s_2	a_1	0	0
s_1	s_2	a_3	1	r_4
s_2	s_1	a_2	p	r_2
s_2	s_2	a_2	$1-p$	r_3

Figure 6.4: A simple MDP transitioning graph, where each arrow from state s to state s' is labeled with the action, transition probability and expected reward; and the corresponding table of transitioning probabilities and expected rewards.

where T is number of time steps in the examined episode until reaching a terminal state, and $0 \leq \gamma \leq 1$ is also known as the *discount rate*. The discount rate shows how far sighted the agent is. If $\gamma = 0$, then the agent focuses only on immediate rewards, whereas if γ approaches 1, the agent puts a greater weight on also taking future rewards into consideration.

In order to determine the best policy for an agent, reinforcement learning methods generally attempt estimate *value functions*. Value functions indicate how beneficial it is for an agent to be in a certain state, or how good it is for an agent to perform a specific action in a given state.

We can define *state-value function* for policy π, denoted by $\mathcal{V}^\pi(s)$, as the expected return when starting in state $s \in \mathcal{S}$ and following policy π

$$\begin{aligned} \mathcal{V}^\pi(s) &= \mathbb{E}_\pi\{\mathcal{G}_t | s_t = s\} \\ &= \mathbb{E}_\pi\{\sum_{k=0}^{T} \gamma^k r_{t+k+1} | s_t = s\}, \end{aligned} \tag{6.4}$$

where $\mathbb{E}_\pi\{\cdot\}$ defines the expected value if the agent follows policy π, and t is any time step.

Let us take a step back and think about what is actually happening. Given that an agent currently finds itself in state $s \in \mathcal{S}$, it can take any action a of a set of possible actions, $\mathcal{A}(s)$. Once this action has been carried out, the world could change in various ways, leaving the agent in one of several new states $s' \in \mathcal{S}$, along with providing a feedback to the agent via a reward r. The state-value function then describes how good or bad it is for the agent to be in a given state, and allows us to distinguish between which future state is the most desirable. Hence, we can describe the relationship between the value of a state s and the values of its successor states s' via the *Bellman-Equation* (Sutton and Barto (2005)):

$$\begin{aligned} V^{\pi}(s) &= \mathbb{E}_{\pi}\{r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_t = s\} \\ &= \sum_{a} \pi(s,a) \sum_{s'} \mathcal{P}^{a}_{ss'}[\mathcal{R}^{a}_{ss'} + \gamma \mathbb{E}_{\pi}\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_{t+1} = s'\}] \\ &= \sum_{a} \pi(s,a) \sum_{s'} \mathcal{P}^{a}_{ss'}[\mathcal{R}^{a}_{ss'} + \gamma V^{\pi}(s')] \end{aligned} \tag{6.5}$$

The Bellman-Equation 6.5 averages over all possible actions and successor states, and weights each of these possibilities according to their probability of occurrence. More precisely, it specifies that the value of state s must be equal to the discounted value of the expected successor state s' plus the expected reward that was obtained along the way. This relationship is illustrated Figure 6.5a.

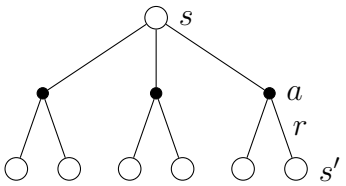
Similarly, we can define the *action-value function*, denoted by $\mathcal{Q}^{\pi}(s,a)$, as the expected return obtained when starting from state s, taking action a, and then following policy π:

$$\begin{aligned} \mathcal{Q}^{\pi}(s,a) &= \mathbb{E}_{\pi}\{\mathcal{G}_t | s_t = s, a_t = a\} \\ &= \mathbb{E}_{\pi}\{\sum_{k=0}^{T} \gamma^k r_{t+k+1} | s_t = s, a_t = a\} \end{aligned} \tag{6.6}$$

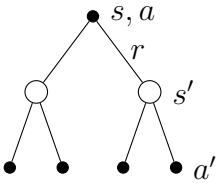
with the corresponding Bellman-Equation being

$$\mathcal{Q}^{\pi}(s,a) = \sum_{s'} \mathcal{P}^{a}_{ss'}[\mathcal{R}^{a}_{ss'} + \gamma \sum_{a'} \pi(s',a') \mathcal{Q}^{\pi}(s',a')]. \tag{6.7}$$

This is demonstrated in Figure 6.5b.



(a) Starting state $s \in \mathcal{S}$; three possible actions in $\mathcal{A}(s)$; each leading to two possible successor states, $s' \in \mathcal{S}$, and associated with intermediate rewards r.



(b) Starting in state $s \in \mathcal{S}$ with action $a \in \mathcal{A}(s)$; two possible successor states, $s' \in \mathcal{S}$, and associated with intermediate rewards, r.

Figure 6.5: Backup diagrams for (a) $\mathcal{V}^\pi(s)$ and (b) $\mathcal{Q}^\pi(s,a)$. Solid nodes indicate actions, hollow nodes indicate states. The paths from root to leaves indicate the possible future scenarios.

For finite Markov decision processes, policies can be partially ordered. A policy π is said to be better than or equal to another policy π', if its expected return is greater than or equal to that of policy π' for all possible states $s \in \mathcal{S}$

$$\pi \geq \pi' \iff V^\pi(s) \geq V^{\pi'}(s). \tag{6.8}$$

An *optimal policy*, denoted by π^*, is a policy that is better than or equal to all other policies, and thereby provides that highest expected return that can be achieved. Accordingly, optimal policies can be defined in terms of the *optimal value functions*. Let the optimal state-value function be given by

$$\mathcal{V}^*(s) = \max_\pi \mathcal{V}^\pi(s) \tag{6.9}$$

for all $s \in \mathcal{S}$. Equivalently, the optimal action-value function is

$$\mathcal{Q}^*(s,a) = \max_\pi \mathcal{Q}^\pi(s,a) \tag{6.10}$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$.

In accordance to Equation 6.5, the value of a state under an optimal policy must be equal to the expected return for the best action from that state. This is also known as the *Bellman optimality equation* for $\mathcal{V}^*$. Therefore, we can

write

$$\begin{aligned}
\mathcal{V}^*(s) &= \max_{a \in A(s)} \mathcal{Q}^{\pi^*}(s,a) \\
&= \max_a \mathbb{E}_{\pi^*}\{\mathcal{G}_t | s_t = s, a_t = a\} \\
&= \max_a \mathbb{E}_{\pi^*}\{r_{t+1} + \gamma \sum_{k=0}^{T} \gamma^k r_{t+k+2} | s_t = s, a_t = a\} \\
&= \max_a \mathbb{E}\{r_{t+1} + \gamma \mathcal{V}^*(s_{t+1}) | s_t = s, a_t = a\} \\
&= \max_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma \mathcal{V}^*(s')].
\end{aligned} \tag{6.11}$$

Similarly, the Bellman optimality equation for $\mathcal{Q}^*$ is given by

$$\begin{aligned}
\mathcal{Q}^*(s,a) &= \mathbb{E}\{r_{t+1} + \gamma \max_{a'} \mathcal{Q}^*(s_{t+1}, a') | s_t = s, a_t = a\} \\
&= \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma \max_{a'} \mathcal{Q}^*(s', a')].
\end{aligned} \tag{6.12}$$

Finally, how we can use this knowledge to find an agent's optimal policy? Given an optimal state-value function $\mathcal{V}^*$, we can do a one step look-ahead (as shown in Figure 6.5a), and then the actions that lead us to be in the best possible successor state are, trivially, the optimal actions for that state. Now, intuitively one might think that this would yield in the selection of actions based only on their short-term consequences. However, as $\mathcal{V}^*$ already takes into consideration the expected returns obtained in the future, thus, executing a one-step lookahead also translates into long-term optimal actions. Therefore, an optimal policy *greedily* directs the agent's behavior with respect to $\mathcal{V}^*$. Similarly, a policy can be established if $\mathcal{Q}^*$ is given, except that a one-step lookahead isn't necessary, and the agent can simply look for the action that maximizes $\mathcal{Q}^*(s,a)$.

In practice, solving the Bellman optimality equation is infeasible, as it requires exhaustive searches over the state space. Hence, most methods that seek to solve reinforcement learning problems instead compute approximate solutions of the Bellman optimality equations. We will continue this discussion in Section 6.5.

6.4 Value Function Approximation

There exists different methods for solving reinforcement learning problems that are based on the value function approach. In order to determine the

policy that maximizes the return obtained by the agent, the mentioned methods maintain a set of estimates of the expected returns for a given policy, e.g. the current one or the optimal one. Let's take, for instance, the optimal state-value function $\mathcal{V}^*$, and look the concepts behind three different methods; *Dynamic Programming (DP)*, *Monte-Carlo methods (MC)* and *Temporal Difference methods (TD)*.

DP requires a complete and perfect model of the environment, which then is used to approximate $\mathcal{V}^*$ (or $\mathcal{Q}^*$), and thereby achieve an optimal policy π^*. Conceptually, these methods execute full sweeps through the entire state space $\mathcal{S}$, thereby updating the value for every state s based on the values of all possible successor states s', their transition probabilities and rewards r. This is shown in Figure 6.6a. This process continues until convergence, that is, until the state-value updates no longer result in any changes. Accordingly, DP approaches are computationally very costly.

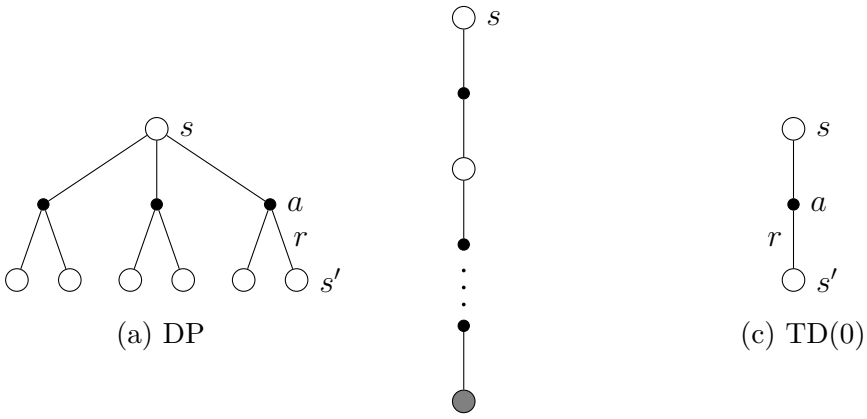
In turn, MC methods don't require complete knowledge of the environment, but instead learn from *transition samples*, as displayed in Figure 6.6b. They are conceptually simple, but cannot be applied to step-by-step incremental computation. Instead of updating the state-values step-by-step, they are updated in an episodic fashion; that is, a sample sequence of transitions is generated, starting within an initial state and ending in a terminal state. MC algorithms only update the state-value estimates and policies after the completion of an episode; and only for those states that were visited during the episode.

Finally, another method is temporal difference learning, which combines MC and DP. TD algorithms need no complete knowledge of the world, and can learn directly from experience while being fully incremental. They approximate the current state-value estimate based on previously learned estimates (like DP), and learn by sampling the environment according to some policy (like MC). Consider the simplest TD method: TD(0). The value estimate for the state node is updated on the basis of the one sample transition from it to the successor state. This is illustrated in Figure 6.6c.

In the next Section we will look at a specific temporal difference method, namely TD(λ).

6.5 TD(λ)

TD(λ) is a temporal difference learning algorithm invented by Sutton and Barto (2005), and seeks to unify Monte Carlo and Temporal Difference meth-



(a) DP (b) MC (c) TD(0)

Figure 6.6: Three methods of updating the value of a state. The root node indicates the state to be updated. Below follow all transitions and leaf nodes whose rewards and estimated values are part of the update.

ods.

Similarly to our discussion of TD(0) in the last section, every time a state is visited, TD(λ) looks forward in time and examines the future states and rewards in order to make a decision on how to best combine them for the update of the state. After having done this and updated the value for the state accordingly, the algorithm moves to the next state and repeats this process. This way, preceding states will never have to be visited again, whereas future states are viewed and processed repeatedly as the time steps progress. Before we can deeping our understanding of TD(λ), we need to look at the spectrum of how states can be updated. As an update to a state is based on future values, we will also refer to it as a *backup*.

To recap, Monte-Carlo methods perform a backup based on all transitions of an episode, whereas TD(0) only requires the value of one successor state for an update. Performing a backup based on an intermediate number of rewards (more than one, but less than all) is referred to as *n-step TD prediction*. Figure 6.7a displays the spectrum ranging from TD(0) to MC updates. In between these two extremes are the n-step backups, which are based on n steps of accumulated rewards and the estimated value of the nth succeeding state.

The λ-*return algorithm* is a special way of averaging n-step backups. This

can be observed in Figure 6.7b. Here, each of the n-step backups is weighted proportional to λ^{n-1}, where $0 \leq \lambda \leq 1$, and the average contains all n-step backups. A backup is carried out towards a return, called the λ-*return*, at each time step t. This λ-return is defined by

$$\mathcal{G}_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1}\mathcal{G}_t^{(n)}. \tag{6.13}$$

As can be observed, the λ-return decreases with time. Accordingly, if we set $\lambda = 0$, we get the previously mentioned TD(0) method; however, if we set $\lambda = 1$, we get the MC method. The λ-return algorithm then performs backups using the λ-return by computing the increment, denoted by $\Delta\mathcal{V}_t(s_t)$, to the value of the state at that time step

$$\Delta\mathcal{V}_t(s_t) = \alpha[\mathcal{G}_t^\lambda - \mathcal{V}_t(s_t)], \tag{6.14}$$

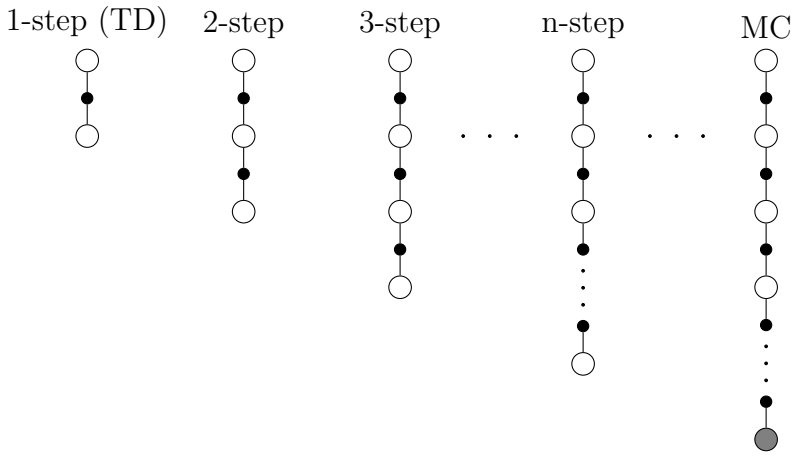
where $\mathcal{V}_t$ is the state-value at time step t, $s_t \in \mathcal{S}$ is the state visited at time step t, and α is a constant step size parameter. Thus, an increment can be understood as looking into the future for the return $\mathcal{G}_t^\lambda$, that follows the visit to state s_t, and then using that return as *target value* for $\mathcal{V}_t(s_t)$ to strive to.

However, as each step updates its state value by using knowledge from the future, this interpretation of TD(λ) is impractical to use in practice. Therefore, we must look at another way that allows us to approximate these theoretical principles.

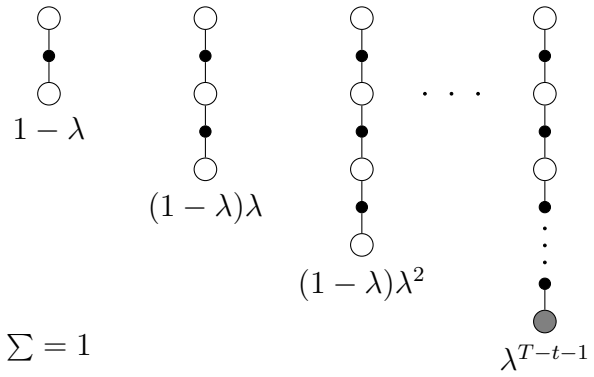
Thus, the technical side of TD(λ) builds upon a view that is directed backward in time. Similarly to the gradient descent method described earlier for neural networks in which we seek to minimize the mean squared error, we seek here to minimize the *TD error*. After each selection of an action or entering of a state, we can evaluate the new state and determine whether things have gone better or worse than expected. Therefore, the reinforcement events that are to be considered are the moment-by-moment one-step TD errors. The TD error at time step t is defined as

$$\mu_t = r_{t+1} + \gamma\mathcal{V}_t(s_{t+1}) - \mathcal{V}_t(s_t). \tag{6.15}$$

Eligibility traces are temporary records of the occurrence of an event (e.g. the visiting of a state or taking of an action). The trace indicates the degree to which each state is eligible for undergoing learning changes. Hence, whenever



(a) Spectrum of one-step backups to MC backups.



(b) TD(λ), λ-return.

Figure 6.7: A comparison of (a) n-step TD prediction and (b) the backup diagram for TD(λ). Hollow circles represent states; solid back ones are actions; and solid gray ones specify terminal states.

a TD error occurs, only the eligible states (or actions) are assigned credit or blame for the error, hence, for all states $s \in \mathcal{S}$

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & \text{if } s \neq s_t \\ \gamma\lambda e_{t-1}(s) + 1 & \text{if } s = s_t, \end{cases} \tag{6.16}$$

where s_t is the state visited at time step t. At every time step, the trace of all unvisited states decays by $\gamma\lambda$, whereas only the trace of the state that

was visited in this time step is increased by 1.

To clarify, at each time step, we calculate the current TD error and assign it backward to each previous state in accordance to the state's eligibility trace at that time. Instead of looking into the future as we did before, we now shout the TD errors backward towards states that were visited in the past. Hence, states that we haven't visited recently will be less impacted (or more precisely, less rewarded or punished) by the error than states that were recently updated.

Hence, the TD error signal triggers proportional updates to all recently visited states, as signaled by their traces. Therefore, for all states $s \in \mathcal{S}$ the increment to $\mathcal{V}_t(s)$ due to a backup is defined as

$$\Delta\mathcal{V}_t(s) = \alpha\mu_t e_t(s). \tag{6.17}$$

Online updates, that is updates that are done time step by time step, then lead to the following value change for all states

$$\mathcal{V}_{t+1}(s) = \mathcal{V}_t(s) + \Delta\mathcal{V}_t(s). \tag{6.18}$$

The entire algorithm can be seen in Algorithm 6.1.

Thus, we conclude the discussion of temporal difference methods for reinforcement learning problems. However, we will see in Chapter 7, how we can use the techniques of neural networks discussed in Chapter 5 to approximate the state-value function for the TD(λ) algorithm.

6.6 Exploration vs Exploitation

We earlier hinted at the trade-off between exploration and exploitation, but have yet to discuss related strategies. Kaelbling et al. (1996) summarizes the most studied approaches, one of which we will look at now for this scope of this thesis.

Consider the simple but widely-known reinforcement learning problem called the *k-armed bandit problem*, where an agent finds itself at a row of k gambling machines (also known as the "one-armed bandits"). The agent is only permitted a fixed number of pulls n, and it can only pull one arm at a time. Hence, it has to decide which machines to play how many times, and in what order. When machine $1 \leq i \leq k$ is played, it pays off the value 1 or 0, depending on a machine-specific probability p_i, where all payoffs are independent and p_i is unknown to the agent. The objective of the gambling

Algorithm 6.1 Pseudocode for the TD(λ) online algorithm.

procedure ONLINE-TD(α, γ, λ)
Inititialization:
    $\mathcal{V}(s) \leftarrow$ arbitrary, $\forall s \in \mathcal{S}$
TD-learning:
    for all episodes **do**
        initialize s
        $e(s) \leftarrow 0, \forall s \in \mathcal{S}$
        repeat
            $a \leftarrow$ action given by π for s
            execute action a
            observe reward r and resulting state s'
            $\mu \leftarrow r + \gamma \mathcal{V}(s') - \mathcal{V}(s)$ ▷ TD error
            $e(s) \leftarrow e(s) + 1$ ▷ Eligibility trace increment
            for all $s \in \mathcal{S}$ **do**
                $\mathcal{V}(s) \leftarrow \mathcal{V}(s) + \alpha \mu e(s)$ ▷ Update state-values
                $e(s) \leftarrow \gamma \lambda e(s)$ ▷ Decay traces
            $s \leftarrow s'$
        until s is terminal

agent is to maximize the sum of payoffs earned during the sequence of n pulls.

This example demonstrates the essential trade-off introduced between exploration and exploitation. As the agent doesn't know the probabilities of the machines, it cannot simply pick the machine that has the highest payoff. The only way for it to determine how beneficial a machine is, is by *exploring*; that is, by picking a random machine it hasn't tested yet and playing a couple of turns. This way, the agent hopes to establish an idea of the machine's payoff probabilities. However, as the agent only has a set number of pulls, extensive exploring may lead to the agent wasting pulls on machines with bad payoffs and never get to *exploit* the knowledge he has obtained so far. Therefore, it is best to develop a strategy that balances on the edge between exploring and exploiting.

One such strategy is called the *ε-greedy policy*, in which the agent chooses an action uniformly at random with probability ε and the best action determined by a value function otherwise. Initially, when an agent has very little or no knowledge of the environment and consequences, this pays off, as the agent gets to explore the state space. However, as the agent learns a value function with more accurate estimates according its obtained knowledge, choosing random actions becomes less and less beneficial. Hence, ε, also known as the *exploration rate*, is set to decay with time, such that the agent gets to explore only the action space that has proven to be beneficial.

7 Designing a Solution

7.1 Defining the Reinforcement Learning Problem

This section will look at the various parts that make up a reinforcement learning problem, such as environment, states and action, and connect them to Heroes of Newerth setting.

7.1.1 Environment Characteristics in Heroes of Newerth

In this section we will shortly detail the environment in which the agent(s) will operate by examining its characteristics in accordance to the properties listed in Chapter 4.

Heroes of Newerth (and other MOBA games in general) is a highly competitive (as well as cooperative) game, in which two or more opposing agents face each other on a predefined map. Human players can observe game events using their eyes and ears, however, they are limited to the parts of the game that are visible to their in-game character. For instance, enemy heroes are only visible if the player (or any of its teammates) are within their immediate vicinity on the map. Otherwise, the player has no knowledge of the position and status of the opposing players.

Generally, most game events are subject to set, unchanging rules, such as the spawn of creep waves or the mechanisms of the teams' defense structures. However, while a lot of actions have a predictable nature, they are nonetheless most often stochastic. For example, the auto-attack of any hero has a specific minimum and a maximum damage, and any such damage dealt within the game is chosen uniformly at random from this range of values. Furthermore, there are chance events within the game that have a probability to occur in certain situations.

What makes MOBA games highly skill-dependent activities is, that everything is subject to change at all times of the game. Unlike games like chess, where players execute their actions in turns, MOBA games are fast paced action games. Hence, quick reaction times are often a must for players in order to perform well. However, this isn't all; players must also be able to think strategically, because no one action determines whether a team looses or wins. Instead, any action a player takes as well as the actions of its teammates and opponents, have an impact on how the future will turn out.

The above discussed properties of Heroes of Newerth's environment are summarized in Table 7.1.

Task Environment	**Characteristic**
Observability	partially observable
Agents	multi-agent
Determinism	stochastic
Episodic	sequential
Stasis	dynamic
Discreteness	continuous
Knowledge	unknown

Table 7.1: Overview of the environment's characteristics in Heroes of Newerth.

7.1.2 States, Actions and Rewards

For games such as Checkers, the space space $\mathcal{S}$ is clear defined, namely, the set of all possible board positions. Furthermore, there are some obvious initial approaches as to how to represent a state s; for instance, one variable for every board position whose value indicates whether the position is occupied by an agent's piece (value 1), whether it is occupied by an opponent's piece (value -1), or if the position is empty (value 0).

However, for MOBA games neither is easily identifiable. This has various reasons. Firstly, there are a large number of factors that describe the environment. Assume we have two heroes that encounter each other in a lane; then, a state could be composed of information about the level of both heroes, their *stats* (e.g. the health percent, mana percent, damage output, etc.), their abilities, their position on the map, their proximity to units and objectives, and so on and so forth. The number of possible data points that could be influential to a state's value is huge. Secondly, what increases this challenge drastically, is the fact that there has been as good as no research in regards to teaching agents to play MOBA games.

Loosely said, a state has to contain information about the status of the opponent and the agent itself. The most important factors are how healthy a hero is, as well as how strong they are. We will investigate this further in Chapter 8.

The set of possible actions for a state $\mathcal{A}(s)$ is composed of the hero's

abilities, the *auto-attack* (which is the default means by which a unit deals damage) and actions needed for the agent's movement (such as moving in specific directions, or fleeing from and pursuing the opponent). Especially the movement-related actions are open for some variation.

The agent only receives rewards at the end of a match, that is, whenever a terminal state has been reached. We define terminal states in more detail in Chapter 8, however, for now it is sufficient to say that a match ends when one of the two heroes is dead. Also the rewards aren't given in the traditional sense, as we have seen for TD(λ). Instead, the estimate for the successor state $\mathcal{V}_t(s_{t+1})$ is replaced by the match's outcome. The reward for winning is 1, the reward for loosing is 0 and the reward for a draw is 0.5. As the reader may realize, these values are exactly what we want the network's estimate to be. For instance, if our agent wins, we want the network's estimate to approach 1.

As the successor states depend only on current state and the actions taken from our agent and its opponent, we can model the Heroes of Newerth gameplay as a large, finite Markov Decision Process.

7.1.3 Value Function

Before we can go into the specifics of the value function, we should decide upon what it is supposed to model. In this, inspiration was drawn from Tesauro (1995)'s TD-Gammon, which uses a value function to estimate the probability (for each player) of winning the game from the given state. However, for simplicities sake, we are only interested in the agent's chance of winning the game and not his opponent's. We hence can state the value function by defining the question we want the neural network to answer for us:

> *Given the current state of the game, what is the predicted outcome of the match from the point of view of the learning agent?*

There are various ways in which the value function can be designed. For one, we could have the network attempt to estimate the probability of winning for every action using the action-value function $\mathcal{Q}(s, a)$. That is, the network has one output for every action, which given the state estimates the probability of winning by taking this action.

Another possibility is that we estimate the value of a state, using $\mathcal{V}(s)$. Actions can then be chosen by simulating the successor states for all actions;

that is, given the current state of the game, the agent establishes within its own memory the state that would result from taking an action, and it does this for all actions. It then uses the neural network to evaluate each of these potential successor states. The action corresponding to the state that proves to be the most promising for the agent, is then chosen and executed. The trouble is that this isn't as simple for a game like Heroes of Newerth as it is for something like Chess, as the state space isn't a fixed set, but instead a continuous space. Thus, simulating the successor states isn't as clear cut. Hence, another way of doing this is by adding one or more variables to the state representation that indicate the agent's actions. Assume the state contains one boolean variable for every action. Then, for each action, the variable that represents an action is set to 1 and all other action-variables to 0, and the network is used to output an estimate. The best estimate indicates the action to be chosen.

As we will discuss in Chapter 8, the latter case proved to be easier to train.

7.2 Value Function approximation using Neural Networks

For the scope of this thesis we seek to use neural networks to approximate the value function, which in our case represents the chance of winning the game from a given state. Simply stated, the learning agent inputs the information about the current state of the game into the neural network, and then lets the neural network compute the approximate value of all possible actions. The action that provides the highest probability of winning will then be chosen.

The network will be trained using to the reinforcement learning algorithm, TD(λ), that is, by using the states and rewards that follow the execution of an action.

Recall from Chapter 6 that TD(λ) updates the value function by

$$\mathcal{V}_{t+1}(s) = \mathcal{V}_t(s_t) + \alpha[r_{t+1} + \gamma\mathcal{V}(s_{t+1}) - \mathcal{V}_t(s_t)]e_t(s)$$

for all $s \in \mathcal{S}$, where

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & \text{if } s \neq s_t \\ \gamma\lambda e_{t-1}(s) + 1 & \text{if } s = s_t. \end{cases}$$

Furthermore, recall from Chapter 5, that the learning strategy in neural networks was based on trying to minimize the mean-squared error on the observed training examples, by computing the gradient of the network's error function. Furthermore, this gradient is the vector of partial derivatives with respect to the network's weights.

When combining reinforcement learning and neural networks, we now seek to train the network by computing the gradients of its outputs (the value approximation) with respect to its weights over an entire episode. In order to represent how recent a gradient was computed, eligibility traces are used to increasingly decay their values as time continues.

Now, let $\boldsymbol{\theta}_t$ denote the vector of the network's parameters at time step t. Thus, $\boldsymbol{\theta}_t$ can be updated by applying

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha\mu_t\mathbf{e}_t, \tag{7.1}$$

where the TD error is defined as before

$$\mu_t = r_{t+1} + \gamma\mathcal{V}_t(s_{t+1}) - \mathcal{V}_t(s_t)$$

and the eligibility traces for each parameter are computed by

$$\mathbf{e}_t = \gamma\lambda\mathbf{e}_{t-1} + \nabla_{\boldsymbol{\theta}}\mathcal{V}_t(s_t). \tag{7.2}$$

Given the input s_t and weights $\boldsymbol{\theta}_t$, the value function $\mathcal{V}_t(s_t)$ is approximated by the network's activation output $\mathbf{a}^L$ at time step t. Hence, instead of computing the partial derivatives of the network's cost function, we compute the partial derivatives of the network's activation output. Thus, the error at each hidden and output neuron of the network can be redefined to

$$\delta_j^l = \frac{\partial a^L}{\partial z_j^l}, \tag{7.3}$$

where j is any unit within layer l, and $l \in 2, 3, ..., L$. This can be easily calculated, because we already know that in the case of the output layer L, when $l = L$,

$$\boldsymbol{\delta}^L = \frac{\partial \mathbf{a}^L}{\partial \mathbf{z}^L} = \sigma'(\mathbf{z}^L) = \mathbf{a}^L(1 - \mathbf{a}^L).$$

Finding the error of hidden units remains unchanged.

When putting it all together, $\nabla_{\boldsymbol{\theta}_t} \mathcal{V}_t(s_t)$ can be computed by finding the partial derivatives of the network's output with respect to the network's weights:

$$\frac{\partial \mathbf{a}^L}{\partial w_{ji}^l} = a_i^{l-1} \delta_j^l, \tag{7.4}$$

where i is any unit of layer $l-1$, j is any unit of layer l, and w_{ji}^l is the weight of the link from unit i to unit j. Finally, the error at neuron j can be computed by

$$\delta_j^l = \begin{cases} a_j^l(1-a_j^l) & \text{if } l \text{ is the output layer, } l = L \\ a_j^l(1-a_j^l) \sum_k \delta_k^{l+1} w_{kj}^{l+1}, & \text{if } l \text{ is a hidden layer layer, } l \in \{L-1, ..., 2\}. \end{cases} \tag{7.5}$$

The pseudocode for the gradient descent version of TD(λ) is listed in Algorithm 7.1.

Algorithm 7.1 Pseudocode for the gradient descent TD(λ) online algorithm.

procedure ONLINE-BP-TD(α, γ, λ)
  $\boldsymbol{\theta} \leftarrow$ small random values
  for all episodes **do**
    $\mathbf{e}_0 \leftarrow 0$
    $s_1 \leftarrow$ initial state of episode
    $t \leftarrow 1$
    repeat
      $a \leftarrow$ action with the best chance of winning from state s_t $\triangleright$ as estimated by the network
      execute action a
      observe reward r and resulting state s_{t+1}
      $\mu_t \leftarrow r_{t+1} + \mathcal{V}_t(s_{t+1}) - \mathcal{V}_t(s_t)$
      $\mathbf{e}_t = \lambda \mathbf{e}_{t-1} + \nabla_{\boldsymbol{\theta}_t} \mathcal{V}_t(s_t)$
      $\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \alpha \mu_t \mathbf{e}_t$
      $t \leftarrow t+1$
    until s_t is terminal

7.3 Model

7.3.1 Game Mechanisms

Before we look at how the AI is designed, we need to discuss the general game mechanisms in Heroes of Newerth relevant to this thesis.

Recall that every hero has access to the same default attack mechanism, the auto-attack. The auto-attack has a specific damage output and attack range. This auto-attack is seen as instantaneous, that is, when a hero decides to auto-attack the opponent (which is within the specified range), the game immediately initiates this action. The instantaneous arrival at the opponent is only limited by the game physics; the time it takes for the attack to travel from the agent to the opponent, which usually isn't more than a fraction of a second (< 250ms). After this attack was completed, the auto-attack can't be used again for a short time, the *cooldown time*. The cooldown time length typically depends on the hero, but often is around 1.5-2 seconds.

Some hero abilities are also instantaneous, others are not. In the latter case they instead are *channeling* abilities, which require the hero to remain unmoving for the the channel duration. First after it is fully completed, the ability will activate and afterwards go on cooldown. If the hero is interrupted while channeling (by moving, using other abilties, or being stopped by *crowd control* abilities of the opponent), the ability's effect will never occur and the ability goes on cooldown anyway. Crowd-control refers to temporary effects of some abilities such as *silences* (during which the affected hero cannot use any abilities) or *stuns* (which completely disables the hero).

7.3.2 AI Structure

The cooldowns of abilities are not global knowledge. Thus, when humans play MOBA games and they see their opponent use an ability, then only instinct and experience will tell them when an ability is about to come available again. However, the more and more humans play, the more reliably they can predict this.

The issue with bots is, that they don't have instincts. Thus, the options are either to somehow design a model that also includes the past, or to simply keep track of when an ability was used. As we base our design on the theory of reinforcement learning and Markov decision processes, the second options seems the only viable one. Therefore, whenever the agent sees the opponent use an ability, it will start a timer for that ability that counts down to when

the cooldown is over. How does the bot "see"? Every time a significant event occurs within the game (for instance, a hero uses an ability), the game engine sends this information to the agent's `oncombatevent()` method, which allows developers to process this data.

Now, lets look at how the agent acts within the game. Every `botframe`, which is 1/20th of a second, the game engine calls the agent's `onthink()` method, which the agent can use to evaluate the next action to take. The choice is then send as a request to the game engine, which executes the action. Figure 7.1 illustrates this interaction between the game's engine and the agent code. Algorithm 7.2 outlines the `onthink()` procedure.

The various events in HoN make training the network challenging. For instance, while a hero is channeling an ability, time still continues around him and the environment keeps changing. Furthermore, different abilities may have different channeling times. If learning would only happen after an action has been executed, this would mean that learning would take time at irregular intervals. This is not what we want, as this may unfairly reward or punish some actions over others. The agent picks a new action 250ms, which is about the reaction time of a human. Learning takes place after an action has been executed, as well as every 250ms in cases where the hero is channeling or otherwise prohibited from executing another action.

Table 7.2 illustrates such a timeline for a single match. At 250ms into the game, the agent picks the first action a_1, which is immediately executed, as it is instantaneous. Thus learning happens immediately afterwards. At 500ms, the next action a_2 is picked, which has a channeling time of 500ms. Thus, at 750ms, while the hero is still channeling, another snapshot of the environment is taken and learning is applied. At 1000ms, a_2's channel time finishes and gets executed. Learning is applied, and the next action a_3 is selected.

7.3.3 Neural Network Layout

We will be working with a feed-forward multilayer neural network that uses one hidden layer. Sarle et al. (1997) point out a few helpful tips for the design of a neural network

- one hidden layer is sufficient for most problems
- the number of hidden neurons should lie somewhere between the number of input units and the number of output units

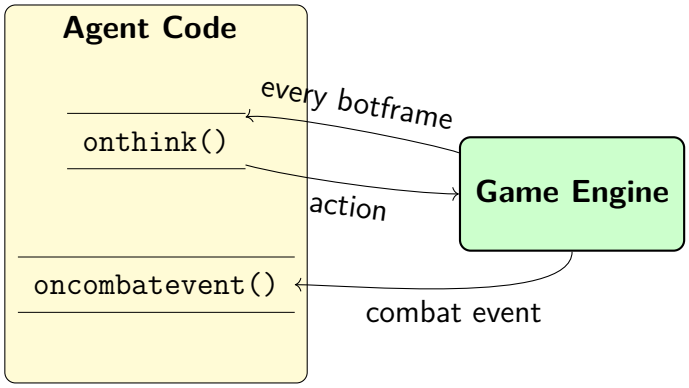


Figure 7.1: Interaction of HoN Game Engine and Agent Code.

0	Match Start
250	Action a_1 (Start & End), instantaneous
500	Action a_2 (Start), channeling time: 500
750	
1000	Action a_2 (End); Action a_3 (Start & End), instantaneous
1250	Hero Stunned (Start), duration: 750
1500	
1750	
2000	Hero Stunned (End); Action a_4 (Start), channeling time: 1000
2250	
...	
$\mathcal{M}$	Match End

Table 7.2: Example time-line of game mechanisms and AI logic. The left-hand-side shows when learning takes place, and the right-hand-side the game events at that point.

- binary inputs should be modeled as $\{-1, 1\}$, instead, as this produces better approximations

Algorithm 7.2 Pseudocode for `onthink()` method.

```
1: procedure ONTHINK
2:     if Start of Experiment then
3:         ε ← 1
4:         decr ← decrease value
5:         θ_0 ← small random values
6:     if s_t is terminal then
7:         e_0 ← 0
8:         t ← 1
9:         s_1 ← initial state of game
10:    s_t ← collect state information
11:    j ← random value
12:    if j ≤ ε then
13:        a ← random action
14:    else
15:        a ← action with the best chance of winning from state s_t
16:    execute action a
17:    observe successor state s_{t+1}
18:    if s_{t+1} is terminal then
19:        V_t(s_{t+1}) ← match outcome          ▷ 1: win, 0: loss, 0.5: draw
20:    else
21:        V_t(s_{t+1}) ← network estimation for state s_{t+1}
22:    θ_{t+1} ← θ_t + α(V_t(s_{t+1}) − V_t(s_t))e_t
23:    t ← t + 1
24:    ε ← ε − decr
```

7.4 Self-Play

The question that has yet remained unanswered is how we want to train our agent, or more precisely, which opponents should we use against our agent.

For one, we could train our agent against human players. However, as it is uncertain how much data is required to train an agent, it is impractical to use this approach as it would take a lot of time, as well as a human subject that is qualified (and has time) to do so. Another option is to train the agent against other AI, such as the Heroes of Newerth bots. We will, in fact, experiment with this, but as it is uncertain whether our agent will be able to achieve the same level of skill as the HoN-bots, it isn't an optimal solution

either. Similarly, we could apply supervised learning, but this would require the knowledge of the correct actions given a specific state. What makes this even less attractive is the fact that the state space in HoN is not a fixed set.

Thus, we will once more get inspiration from Tesauro (1995), where the author uses *self-play* to initially train his agent. *Self-play* is the concept of letting the agent play against itself. While self-play may not lead to teaching the agent how to act in all situations (for instance, a human might play in ways the AI code never considers), this allows the network to establish a basic understanding of the game. Once the network has established this, it can be trained further on other opponents.

7.5 Evaluation

Evaluation will happen in two phases. First, we will evaluate the network's correct prediction rate during self-play. That is, we look for an indication that the network predicts the agent to win when it actually wins, and to loose when it looses. We essentially want to find proof that our agent is learning. These experiments are detailed in Section 8.4 and Section 8.5.

In the second phase we will compare the agent to the benchmark opponent, namely, the HoN's official bots. This will establish how valuable learn-able agents are to the MOBA game genre. We will measure the agent's performance on a variety of factors, but ultimately look at it's win percent in respect to the HoN bot. The discussion of this occurs in Section 8.6.

8 Experiments

This Chapter will document the most important experiments that were made to design and evaluate the learning agent.

8.1 Limitations

In the introduction we touched on the reasoning behind picking Heroes of Newerth as our testing platform. We now need to discuss a couple of limitations encountered during the progress of this thesis.

- **Code Framework**
  HoN allows changing any bot related code when playing on one's own computer. However, it limits the programmer to the pre-defined code base and doesn't allow for the extension of external sources. For instance, there exist many machine learning frameworks, however, as these come in the form of third-party libraries, they cannot be accessed (or imported) from the HoN code base. Thus, for this thesis back-propagation and TD(λ) had to be implemented first, before being able to start coding the learning agent.
  Furthermore, one is additionally limited by fact that all code must be written in lua. Lua is a scripting language, and thus programs often suffer in performance when compared to other languages such as C. Typically there are options as to improve lua performance such as the JIT compiler, or implementing calculation intensive parts in C and calling these from the lua script. But, as before, these options are not available from within the HoN framework.

- **Hardware Demands**
  HoN was released in 2012, and is therefore a relatively new game with corresponding graphic demands. It is possible to lower the graphics (which was done), but it still demands a fair amount. For our game setup, matches typically take between 20 and 120 seconds. This value varies depending on the parameters of the learning agent, as well as the learning stage. Furthermore, after every match the environment needs to be "reset" to the initial state, which also can take 10-30 seconds.
  HoN allows games to be sped up by a factor of 0-100. However, this speedup is limited by the computing power available. A realistic

speedup is around 9-10 times. However, even then there occasionally can occur a couple of performance drops, e.g. where a match isn't sped up but instead runs in real time. Trivially, this limits the number of tests that can be run in the time available.
Table 8.1 lists the specifications of the computer used for the experiments.

- **Real-Time Gameplay**
  The various environment states within matches cannot be saved, such that a game can be rewound and restarted from a specific state. Thus, when a match begins, it is played until the end. The agent can only learn from the sequence of actions it has taken and the states it has visited in that specific match.

- **Data Collection**
  This also presented a challenge. HoN doesn't offer any of the usual input-output support. That is, one cannot write game data to a simple text file. Instead, they provide their own database that can store information. However, these database files as these aren't meant to be accessed from outside of the game, thus the data within is not easy to retrieve. Hence, a program dedicated to getting this data and storing it in unsable formats had to be made as well.

- **Data Sharing**
  Bots can only share data via the HoN database files. Usually, self-play involves both agents to access and modify the same network values. This is done to increase the speed at which the agent learns. However, this could not be implemented effectively. The agents have to update the network values after every action they take; however, when this was tried during the tests, the data was corrupted more often than not. Thus, another attempt was made in which one agent updates the network after each action, and the other instead saves all states and actions it visited during the match. Once the match was over the second agent updated the network in a batch-like fashion. This worked in the sense that the data remained intact, but proved to be very inefficient in terms of performance, that is, the batch-updates would sometimes take minutes between matches.
  Thus, the solution was to have both agents use the same AI logic, but each maintain their own neural network.

- **Limited Generalization**
  As stated, the hardware demands of HoN lead to the tests taking their time. Thus, within the scope of this thesis, the agent code cannot be tested on more than a few selected heroes.

CPU	Intel(R) Core(TM) i7-4770K CPU @3.50GHz
RAM	16GB
System type	Microsoft Windows 10, 64-bit
GPU	GeForce GTX 1060 6GB

Table 8.1: Specifications of the Desktop Computer used for the experiments.

8.2 Custom Maps

As we discussed in Chapter 2, the most played map in Heroes of Newerth is Forest of Calvadar. However, as the plan is to develop a learning agent for the laning phase only, it isn't necessary to use the entire map as testing grounds.

Heroes of Newerth allows the creation of custom maps, thus, the graphical requirements needed to execute tests could be drastically reduced by designing our own maps, which are a simpler and smaller version of the original map. For instance, for the laning phase only a rectangle-like area is required, with an optional tier one tower for each team. For simplicity, we assume that agents cannot return to their base to heal, as this could lead to potentially never ending games. Furthermore, we exclude all landscape obstructions, to achieve equal win conditions on both sides.

In Section 8.6.1 we will describe the map used for the final tests in more detail.

8.3 State Representation & Actions

Let's shortly take a look at the design of our final agent; that is, a state's features as well as the set of actions. While some of the features remained the same throughout the development progress, a great deal was first determined after extensive experiments. However, to give the reader an idea of the scope of a state, we will start out by presenting our agent's structure.

The first question we must ask us is what actually matters in a laning phase? Ultimately, a hero is more valuable to a team when he levels faster or gains more gold than his lane opponent. The hero does this by either killing creeps, killing the opponent, or killing the enemy tower. We seek to model a state's features such that they answer the agent's following questions:

- *How healthy am I? How healthy is my opponent?*
  This is important because the agent needs to know how close it is to loosing a match by dying; and hence, if it should attack or retreat.

- *How dangerous am I? How dangerous is my opponent?*
  Health is not everything; if the opponent has no abilities left to use, then the agent may be in no immediate danger even if the agents health is low.

- *How dangerous is the environment for me? How dangerous is the environment for my opponent?*
  Towers and creeps are dangerous as well and taking damage from them should not be ignored.

- *Where am I current? Where is my opponent?*
  This is important, because it tells the agent how far away it is from the various objects in the game, as well as the opponent itself. This be a deciding factor in the agent's play style.

- *Can I kill creeps? Can my opponent kill creeps? What is my creep score? What is my opponent's creep score? When does the next creep wave spawn?*
  While heroes can use abilities on creeps it generally isn't a practice among players; especially within the HoN environment, where the mana costs of abilities are very high. Instead, players auto-attack creeps. Thus, the agent needs an indicator for when a creep has low enough health to kill. Furthermore, it also is important to know how well the agent has been farming in general.

- *Can I destroy a building? Can my building be destroyed?*
  As the ultimate objective in HoN is to destroy the enemy's base, it is very important to destroy other defensive structures on the map. Furthermore, apart from the strategic advantage this provides, all heroes within a team obtain gold when an enemy tower is destroyed.

Features that are independent from the hero are listed in Table 8.2. The health, mana and distance related features attempt to model the status of the heroes, as well as the teams' other units. The remaining ones hint at game mechanisms; for instance, whether the hero is able to last hit or deny a creep, when the next spawn of a creep wave is, or if a hero is taking damage from enemy creeps or the opponent's tower.

Features that model the hero's abilities are shown in Table 8.3. These indicate whether an ability is available, and if not how long it will take to become available again. Furthermore, the possible effects of the abilities (buffs or debuffs) are represented as well.

Finally, we should look at the set of actions, which are detailed in Table 8.4. Trivially, we need one action per ability, and if an ability can be used on either a friendly target or an enemy target, then we need two actions per ability. We need the agent to be able to run toward and away from its opponent, and to stand still and wait. Furthermore, we need it to be able to attack all enemy units (heroes, creeps and towers).

Let it be noted that all agents will be trained at a starting level 16, as this allows the heroes to have a sufficient health pool to be able to not die to randomness. Additionally, all abilities of the heroes are leveled to max, such that there is no imbalance between them. Items are excluded from use.

General Input		
Name	***Range***	
Health	$[-1, 1]$	Health percent of hero; -1 at 0%, 1 at 100%, $\forall$*heroes*
Mana	$[-1, 1]$	Mana percent of hero; -1 at 0%, 1 at 100%, $\forall$*heroes*
Auto-Attack Range	$\{-1, 1\}$	1 if hero is in AA-range of opponent, -1 else; $\forall$*heroes* if and only if both heroes aren't the same type
Auto-Attack Ready	$\{-1, 1\}$	1 if auto attack is ready, -1 if it is on cooldown, $\forall$*heroes*
Distance (between heroes)	$[-1, 1]$	-1 when heroes are right on top of each other, 1 when heroes are at maximum possible distance from each other
Distance (to center)	$[-1, 1]$	-1 if at the center coordinate, 1 if far away from it; for both coordinates (x and y), $\forall$*heroes*
Creep Score	$[-1, 1]$	-1 if hero has not killed any creeps, 1 if it has kill all the creeps it needs to win, $\forall$*heroes*
Creep Aggro	$\{-1, 1\}$	if hero takes damage from enemy creeps; -1 if hero has no aggro, 1 if hero has aggro, $\forall$*heroes*
Creep Health	$[-1, 1]$	average health of creep wave; 1 if all have full health, -1 if all are dead, $\forall$*teams*
Creep Wave	$[-1, 1]$	time until spawn of next creep wave, -1 when creep waves spawns, 1 when creep wave spawn is maximum duration away
Last Hit	$\{-1, 1\}$	1 if hero can kill an enemy creep by single auto-attack, -1 else
Deny	$\{-1, 1\}$	1 if hero can kill a friendly creep by single auto-attack, -1 else
Tower Range	$\{-1, 0, 1\}$	-1 if hero is out of range of enemy tower, 0 if hero is in tower range but has no aggro, 1 if it is in tower range and has aggro, $\forall$*heroes*
Tower Health	$[-1, 1]$	health percent of tower, -1 at 0%, 1 at 100%; $\forall$*teams*

Table 8.2: Features related to general information.

Ability ***Name***	**Input** ***Range***	
Cooldown	$[-1, 1]$	-1 if ability was just used, 1 if ability is available; ∀abilities∀heroes
Range	$\{-1, 1\}$	1 if hero is in ability-range of opponent, -1 else; ∀abilities, ∀heroes if and only if both heroes aren't the same type
Buff/Debuff	$[-1, 1]$	Duration of Buff/Debuff effect; -1 if no buff/debuff, 1 if buff/debuff was just applied

Table 8.3: Features related to hero abilities.

Actions	
Pursue	move towards the enemy hero
Flee	move away from the enemy hero
Hold	don't do anything and stand still
Auto Attack Enemy Hero	hit the opponent
Auto Attack Creep	hit a creep; priority order: last hit enemy creep, deny friendly creep, hit random enemy creep
Auto Attack Enemy Tower	hit the enemy tower
Ability	use an ability (on enemy hero), ∀abilities

Table 8.4: The set of actions.

8.4 Stage 1: Basics

In this section we will look at the early states of the experiments in which everything, from state features to network refinement to parameter settings, had a big question mark attached to them. However, these steps were important in finding an appropriate solution for our learning agent.

8.4.1 The Battle Arena

Laning requires a player to think about multiple challenges at the same time, such as battling the opponent, killing creeps, and deciding when to push forward or fall back. Thus, instead of trying to model an agent that is capable of doing all of the above, it was instead decided to break the problem into smaller parts at first. Thus, a map (our "Battle Arena") was created that is a simple square, in which only the two opposing heroes exist.

8.4.2 Experiment 1: Different Heroes

Typically, a game consists of ten different heroes, that is, no two players can play the same heroes. Thus, the first tests included two different heroes (*Predator* and *Swiftblade*) that were thought of to be about equally matched, and relatively simple to play.

The network input layer included following features

- health & mana, distance between heroes, auto-attack range, range & cooldowns for abilities, buffs & debuffs (as described in Table 8.2)
- for both heroes four boolean inputs that indicate whether a it is close to the four borders of the arena
- one boolean input for each action

The set of actions included two abilities for each hero (the other abilities were passive), the auto-attack and nine movement related actions (hold, and movement into the eight directions: north, north-east, east, south-east, south, south-west, west, north-west). This left us with a total of 41 input units, 20 hidden units and 1 output unit. The parameters took following values: $\alpha = 0.2$, $\gamma = 1$, and $\varepsilon = 1$, which was decreased by 0.0001 after each time step.

λ	**Predator**	**Swiftblade**	Match Length
0.95	22.38%	77.62%	12.5s
0.75	20.35%	79.65%	11.6s
0.65	22.76%	77.24%	11.5s
0.35	23.54%	76.46%	11.2s
0.15	16.74%	83.26%	13.5s

Table 8.5: Win percent and average match length of the Predator vs Swiftblade setup.

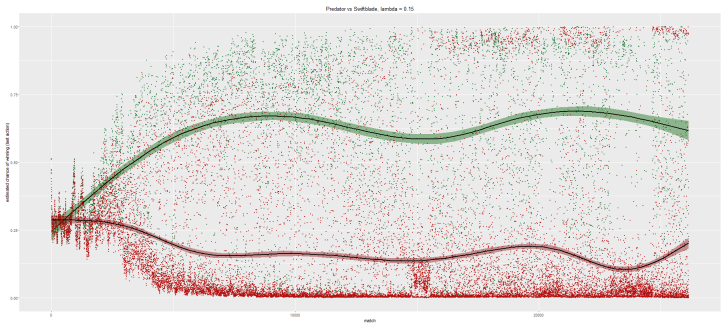
The win percentages of the agents as well as the average match length for various values of λ are shown in Table 8.5. Plots for the most interesting setups are shown in Figures 8.1 and 8.2. The points indicate the outcome prediction for the last action the agent took in the game, before the match ended in a terminal state.

Both the table and the plots clearly indicate that the Swiftblade agent seems to have a huge advantage. In fact, within the MOBA community it is well-known that no two heroes are equally balanced. Each hero has its strengths and weaknesses, and brings different utilities to the team. Thus, most MOBA games continuously put out patches that rebalance the state of the game.

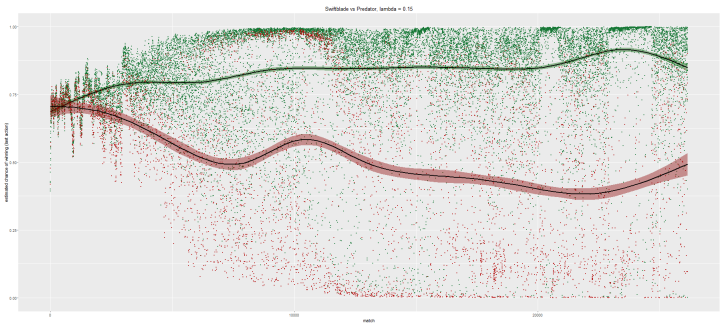
Another thing the graphs tell us is that the network requires relatively few games ($\approx$ 10000) to move from its initial random prediction to a more decisive evaluation of the game. However, it also indicates that when a hero enters a trend (e.g. wins a lot or looses a lot), the opposite result isn't predicted with such accuracy. For instance, Predator's evaluation of loosing states quickly approaches 0, whereas its prediction for wins seems to stagnate around 0.8 or lower. Similarly, Swiftblade's evaluation behave the opposite way. This seems to be caused by the fact that Predator cannot seem to find a reliable strategy to beat Swiftblade, and Swiftblade seems to fast find its optimal strategy.

Thus, it became clear that to be able to truly evaluate the learning curve of an agent, the opposing heroes had to be the same. We will continue this line of thinking in the next section.

(a) $\lambda = 0.15, \gamma = 1, \alpha = 0.2, \approx 30000$ games

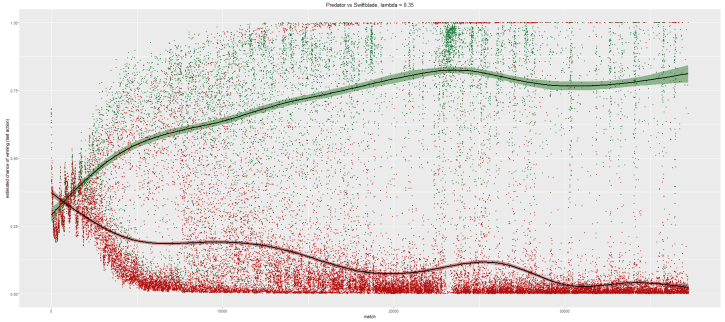


(b) Predator (Team 1)

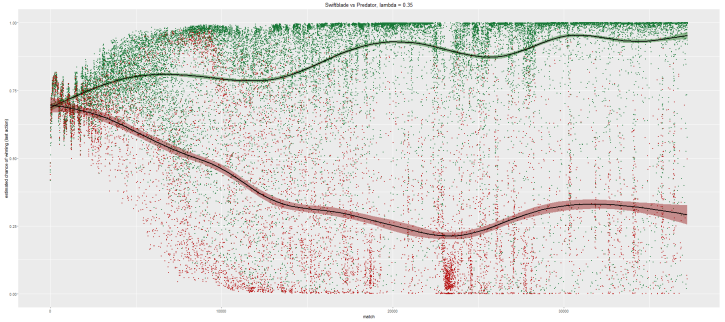


(c) Swiftblade (Team 2)

(d) $\lambda = 0.35, \gamma = 1, \alpha = 0.2, \approx 35000$ games



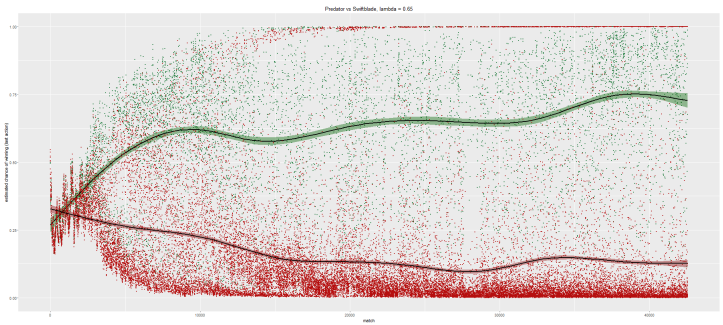
(e) Predator (Team 1)



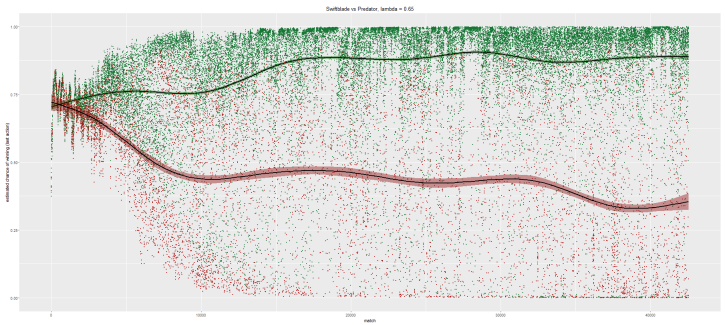
(f) Swiftblade (Team 2)

Figure 8.1: Predator vs Swiftblade setup, x: match number, y: neural network's estimate. Plots the agent's evaluation of its last action before a terminal state was reached; red dots indicate losses, green dots indicate wins; the corresponding lines indicate the trend.

(a) $\lambda = 0.65, \gamma = 1, \alpha = 0.2, \approx 45000$ games

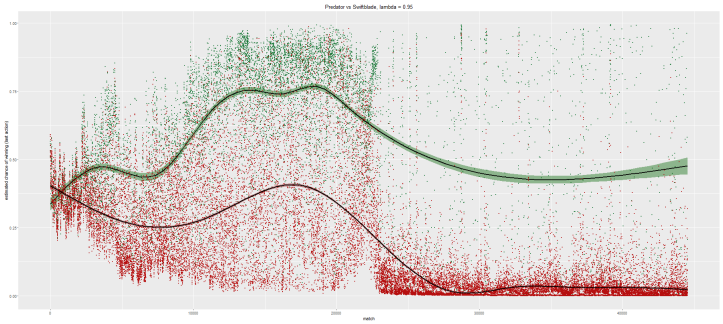


(b) Predator (Team 1)

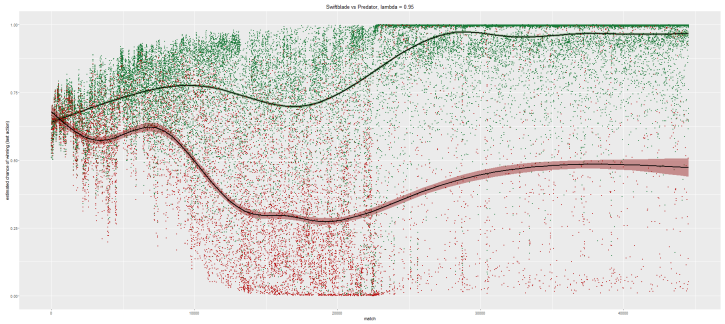


(c) Swiftblade (Team 2)

(d) $\lambda = 0.95, \gamma = 1, \alpha = 0.2, \approx 45000$ games



(e) Predator (Team 1)



(f) Swiftblade (Team 2)

Figure 8.2: Predator vs Swiftblade setup, x: match number, y: neural network's estimate. Plots the agent's evaluation of its last action before a terminal state was reached; red dots indicate losses, green dots indicate wins; the corresponding lines indicate the trend.

λ	**Predator1**	**Predator2**	**Match Length**
0.75	51.43%	48.57%	23.1s
0.35	55.3%	44.7%	22.2s

Table 8.6: Win percent and average match length of the Predator vs Predator setup that has actions as input features.

8.4.3 Experiment 2: Different Value Function

The next experiment uses the same hero, Predator, on both ends of the spectrum, and the neural network setup from Experiment 1. The objective was to determine whether it was more beneficial to have actions input to the network as features, such that the network approximates $\mathcal{V}_t$, or to have one output unit per action, such that the network estimates $\mathcal{Q}_t$.

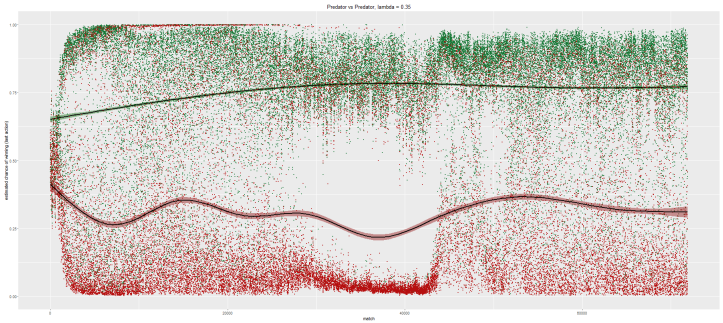
What became immediately clear from watching the agents battle, and which can also be seen by comparing the match lengths of Table 8.6 and Table 8.7, is that the agents that tried to approximate the action-value seemed a lot more "indecisive". Whenever the agents were confronted with a state that didn't relate to anything they had encountered before, it seemed like the agents engaged in random behavior because the weights leading to the output units hadn't been configured appropriately yet.

Instinctively, when we have an output unit for each action-value, we put a greater importance on the actions themselves, as they are regulated by the weights leading from the hidden units to the output units. However, when we try to find the next action to take by flipping the action features "on or off" in the input layer (setting them to -1 or 1), we set the actions equal to the rest of the features, and thus give more control to the network itself. In accordance, we can observe by comparing Figure 8.3 and Figure 8.4, that using actions as input units seems to result in clustering the predictions more closely to the target values (0 for losses, 1 for wins), whereas having an output neuron per action-value seems to create more wide spread estimates; this is especially visible in the predictions made for losses.

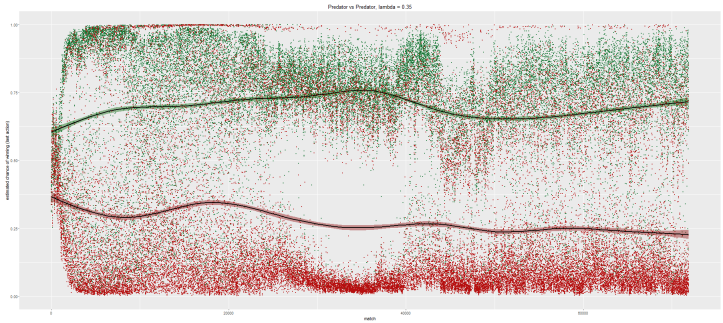
8.5 Stage 2: Learning

During a set of intermediate experiments that are not worth mentioning here, it was confirmed that the fewer inputs features make training a lot easier. Also, a new more efficient way of representing the heroes position's was de-

(a) $\lambda = 0.35, \gamma = 1, \alpha = 0.2, \approx 65000$ games

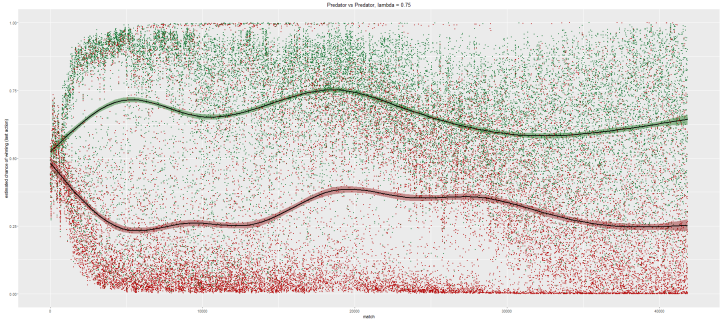


(b) Predator (Team 1)

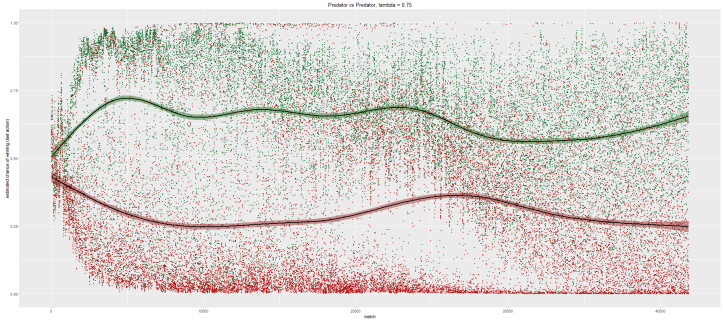


(c) Predator (Team 2)

(d) $\lambda = 0.75, \gamma = 1, \alpha = 0.2, \approx 40000$ games



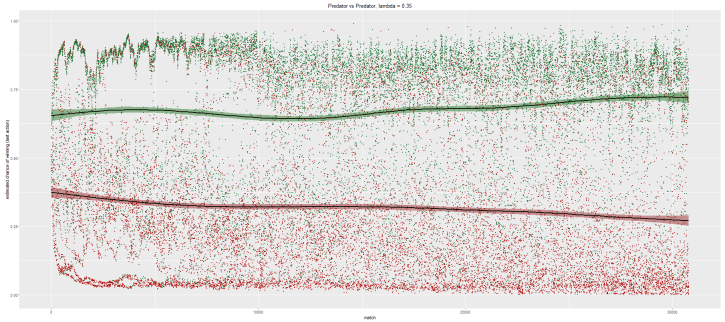
(e) Predator (Team 1)



(f) Predator (Team 2)

Figure 8.3: Predator vs Predator setup that uses actions as input features, x: match number, y: neural network's estimate. Plots the agent's evaluation of its last action before a terminal state was reached; red dots indicate losses, green dots indicate wins; the corresponding lines indicate the trend.

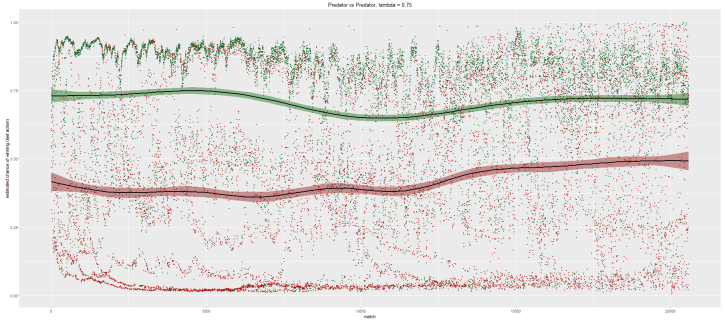
(a) $\lambda = 0.35, \gamma = 1, \alpha = 0.2, \approx 30000$ games



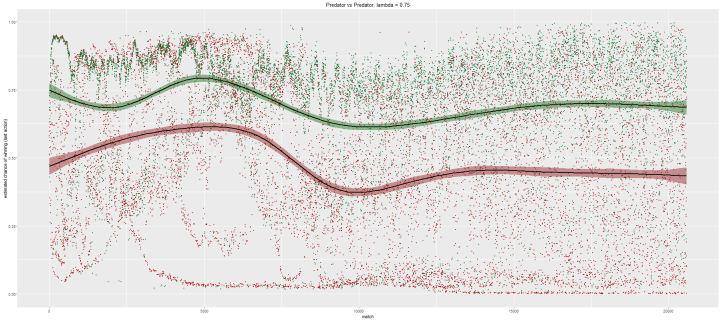
(b) Predator (Team 1)

(c) Swiftblade (Team 2)

(d) $\lambda = 0.75, \gamma = 1, \alpha = 0.2, \approx 25000$ games



(e) Predator (Team 1)



(f) Predator (Team 2)

Figure 8.4: Predator vs Predator setup that has action-values as output units, x: match number, y: neural network's estimate. Plots the agent's evaluation of its last action before a terminal state was reached; red dots indicate losses, green dots indicate wins; the corresponding lines indicate the trend.

λ	**Predator1**	**Predator2**	**Match Length**
0.75	50.4%	49.6%	46.3s
0.35	50.5%	49.5%	39.4s

Table 8.7: Win percent and average match length of the Predator vs Predator setup that has actions as output units.

termined. This is in accordance to what we listed in Table 8.2. Additionally, the learning rate α was increased from 0.2 to 0.3, as it allowed the training to be sped up by a bit, while not over-fitting the network.

Also, in the start of training, when agents are guided by random actions, we limited the way the action $flee$ could be used; namely, only when the hero was being attacked by the enemy. Otherwise, initial training would simply take too long and make games take unnecessarily long. Once training had progressed enough, we removed this restriction again.

Furthermore, a new hero was selected for the learning agent. One of the reasons behind choosing the initial two heroes, Predator and Swiftblade, was that they were very simply heroes, and thus allowed us to get a feel for the challenge at hand. However, now that we have established a baseline for training, it is time to move on to a more advanced hero; Succubus.

In this section, we seek to show the reader that our network does indeed learn.

8.5.1 Experiment 3: Weight Updates

This experiment uses 39 input units, 15 hidden units and 1 output unit. This includes all the concepts of Table 8.2 except the features related to towers or creeps. Furthermore, in this test we diversify the rewards agents can obtain at the end of a match. The rewards for loss, draw and win are $\{0, 0.5, 1\}$ for agent Succubus 1, and $\{-1, 0, 1\}$ for agent Succubus 2, respectively. This is is done in an attempt to explore whether punishing losses (and draws) more harshly will result more "positive" behavior by the agent.

We evaluate the learning curve by first selecting 10000 states from a set of 10000 games that were generated through random walks, that is, matches in which the agents only choose random actions. Then, we use a network's weight before training, during training (at the half-time point), and after training is finished. The Figures 8.5, 8.6 and 8.7 illustrate the experiment results.

The first conclusion that we can draw, is that an increased negative reward does not necessarily lead to more positive behavior, but instead improves the prediction rate of losses. We can also observe that the evaluating the same states using different weights supports our expectation: at the start the weights will have no correlation to the states, whereas as training continues, more and more estimates approach 0 and 1. Furthermore, we can see that the weights' approximations behave in accordance to the training at that time. Consider, for instance, Figure 8.6. At the half-way point of training, we're at an extreme point for loss prediction (very close to 0), and at a high point for win prediction (≈ 0.8). In the corresponding figure to the right, we can see that at half-point the weights mostly predict values very close to 0 or 1. After training finishes, at which point the win predictions are at about ≈ 0.75 and the loss predictions have risen to ≈ 0.45, we can see that most blue dots are clustered at the right side of the section (close to one), and that a notable number of blue points is situated around 0.4.

Now that we have established that our network does indeed learn, we can execute our first benchmark test. The next section discusses this experiment.

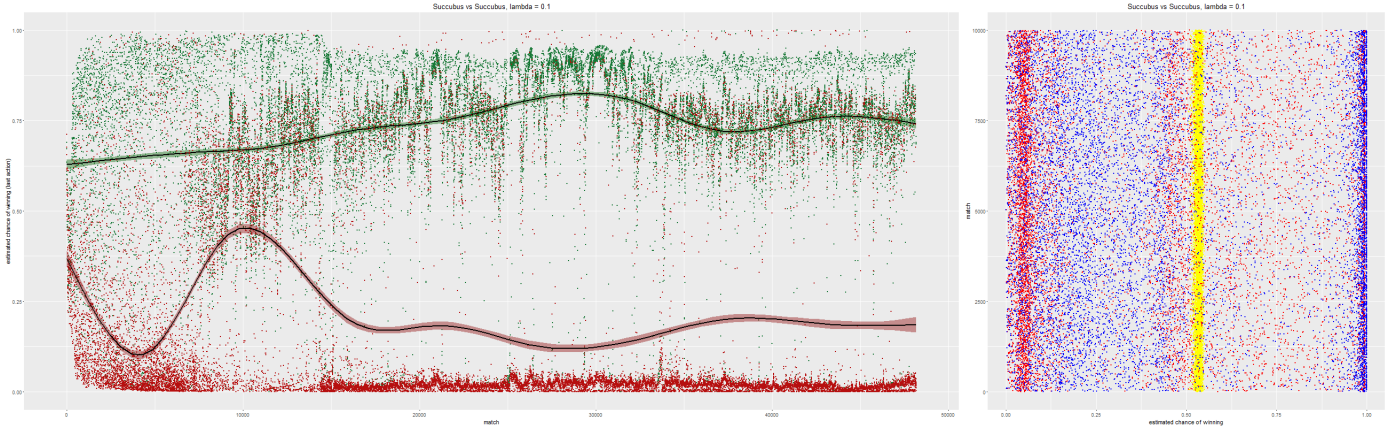
8.5.2 Experiment 4: First Benchmark Test

In this experiment we test for the first time if our agent model can learn when playing against the in-game HoN bot.

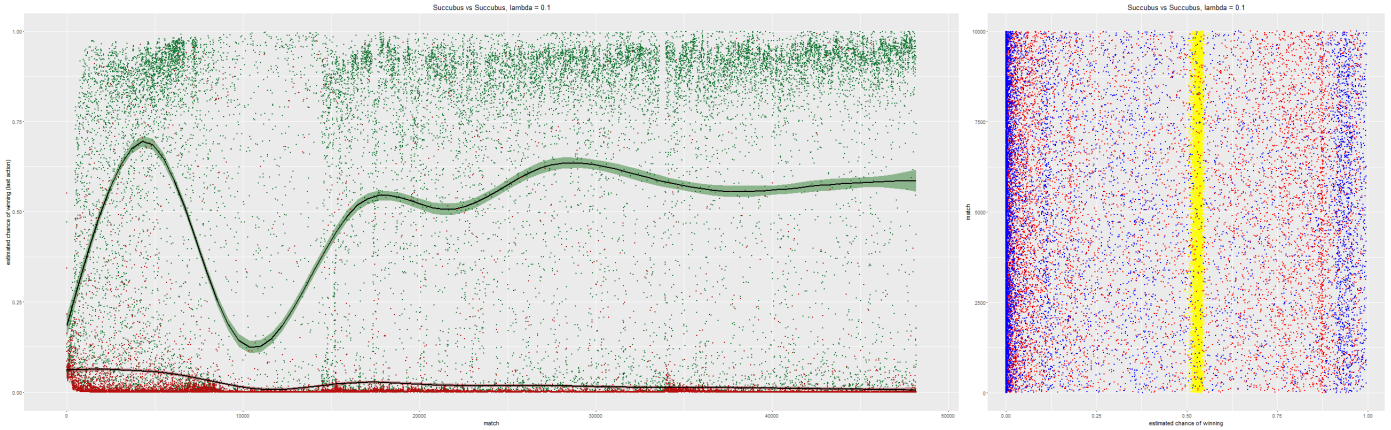
Beforehand it has to be mentioned that in order to use the HoN bot, it had to be stripped of all of its behavior that isn't related to fighting. Essentially, the only behavior that were kept were `HarrassHero` and `RetreatFromHero`. Furthermore, it had to be made more aggressive by default, as it otherwise attempted to do nothing else but run away. Lastly, the `RetreatFromHero` behavior was also modified slightly to run directly in the opposite direction of our agent. Both of these modifications were nessecary, as the HoN bots typically tend to play very save and want to retreat to a nearby structure; which in this scenario doesn't exist.

With this test we ultimately were only interested in whether our agent could learn a better sequence of actions than the official bot.

We used the same network structure as the last section. Our agent was put into the arena with the HoN bot, and no pre-trained weights. It thus had to learn from scratch. Table 8.8 and Figure 8.8 show the bots performance. In the initial learning stage our agent gets beaten in about 9 out of 10 matches.

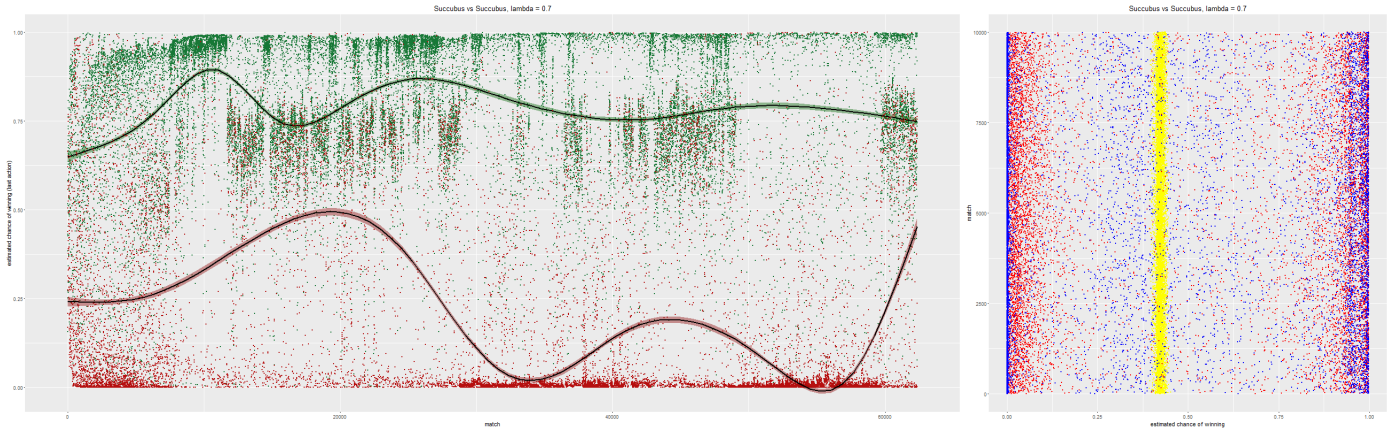


(a) Succubus (Team 1)

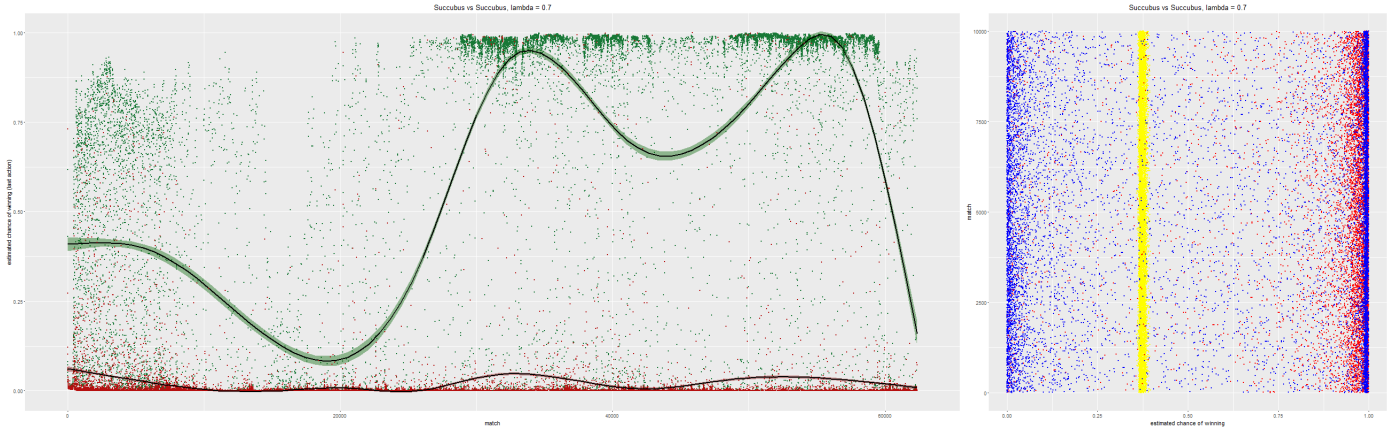


(b) Succubus (Team 2)

Figure 8.5: Succubus vs Succubus. $\lambda = 0.1, \alpha = 0.3, \gamma = 1$. The left plots the agent's evaluation of its last action before a terminal state was reached; where x: match number, y: neural network's estimate; red dots indicate losses, green dots indicate wins; the corresponding lines indicate the trend. The right plots show network evaluation of 10000 random states, where x: neural network's estimate of given state, y: the state; yellow indicates weights before training, red indicates weights at half-time of training, blue indicates weights at end of training.

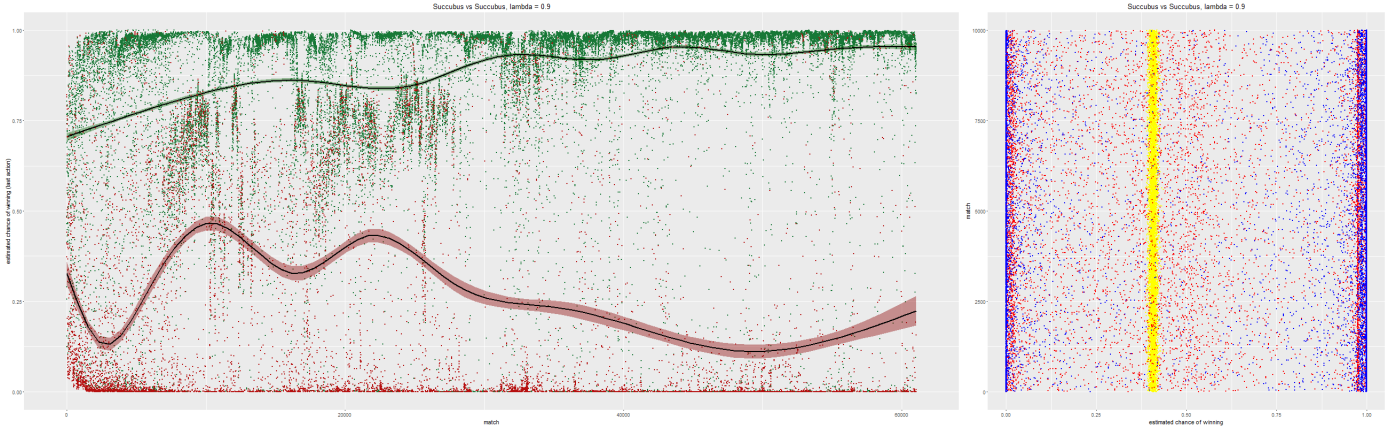


(a) Succubus (Team 1)

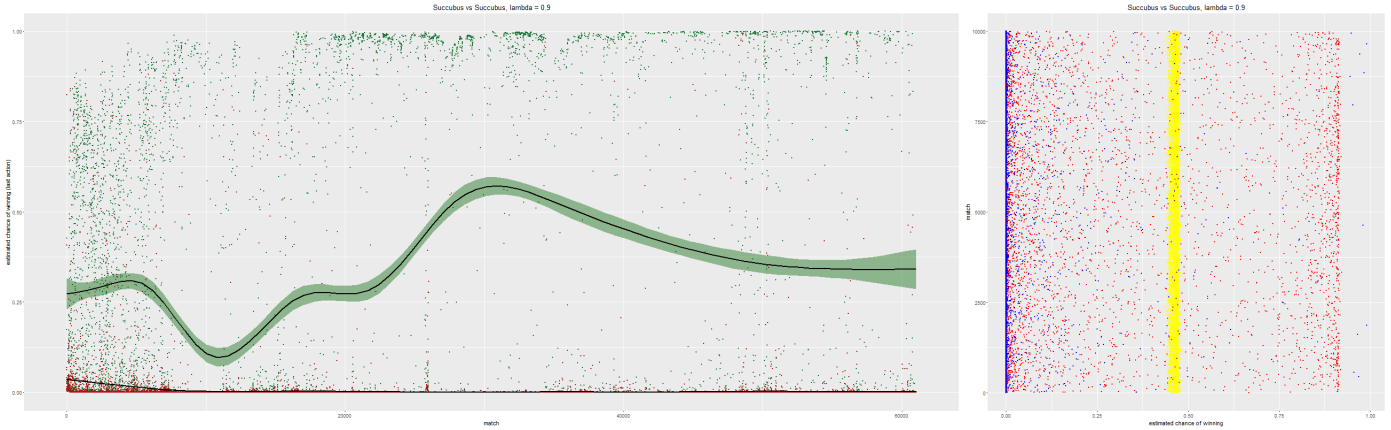


(b) Succubus (Team 2)

Figure 8.6: Succubus vs Succubus. $\lambda = 0.7, \alpha = 0.3, \gamma = 1$. The left plots the agent's evaluation of its last action before a terminal state was reached; where x: match number, y: neural network's estimate; red dots indicate losses, green dots indicate wins; the corresponding lines indicate the trend. The right plots show network evaluation of 10000 random states, where x: neural network's estimate of given state, y: the state; yellow indicates weights before training, red indicates weights at half-time of training, blue indicates weights at end of training.



(a) Succubus (Team 1)



(b) Succubus (Team 2)

Figure 8.7: Succubus vs Succubus. $\lambda = 0.9, \alpha = 0.3, \gamma = 1$. The left plots the agent's evaluation of its last action before a terminal state was reached; where x: match number, y: neural network's estimate; red dots indicate losses, green dots indicate wins; the corresponding lines indicate the trend. The right plots show network evaluation of 10000 random states, where x: neural network's estimate of given state, y: the state; yellow indicates weights before training, red indicates weights at half-time of training, blue indicates weights at end of training.

Match Range	Succubus Agent	Match Length
1 - 1817	10.5%	59.7s
1818 - 3634	43.4%	60.6s
3635 - 5451	73.0%	56.6s
5452 - 7268	59.3%	46.0s
7269 - 9085	71.2%	42.9s
9086 - 10902	81.1%	41.4s
10903 - 12719	83.3%	41.4s
12720 - 14543	84.7%	41.8s

Table 8.8: Win percent for different stages of learning, and average match length of the Succubus vs HoN bot setup.

It then slowly manages to improve it's behavior, which by the end leads to a win percent of 84%.

With these results we can continue on to the final stage of experiments.

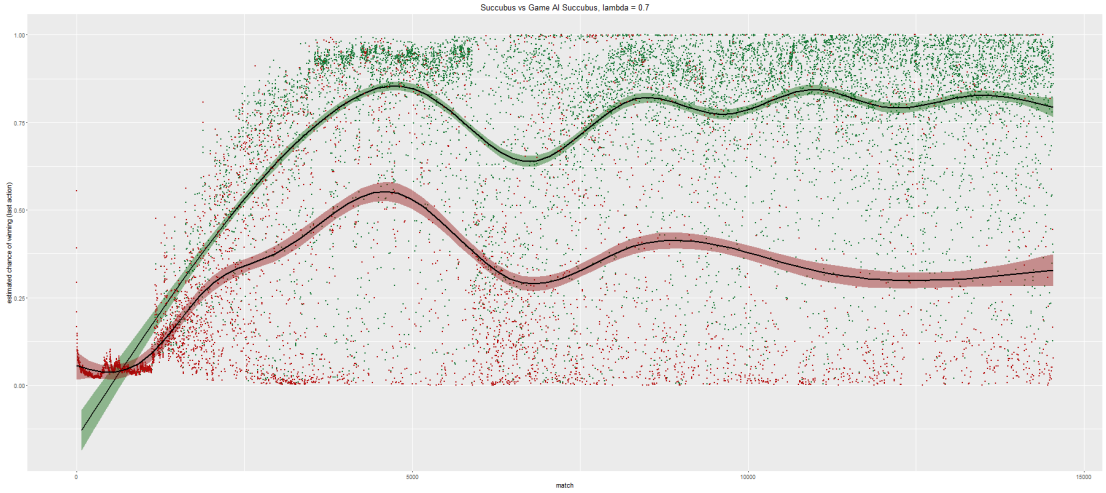


Figure 8.8: Succubus vs HoN bot, x: match number, y: neural network's estimate. Plots the agent's evaluation of its last action before a terminal state was reached; red dots indicate losses, green dots indicate wins; the corresponding lines indicate the trend.

8.6 Stage 3: Extensions

Earlier, we limited our agent to simple one-vs.-one combat without any other game mechanics. However, this is not a sufficient test. First of all, the game AI is not suited for merely combat, thus testing our learning agent against the official one doesn't seem like a good enough quality test. Furthermore, this thesis seeks to make a direct connection to the MOBA genre; that is, the goal is to model an agent that could actually be used (abide with a few modifications) within real matches.

8.6.1 Game Setup

Within this final test phase, the earlier setup of our battle arena is extended to a proper laning phase scenario, as displayed in Figure 8.9. A tower for each team is introduced, as well as the spawn of creep waves. The distances from one tower to the other as well as the minion's walk paths are designed to resemble HoN's official map, however, in such a way, that neither side has an advantage.

Lets shortly look at the gameplay of this scenario. Every 30 seconds, and starting at match begin, a creep wave spawns shortly behind the team's tower. It then travels, as it would in a real HoN match, to the center of the map, where it engages in a battle with the enemy's creeps. The agents spawn at 4 seconds into the game, shortly in front of the tower. This allows the creeps to have traveled most of the distance to the center. This is how a normal laning phase would happen as well: a creep wave meets the other, followed by the lane's heroes. The time-line of these game events are shown in Table 8.9.

The agents now obtain the full state representation and action set, that we discussed in Section **??**. This translates to 56 input units (11 of which are actions), 25 hidden units and one output unit.

Finally, all that remains is to define the win conditions of a game, which are

- the opponent dies,
- the agent reaches 15 creep kills before the opponent,
- the enemy tower is destroyed.

Match Time (*in seconds*)	**Game Event**
0	• Game Start • Spawn of first Creep Wave
4	Heroes Spawn
5	Creep Wave reaches Map Center
30i	Spawn of Creep Wave number $i+1$, for $i = 1, 2, 3, ...$
35i	Creep Wave $i+1$ reaches Map Center, for $i = 1, 2, 3, ...$
$\mathcal{M}$	Match End due to reaching a terminal state

Table 8.9: Time-line of unchanging events occurring within every match.

This opens the experiment to develop in several ways, and opens up for a multitude of strategies. A hero can either play aggressive and attempt to kill the opponent, or it can play defensive and kill and deny creeps, or it can push creep waves and destroy the tower.

8.6.2 Experiment 5: Training in Extended Environment

The first step is to again let our agent train by playing against itself. The results are described in Table 8.10. The table lists the overall win percent, as well as the win percent of the agent by killing a tower or the opponent. Winning by killing enough creeps was omitted as this never occurred during these tests. As can be observed, the bots typically win by killing the opponent.

The plots for the varying λ values are shown in Figures 8.10, 8.11 and 8.12. The conclusion we draw from these results (as well as all the ones before) is that $\lambda = 0.9$ provides the most competitive learning agent; as can be seen, neither agent ever manages to completely dominate the other. Instead, they alternatively seem to find strategies to beat the other. We suspect the reason why this is different for the test with $\alpha = 0.9$ and 40 hidden units, is that there simply hasn't been enough training examples. In fact, when observing the agents in this test, they adherently seem very stupid (especially when compared to the $\alpha = 0.9$ test with only 25 hidden units); one side (the loosing one) tries force aggression and continuously pursues the other to their tower, at which the former then dies. Thus, the latter agent

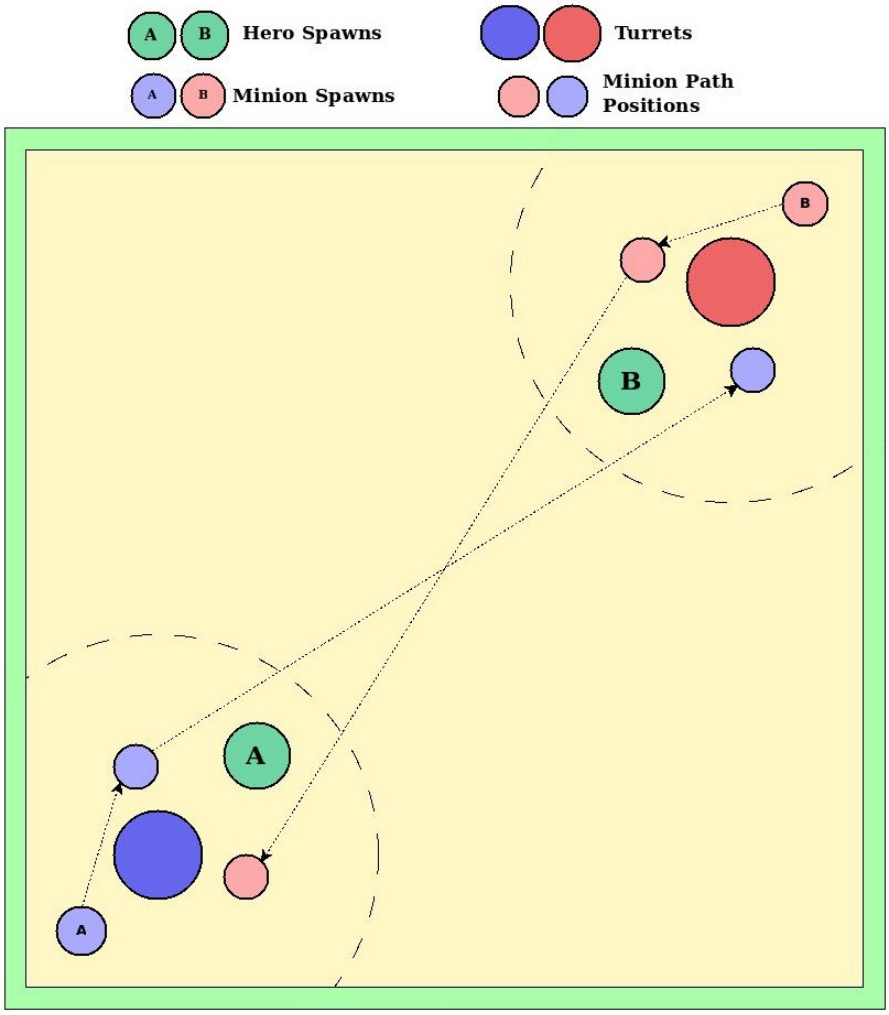


Figure 8.9: The map layout of the game setup for all tests.

λ	**Succubus1**			**Succubus2**			**Match**
	Overall	*Tower*	*Hero*	*Overall*	*Tower*	*Hero*	**Length**
0.9	42.6%	0%	42.6	57.4%	0%	57.4%	26.4s
0.7	53.9%	0.01%	53.84%	46.2%	0%	46.14	38.8s
0.5	66.3%	0.01%	66.3%	33.7%	0.04	33.6%	33.5s
0.3	73.9%	0.02%	73.9%	26.1%	0.02%	26.1%	26.8s
0.1	51.1%	0.03%	51.0%	39.1%	0.02%	39.0%	29.1s
$0.9^{(a)}$	26.6%	0.01%	26.6%	73.4%	0%	73.4%	25.1s

Table 8.10: Performance of the our agent during self-play, where $^{(a)}$ uses an increased number of hidden units (40). Analysis of 1000 games for each case.

(the winning one) has learned to spent its majority of the time by simply waiting within its tower range.

Why the loosing agent doesn't seem to manage to adopt a different strategy is discussed in the next section.

8.6.3 Experiment 6: Second Benchmark Test

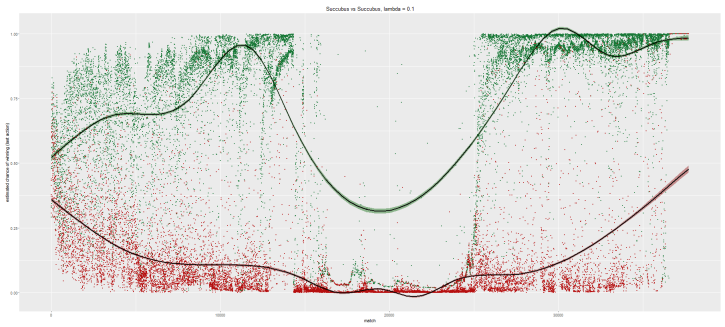
Now, we turn to testing out agent in laning phase scenario against the HoN bot. Thus, we finally don't have to modify the HoN code. It has access to all behaviors apart from `Shop`, `HealAtWell` and the team related ones, as neither shop, nor well nor team mates exist. No alterations to the utility or behavior execution was made.

Table 8.11 summarizes two benchmark tests. *Damage done* indicates the damage our agent inflicted on the opponent, with respect to all hero damage done. *Creep kill* informs about the percentage that our hero farmed better than the opponent. Overall, tower, creep and hero indicate the win percents as defined before.

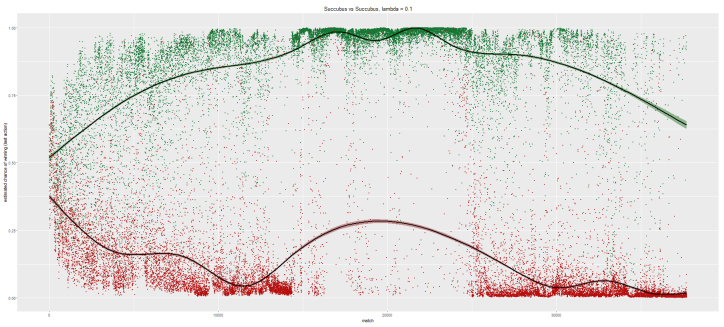
The first benchmark test is executed by using the weights we trained in the previous section for the parameter $\lambda = 0.9$ (and 25 hidden units). Then, out agent plays against the HoN bot for 1000 games. Learning is still enabled, all actions are chosen by using the network to evaluate the game state. The second test is slightly different. The agent is put into the game without any previous training. It is allowed to play again the HoN bot for ≈ 15000 games.

When we look at the win percent achieved by our learning agent, the initial reaction may be to assume that out agent can simply not complete

(a) $\lambda = 0.1, \alpha = 0.3, \gamma = 1, \approx 40000$games

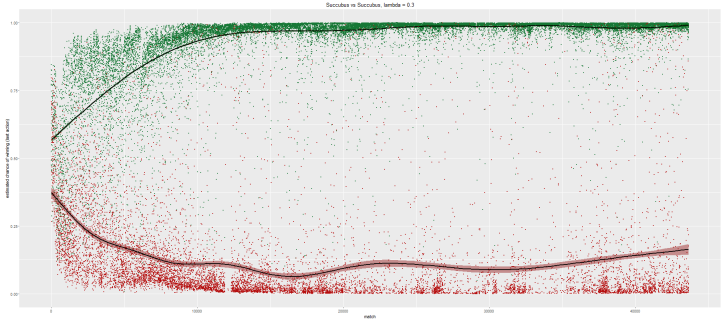


(b) Succubus (Team 1)

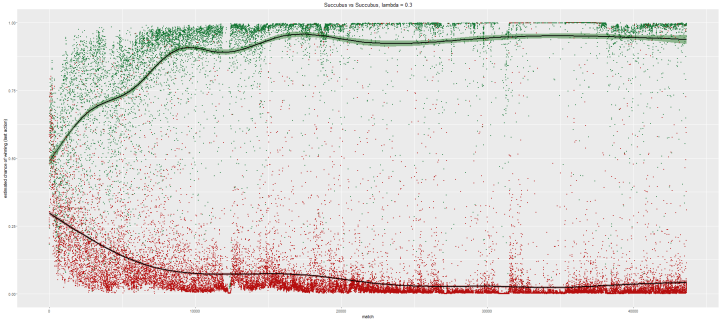


(c) Succubus (Team 2)

(d) $\lambda = 0.3, \alpha = 0.3, \gamma = 1, \approx 45000$games



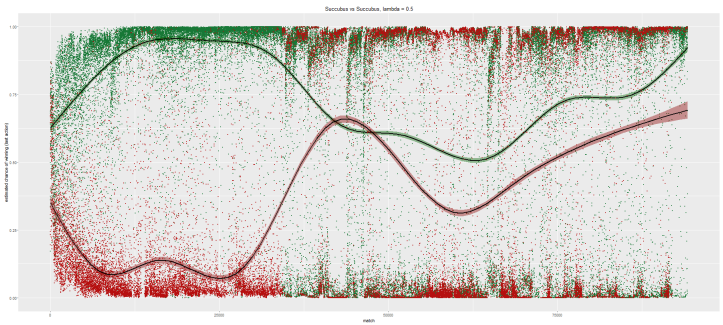
(e) Succubus (Team 1)



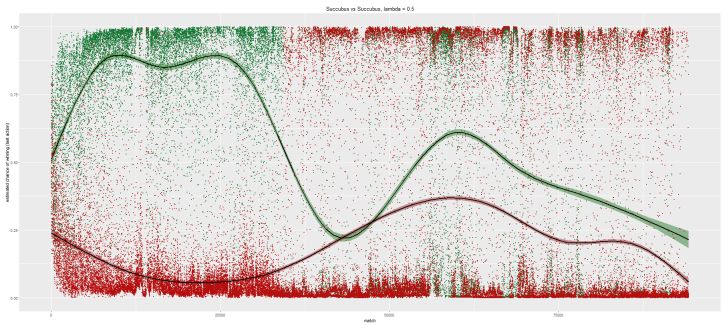
(f) Succubus (Team 2)

Figure 8.10: Succubus vs Succubus, x: match number, y: neural network's estimate. Plots the agent's evaluation of its last action before a terminal state was reached; red dots indicate losses, green dots indicate wins; the corresponding lines indicate the trend.

(a) $\lambda = 0.5, \alpha = 0.3, \gamma = 1, \approx 80000$games

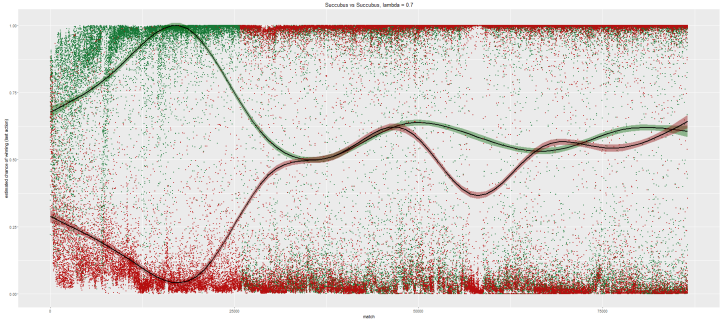


(b) Succubus (Team 1)

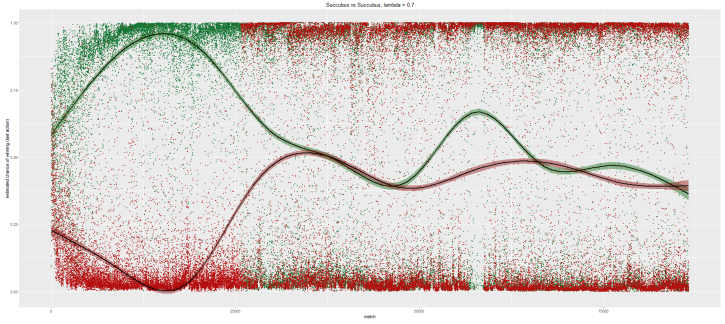


(c) Succubus (Team 2)

(d) $\lambda = 0.7, \alpha = 0.3, \gamma = 1, \approx 80000$games



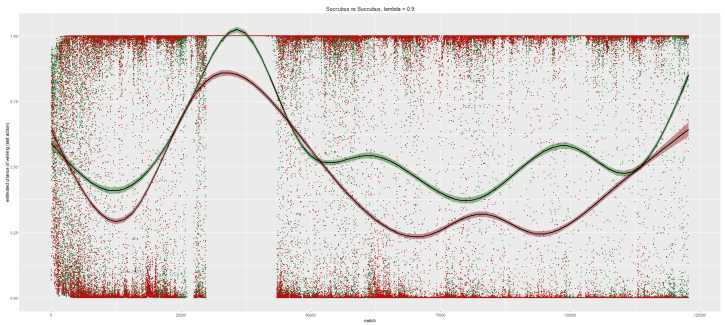
(e) Succubus (Team 1)



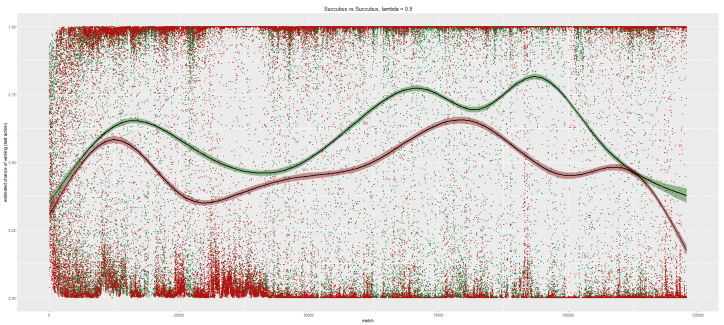
(f) Succubus (Team 2)

Figure 8.11: Succubus vs Succubus, x: match number, y: neural network's estimate. Plots the agent's evaluation of its last action before a terminal state was reached; red dots indicate losses, green dots indicate wins; the corresponding lines indicate the trend.

(a) $\lambda = 0.9, \alpha = 0.3, \gamma = 1, \approx 120000$games

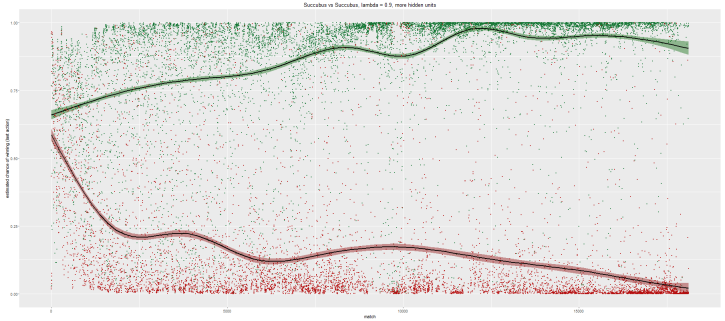


(b) Succubus (Team 1)

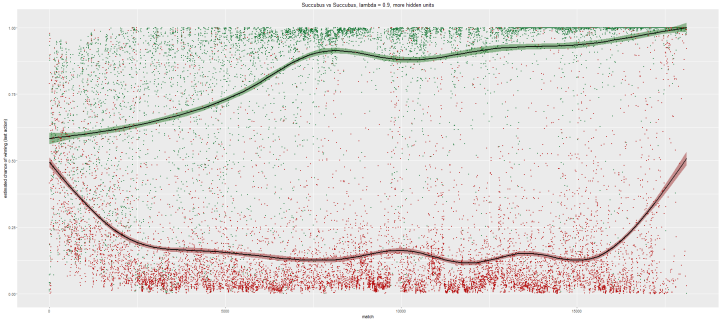


(c) Succubus (Team 2)

(d) $\lambda = 0.9, 40$ hidden units, $\alpha = 0.3, \gamma = 1, \approx 120000$games



(e) Succubus (Team 1)



(f) Succubus (Team 2)

Figure 8.12: Succubus vs Succubus, x: match number, y: neural network's estimate. Plots the agent's evaluation of its last action before a terminal state was reached; red dots indicate losses, green dots indicate wins; the corresponding lines indicate the trend.

with the official AI. However, when actually watching the agents play, the problem immediately becomes clear.

Our agent actually plays better for the majority of a match, but lacks the proper knowledge as to how to actually end it. This is also shown by the damage our agent does in comparison to the HoN bot, as well as the creep kills obtained by (b). Furthermore, most of the time our agent simply dies because it runs ahead of its own minion wave and gets attacked by the enemy minions; or because it tries to reach the opponent and dies to the tower.

The reader may question why the bot doesn't learn that these actions are inappropriate and lead to a loss. The issue has two causes. For one, for such a complex environment that we created here, even the 120000 games that our agent trained aren't enough to properly explore the entire state space. Furthermore, as stated in Tesauro (1995), agents that train exclusively by self-play, may not encounter all possible sequences of moves during training. As the HoN bot was specifically coded by a human, it may do things our agent didn't happen to stumble upon during self-play.

Another reason for why our bot doesn't learn lies in the lack of positive reinforcements. When our agent attacks the HoN bot, the latter essentially just tries to hide under its tower for most of the game. Thus the only other options our bot has, is to either kill the tower, or get enough creep kills. The former is problematic, because our agent has learned to play aggressive during self-play, thus when the hero is close enough to the tower it also is close enough to the enemy hero and will instead alternate to attack it. The second (killing enough creeps) follows a similar train of thought; the bot has rarely encountered a situation during self-play in which it felt it was more appropriate to kill creeps than it was to attack the opponent.

How can we remedy this? There are multiple options. For one, we could generate a set of supervised learning examples, that teach our bot that these behaviors are bad and what to do instead. However, this requires us to specifically know the values the other features can take as well. What most successful applications of learning agents (such as AlphaGo by Silver et al. (2016)) do instead, is that they collect these training examples by recording plays made by professional human players. Ideally this would be what we would want to use to train an agent for MOBA games. However, HoN doesn't support this sort of data collection.

Another options is actually quite simple. Every MOBA player learns very quickly that aggroing the enemy creep wave is a stupid idea, and that running under the opponents tower will inevitably lead to one's death. Thus a simple

λ	**Damage Done**	**Creep Kills**	**Win Percent**				**Match Length**
			Overall	*Tower*	*Creeps*	*Hero*	
$0.9^{(a)}$	86%	-87%	13.6%	0.1%	0%	13.5%	44.6s
$0.9^{(b)}$	83%	57%	6.3%	0.2	0%	6.1%	44.7s

Table 8.11: Performance of the our agent against the in-game AI,where $^{(a)}$ starts with pretrained weights, and $^{(b)}$ starts without pretrained weights. Analysis of 1000 games for $^{(a)}$, and $\approx$ 15000 games for $^{(b)}$

rule that prohibits these actions could fix the problem. Even TD-Gammon (Tesauro (1995)) was later adjusted to include human-defined knowledge.

Hence, we will make the following modification to our agent. We will tell it to position itself behind its own minion wave or to wait for a new wave to spawn in case no creeps are currently alive, and to immediately retreat should it aggro creeps or the enemy tower.

It is important to clarify, that this is not equivalent to hard coding a bots behavior. First of all, we simply restrict the action space in certain situations to contain "fleeing" or "waiting" only. Furthermore, once the bot receives enough positive reinforcements this way, it is assumed that it learns to adopt a winning strategy. Lastly, for the majority of a game, the agent is still required to make decisions without any of our "expert knowledge". Whenever it chooses to pursue or attack the opponent, what abilities to use and in what order, when to attack creeps, when to attack the tower, all these decisions are still entirely up to the approximation of the neural network.

We will see in the next section, how this small change affected our agent's performance.

8.6.4 Experiment 7: Improvements

We again execute two benchmark test. The first, (a), is executed again from the same weights as we used in the previous experiment. It essentially is not allowed to retain any knowledge of the HoN bot, but only is allowed to use what it learned from self player. The second test, (b), works a bit different. The agent uses the weights it learned by playing against the HoN bot. Recall, that it's win percent was an unimpressive 6.3%. It thus never found any good strategy to beat the HoN bot. In both cases, the agent plays 1000 games against the HoN bot.

Table 8.12 summarizes the new results. While the damage done to the

λ	**Damage Done**	**Creep Kills**		**Win Percent**			**Match Length**
			Overall	*Tower*	*Creeps*	*Hero*	
0.9	71%	-84.1%	43.7%	18.0%	0%	25.7%	221.9s
$0.9^{(c)}$	71%	24%	47.4%	38.6%	0.5%	8.3%	31.2s

Table 8.12: Performance of the the modified agent against the in-game AI, where $^{(c)}$ starts without pretrained weights. Analysis of 1000 games for each case.

opponent drops slightly in both cases, the win percent reaches a competitive level. Furthermore, we can identify two diverse strategies.

Consider the test of agent (b). Here, the strategy of our agent is to start out aggressive, such that the HoN bot would enter its retreat-behavior and flee, after which our agent uses this opportunity to push the creep wave into the enemy tower. Furthermore, while the opponent likes to use its mana on a healing ability, our agent prefers to reduce the HoN bot's damage output. This again leads to our agent being able to push the creep wave faster than the opponent. Correspondingly, this causes our agent to significantly out-damage as well as out-farm the HoN bot. Furthermore, this strategy can be observed by the fact that the agent wins the majority of its matches by destroying the enemy tower.

The agent (a), works a bit different. It still doesn't engage in actively killing creeps. It mostly just harasses the opponent, and falls back when it gets too close to the enemy tower. Thus, the HoN bot is lured out into the middle of the map, at which point our agent can again begin to harass the opponent. Of course, the HoN bot does its fair share of harassment as well, and doesn't hide under its tower all the time. However, our agent manages to be ellusive and drag the match out, until finally one of the two is slain. One can observe this by looking at the difference in average match lengths. Before our "expert intervention", the matches lasted merely three quarters of a second. Now, our bot manages to survive for over three minutes on average.

In the next section we conclude this Chapter by reviewing the agent that was created.

8.7 Discussion

As we have seen our agent isn't almighty. However, we have shown that, given enough training and a sufficient sample space of states, it most certainly learns. Furthermore, our agent manages to adopt entirely different strategies when playing against the official bot in our final test. If given enough time, our agent may very well learn enough such that it finds a strategy to dominate the opponent completely.

There are many options as to how to expand upon our solution. For one there is supervised learning that trains the agent using expert knowledge or player data, for another one could experiment with intermediate rewards. As we have seen during the initial part of our last benchmark test, learning agents seem to have problems adopting when they are starved of positive reinforcement signals. Thus, supplying each action with an immediate reward may very well lead to faster and better learning. However, experimenting with reward functions falls outside of the scope of this thesis.

In addition, one should never forget the advantages of learning agents; they go beyond scripted or hard coded behavior, which is typically quite predictable and thus, easily exploitable. Learning agents may (or may not) require large amounts of data in order to be able to adopt "good" behaviors, but they are also timeless. This is a very important factor for AI in MOBA, as their environment is continuously updated and changed.

Another reason why our agent model may prove to be significant, is that not all MOBA games offer bot code, or sometimes only for a subset of heroes. HoN, for example, offers bots for only about 1/3 of their total number of heroes.

Finally, we should address the design of our agent. Currently it is specifically designed to play against one hero (and furthermore, the same hero). However, this doesn't mean the agent cannot be generalized to other heroes or be modeled into a version that functions independently from the enemy hero. However, this will require a lot of data processing, which goes beyond the means for this thesis, and furthermore, will most likely require to have access to sets of expert plays.

9 Conclusion

In this thesis reinforcement learning was combined with neural networks in an attempt to develop a learning agent that plays the popular MOBA game, Heroes of Newerth, with little to no prior game-knowledge. As good as no research has been done in this direction, as video games in general prove to be too computationally expensive to be attractive for most AI studies. Thus, most commercial video games suffer from unsatisfying computer agents.

We looked at the concepts of reinforcement learning, and ultimately, the TD(λ) algorithm, which became a popular temporal difference learning method after achieving great success and world-wide recognition in Tesauro (1995)'s TD-Gammon agent. Furthermore, we discussed the use of neural networks for value function approximation.

The agent was modeled to use a neural network to develop a function that could be used to evaluate the state of a match. Using reinforcement learning, the agent taught itself through self-play the basic concepts of the game, as well as its win conditions. In extended experiments our agent was evaluated for its learning potential as well as tested against already existing AI. The results showed that, with a few minor modifications, the agent could adopt diverse game-play strategies and compete at the AI's level.

Finally, the underlying, secondary objective of this thesis was to engage future researchers to follow the same path, and seek to find efficient methods to model AI for complex video game environments.

10 Future Work

In the final section of our experiments, we discussed the relevance of our agent and the possible way in which to improve it. Currently, our agent is not very general, thus developing an agent that encompasses a broader view of potential opponents would be one of the first steps to make in the future. Additionally, as this thesis only allowed for a shallow experimentation with various neural network features, future work in this direction would certainly be interesting.

We also have to mention supervised learning. Recent successes such as Silver et al. (2016) have proven that reinforcement learning and self-play can be combined with supervised learning from human expert games to achieve new levels of greatness. Another variant of this would be to let our agent play against real human players, if given the possibility. It would most certainly clarify as to how far learning agents can be applied to MOBA games.

Another possibility is to move away from traditional feed-forward multi-layer networks, and look towards recurrent or convolutional ones. Especially recurrent neural networks offer an allure for the MOBA game challenge, as they allow the incorporation of an internal memory over a sequence of inputs.

Last but not least, the agent could be made more complex by picking its next action by not only considering all of its own moves but also those of its opponents.

References

Thor Bagge and Kent Grigo. Getting to know the captain's mistress with reinforcement learning. 2016.

Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford university press, 1995.

Kur Hornik, Maxwell Stinchcombe, and Halber White. Multilayer feedforward neural networks are universal approximators. *Neural Networks*, 2: 359–366, 1989.

Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4: 237–285, 1996.

Ben Kröse and Patrick van der Smagt. *An Introduction to Neural Networks*. University of Amsterdam, 5th edition, 1993.

Erik Kvanli and Eirik M Hammerstad. A coalition based agent design for heroes of newerth. 2014.

Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1:541–551, 1989.

Yann LeCun, Yoshua Bengio, and Hinton Geoffrey. Deep learning. *Nature*, 521:436–444, 2015.

Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. 2013.

Stuart J. Russel and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3rd edition, 2010.

Waren S. Sarle. Neural networks and statistical models. 1994.

Warren S Sarle et al. Neural network faq. *Periodic posting to the Usenet newsgroup comp. ai. neural-nets*, 1997.

Wolfram Schultz. Behavioral dopamine signals. *Trends in neurosciences*, 30.5:203–210, 2007.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 2nd edition, 2005.

Gerald Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

A Gaming Terminology

Term	Meaning
Ability Power	Increases magic damage dealt.
AD	Attack-Damage; *see Attack Damage*.
ADC	Attack-Damage Carry; The role in a game that plays in the bottom lane of the game.
AP	Ability Power; *see Ability Power*.
Armor	Obtained by items or skills. Reduces physical damage taken.
Assassin	A type of champion, that bases its skills on burst damage
Assist	Gained when a champion is part of the kill of an enemy champion, but doesn't strike the killing blow.
Attack Damage	Increases physical damage dealt.
Base	The base of operation for each team.
Bot	*see Bottom*
Bottom	The lane at the bottom side of the map.
Bruiser	A type of champion, that bases its skills on physical, melee damage.
Buff	
Casttime	The time it takes to cast a spell.
CC	Crowd Control; such as stun, fear, etc.
Champion	Various characters a player can take.
Channel	Channeling skills require the hero to remain stationary for a given amount of time.
Creeps	Allied, enemy or neutral units, that can be killed for gold and experience.
CS	Creep Score
Debuff	
Dmg	Damage
Deny	Killing an allied unit to deny gold to the opponents.
Experience	Points the hero obtains to level up.
Fighter	*See Bruiser*
Fountain	The structure in each team's base that provides absolute safety for that specific team. It regenerates HP and MP, as well as allows the hero to buy items.
FPS	First Person Shooter

Term	Meaning
Gear	Items the champion can buy from the shop that will improve its damage and survivability.
Gold	Resource used to buy items.
Health	A champion has a certain number of health points. When these reach zero, the champion dies.
Hero	*See Champion*
HP	Hit Points; *see Health*
HoN	Heroes of Newerth
Interrupt	A skill's cast or channeling time can be interrupted, preventing the skill from being executed.
Jungle	All areas outside of the lanes.
Jungler	A type of champion that doesn't play in a lane, but instead kills jungle creeps and moves all around the map to assist its teammates.
K/D/A	Kill/Death/Assist ratio
Kiting	Using slowing abilities or movement enhancements to
Lane	The three open roads in the game that lead along the map from one side to the other: top lane, mid lane and bottom lane.
Last Hitting	Obtaining the killing blow and thereby the gold of an enemy unit.
LoL	League of Legends
Mage	A type of champion, that generally uses ranged magic damage skills
Magic Resist	Obtained by items or skills. Reduces magic damage taken.
Mana	A resource consumed by heroes to use their skills. When this resource turns zero, skills can no longer be cast. Mana generally generates slowly over a period of time.
Marksman	A type of champion that generally works together with a support champion. It normally does ranged attack damage.
Melee	Close-range attacks
Mid	The lane at the middle of the map.
Mid Laner	The champion that plays most of the game in the middle lane.
MMO	Multi-Massive Online
MOBA	Multiplayer Online Battle Arena
Mobs	*See Creeps*
MP	Mana Points; *see Mana*

Term	Meaning
MR	Magic Resist; *see Magic Resist*
Nexus	The ultimate structure in each team's base. It must be destroyed to win the game.
NPC	Non-Player Character
Pushing	A behavior a player can adopt. Killing the enemy creeps fast will result in the allied enemy creeps to push forward faster.
RPG	Role-Play Game
RTS	Real-Time Strategy
Silence	Interrupts skill casts or prevents them from happening.
Stats	Heroes have different stats, such as their HP, MP, AP, AD armor, MR, etc.
Stun	Prohibits the hero from moving or casting.
Support	(1) The champion that generally plays alongside the ADC in the bot lane. (2) A support generally has supportive abilities that can help team mates rather than fight opponents.
Tank	A type of champion that gathers a lot of HP to defend its team mate, rather than deal damage to enemies.
Top	The lane at the top of the map.
Top Laner	The champion that plays most of the game in the top lane.
Tower/Turret	Structures each team has that need to be taken down in order to advance forward and kill the nexus.
Trading	Fighting with an enemy champion generally results in each champion trading in some HP to deal dmg.
XP	Experience Points; *see Experience*