Reinforcement Learning for Heroes of Newerth

Verstärkendes Lernen in Heroes of Newerth
Bachelor-Thesis von Julius Willich genannt von Pöllnitz
Januar 2015

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Reinforcement Learning for Heroes of Newerth
Verstärkendes Lernen in Heroes of Newerth

Vorgelegte Bachelor-Thesis von Julius Willich genannt von Pöllnitz

1. Gutachten: Prof. Dr. Jan Peters
2. Gutachten: Prof. Gerhard Neumann

Tag der Einreichung:

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 14. Januar 2015

(Julius Willich genannt von Pöllnitz)

Abstract

Recently computer games became more and more complicated and complex, leaving plenty of room for humans to come up with tactics and strategies. Artificial intelligence (AI) approaches for computer controlled opponents, however, are lagging behind. Most artificial intelligence opponents in current games are rule-based and therefore follow the same pattern repeatedly. The AI agent is not able to adapt to the current game situation or learn from the player. Higher difficulty levels in the game are usually achieved by cheating (e.g., cheaper units), instead of smarter behavior. This often leads to the player feeling treated unfairly. A solution to this problem could be the use of reinforcement learning. Having a game AI that learns from its flaws instead of using cheats to overpower the player would result in a more appealing game experience for the player.

In this thesis, fitted Q-Iteration (FQI) with extra trees (ExT) will be applied to learn Heroes of Newerth (HoN) to see if it is a viable alternative for current game AI. We will show that the algorithms chosen are robust towards irrelevant features, but also point out weaknesses in their performance. It will also show that robustness can be a curse when it comes to optimizing parameters, since changes often do not have a significant impact in the performance. For example, most of the settings for the algorithm resulted in similar results concerning the agent's performance. Another weakness of ExT are the random cuts which struggle with multiple features conveying similar information. For example, adding multiple features that only change when the hero gains a level leads to chaotic results in the performance. We will see that a naive approach in configuring the algorithm is only sufficient to achieve a small degree of self improvement. Better performance might be achieved by evaluating the relevance of features and filtering them accordingly. Improving the representation for actions would be another way to improve overall performance.

Contents

Figures and Tables

List of Figures

List of Tables

Abbreviations, Symbols and Operators

Acronyms

$Q(s,a)$ Q-function

$R(s)$ reward function

$V(s)$ value function

$\pi(s)$ policy function

π policy

r reward

v value

AI artificial intelligence

ExT extra trees

FPS first person shooter

FQI fitted Q-Iteration

HoN Heroes of Newerth

HP hit points

LSPI least-squares policy iteration

MDP Markov decision process

MOBA multiplayer online battle arena

MP mana points

RL reinforcement learning

RTS real time strategy

XP experience points

Glossary

armor

Gained by leveling up or buying equipment. Reduces the incoming physical damage.

creep

Computer controlled unit that is weaker than hero units. Spawns every 30 seconds in groups which move on a set path over the lane into the enemy base, attacking all enemy units or structures on their way.

deny

Killing an allied unit. Attacking allied units (or structures) is only possible when they are already low on health points. Killing them results in the enemy only gaining half of the experience points (XP) and no gold.

experience points

Experience points are gained by killing enemy creeps and heroes. When the hero earned enough experience points it gains a level, which improves its stats.

fountain

A special structure in every team's base that regenerates HP and MP of allied units inside. It also targets enemy units with one of the strongest attack in the game, if they enter. It is also the place where heroes respawn.

gold

Resource used to buy items at the shops.

hero

A special unit type, controlled by either a player or a sophisticated AI. It has special abilities and can gain gold to purchase items and XP to level up.

hit points

When a units health points are reduced to zero, the unit is killed. Damage inflicted on the unit is first reduced by armor or magic armor and then deducted from the current HP value.

last hit

Reducing a unit to zero (or less) health points thereby killing it and gaining gold and experience points.

magic armor

Gained by leveling up or buying equipment. Reduces the incoming magic damage.

mana points

Mana Points are used to activate a heroes abilities.

Markov chain

A chain of states with stochastic transitions, independent from their past states.

Markov decision process

Extension to Markov chain including a decision which influences the transition.

policy

The decision policy the AI agent uses.

policy function

The policy function returns the optimal action given the current state.

Q-function

The Q-function returns a reward given a state-action combination.

reward

The reward describes a complex, multi feature state as a single value, which represents the desirability of that state.

reward function

The reward function returns a reward value given a state.

stats

Heroes have different so called stats which define their health points (HP), armor, magic armor. They can be increased by gaining a level or buying items.

value

The value of a state describes its desirability with respect to it being the basis of future decisions.

value function

The value function assigns a value to a given state.

1 Introduction

Designing challenging and fair computer-controlled opponents is a big problem in modern computer games. The current standard for enemies is rule-based behavior, which does not allow to adapting to the player's behavior. Such approaches usually lead to strategies on the human players' side which focus on exploiting weaknesses in the current game artificial intelligence (AI) agents' rule set. Stronger opponents are usually emulated by cheats such as cheaper units or faster production rates. Both problems, predictability and unfair opponents, could be solved by using reinforcement learning (RL) algorithms for game AI agents. In this thesis, the suitability of fitted Q-Iteration (FQI) in combination with extra trees (ExT) will be analyzed. FQI is a modified version of Q-Iteration, which is a RL algorithm that employs a reward function. The use of a reward function allows for easy modification of the AI agent's goal and its behavior, without modifying the code itself. Such approach would allow for easy adaption of the AI agent. ExT is a modified version of the standard regression tree algorithm and is the function approximator used with FQI. The advantages of ExT compared to the standard regression are the runtime optimization achieved by random cuts and the resistance towards noise and over fitting, achieved through the use of multiple trees in a forest.

1.1 Motivation

Many modern computer games rely on artificial intelligence (AI) for multiple different purposes. One of which is to provide enemies with controllable difficulty for players in single player games. Another is compensating for missing players in games that rely on a set team size.
The problem with those AI-controlled enemies is that they usually are realized as simple, deterministic scripts, and they are not able to learn from their mistakes. Such AI controlled opponents are easily exploited once the player finds a weakness.
Learning approaches would result in AI agents that fix those weaknesses. Furthermore, the computer could start exploiting errors made by the player, thereby teaching the player how to improve their performance in the game. The problem is that self-learning AI in general and for computer games in particular still is an open research topic.
For computer scientists, games provide complex testing environments for a variety of reinforcement learning (RL) algorithms. A commonly used game genre for RL is real time strategy (RTS). Those games rely on strategy and tactics to win, rather than reflexes or accuracy in controlling units in the game. Since short reaction time and high accuracy are easily realized with computers, those games present a testing environment which does not favor the computer too much. Typical goals for RTS games are collecting resources, scouting for enemies and spending the resources on buildings or units. The necessity for scouting arises from the game mechanic called fog of war. It hides unit movement in parts of the game map that are currently not visible to the player's units, thus providing a level of uncertainty to the RL problem.
Most approaches to learning RTS games rely on the programmer's domain knowledge to identify important features or generate complex actions to speed up the learning process. Forming sub-goals or even whole strategies for the algorithms to use is another way of improving the algorithm's performance.

Figure 1.1: The "fog of war" —colorized red— hides information from the player. Units and the health points of buildings are hidden. A possible position of another unit is marked with a blue circle. The positions of buildings and the general terrain are not hidden since they are not subject to change.

Multi-layer approaches are a different commonly used method to increase an algorithm's performance. Such approaches split the game into multiple abstraction levels, like micro-managing the movement of units or ensuring a constant flow of resources. Usually one level of abstraction is learned, such as when to spend resources and when to gather them, while

the rest is provided, such as complex actions usable by the agent. Such actions, gathering resources for example, would not need to be learned by the agent, it would only have to learn when best to spend resources.
Finding an approach that is not influenced by irrelevant features and which works with little domain knowledge concerning the goals of the game, would allow for easy integration into different computer games. Ideally we want to use the same approach on many different games, using only minor adjustments like a new reward function.
In this thesis, extra trees (ExT) will be used as function approximator, because they have proven to be effective concerning computation time and resistant toward irrelevant input features. They work well with noisy data and a high amount of features, plus they can be tuned to achieve the needed accuracy. Fitted Q-Iteration (FQI) is a modified version of Q-Iteration allowing for arbitrary function approximators to be used. It only needs a simple reward function to work and therefore should be easy to use with and adjust to a new game with little domain knowledge.

1.2 Related Work

Games that use machine learning approaches are used can be divided into three main categories.

- The action genre, which includes first person shooter (FPS) where the computer has an advantage due to its precision and reaction time [13][11][12].
- The turn-based strategy genre, which includes board games, where humans have the advantage of generalization and their intuition, and computers have sufficient time to compute the next step using sophisticated algorithms [16][5][14][1].
- And the real time strategy (RTS) genre, a mixture, where strategy and precision in controlling units can improve the player's or the computer's performance [4][3][6][15][7].

The focus for FPS games is on navigation, staying alive, and attacking the enemy while the focus in RTS games is on maintaining an efficient economy and expanding control over the game map. Heroes of Newerth (HoN) is placed somewhere in between RTS and action games. The player focuses on a single hero unit but still needs to maintain a small economy. He needs to increase the team's knowledge of the map and form strategies to attack targets with the whole or part of the team. The approach in this thesis breaks the game down into multiple sub-tasks as seen in the paper by McPartland and Gallagher [12] and the thesis by Dimitriadis [7]. The idea behind this approach is the learn every task individually and subsequently learn how to best combine to learned tasks. This thesis however will only cover one of those sub tasks.

1.2.1 Reinforcement learning for computer games

Let us take a look at the basics of reinforcement learning (RL) for computer games. The principle behind the RL used in this thesis are Markov decision processes (MDPs), which are based on Markov chains. Markov chains are sequences of discrete states with stochastic transition functions where the next state only depends on the current state and not on the history of the sequence. This is known as the markov property and allows for the states to be treated independently. MDPs add a reward and the choice of an action to Markov chains allowing to choose an action based on the reward r they earn. The basic idea behind RL is to find the optimal control policy π to maximize the discounted sum of rewards r over the potential infinite horizon, given by $\sum_{t=1}^{\infty} \gamma^t R(s_t, a_t, s_{t+1})$. The variable γ is called the discount factor and regulates how much an instantaneous reward is preferred over a future reward and is usually close to 1. The policy governs which action a to take based on the information about the current state s. The desirability of a state is defined by its reward r which is defined by its reward function $R(s)$. It maps a complex state with multiple features to a single reward value. The reward function $R(s)$ itself can be adjusted to achieve the desired behavior. It is usually represented as a sum of different features, each of which is weighted by their desirability. The artificial intelligence (AI) agent's goal is to maximize the reward obtained over a given period of time. Another important function is the so called value function $V(s)$ given by

$$V(s_t) = r_t + \gamma \int_{s_{t+1}} P(s_{t+1}|s_t, a) V(s_{t+1}) ds_{t+1}, \tag{1.1}$$

where $P(s_{t+1}|s_t, a)$ is the probability of transitioning to state s_{t+1} when in state s_t and taking the optimal action a. Intuitively, the value function is a reward function with added planning. The value v of a state is governed by its reward plus the discounted reward that can be expected in the future. This means that a state with a small reward can still have a high value because it is the base for further lucrative actions. One way to store the policy and value function is the tabular form. The policy could then be easily realized by choosing the best action according to this table. This of course

poses a problem for high dimensional state and action spaces.
Building a complete value function table requires a lot of samples from the game or a sufficient model. The problem in computer games is that the state and action spaces are high dimensional, and each dimension has a high range of values. Possible features for describing a state include health points (HP), easily range from zero to multiple hundred per entity and entity position, which can range from zero to multiple thousand distance units per axis. Another typical mechanism in computer games that impedes the computation of a complete value function is fog of war, which hides information about the current state from the agent. Also many computer games incorporate random factors for different aspects of the game which add noise to the gathered data. A standard way to deal with those problems is function approximation which is used in many RL approaches such as Q-Learning, SARSA, and LSPI.
Function approximation is the process of finding a function which best describes the data points from a given observed data. The solution can often be represented as a linear combination of weighted features. In that case, the weights $\boldsymbol{\omega}$ can be learned and the right features have to be chosen for the feature vector $\boldsymbol{f}$. Finding the best features for the feature vector $\boldsymbol{f}$ is usually not an easy task, thus approaches that work without it are easier to apply to new problems. The update function for those weights can be represented as

$$\boldsymbol{\omega}_{t+1} = \boldsymbol{\omega}_t + \alpha(r + \gamma(Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\boldsymbol{f})), \tag{1.2}$$

where a_{t+1} and a_t are the best actions given the state s_{t+1} and s_t, respectively. The learning rate α governs how fast old knowledge is overridden with new information.
Q-Learning is an off-policy algorithm introduced in "Reinforcement Learning: An Introduction" by Sutton and Barto [17]. It aims to improve the so called Q-Function Q(s,a), which is similar to the value function. It is the expected summed future reward when starting in state s and taking action a and can be described by the equation

$$Q(s_t, a) = r_t + \gamma \int_{s_{t+1}} P(s_{t+1}|s_t, a) Q(s_{t+1}, a_{t+1}) ds_{t+1}, \tag{1.3}$$

where a_{t+1} describes the action taken in state s_{t+1} under the current policy. Intuitively, the difference to the value function is that it not only maps a state to a value but a state-action pair. Off-policy algorithms can learn an estimation and a behavior policy. They can update the estimation with theoretical actions which means that they can learn behavior never experienced during exploration.
Q-Learning also is a batch mode algorithm, which means that the updating phase is separated from the information gathering and evaluation phase. It also separates the computationally intensive task of updating the value function from the time-critical task of decision making during the game. During updating, all data gathered up to this point is used, even the data from past iterations. Updating Q(s,a) is achieved by the following equation

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right], \tag{1.4}$$

where α is called the learning rate and governs how quickly old information is overridden by new experience.
SARSA is very similar to Q-Learning, but it is an on-policy algorithm. As an on-policy algorithm it is limited to real experience when it comes to improving its policy. This leads to better results but needs more computational power during the game. It is also introduced in "Reinforcement Learning: An Introduction" by Sutton and Barto [17].
Another difference to Q-Learning is that SARSA needs quintuples as input, in the form of $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$ —which also led to its name— while Q-Learning works on quadruples in the form of (s_t, a_t, r_t, s_{t+1}). This eliminates the need to chose an action to take a_{t+1} since it is already provided. The resulting update function,

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma\, Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right], \tag{1.5}$$

is similar to the Q-Learning update function.
Least-squares policy iteration (LSPI) is a function approximation approach introduced in "Least-Squares Policy Iteration" by Lagoudikas and Perr [10] is, like Q-Learning, an off-policy, batch-mode algorithm. It works without a learning rate α which is not the case for Q-Learning and SARSA. LSPI also does not compute the value function directly, instead it computes the optimal weights $\boldsymbol{\omega}$ for a number of basis functions Φ. A basis function $\boldsymbol{\phi}$ can either be a feature or an equation containing multiple features of a given state. The Q, or value function is then given by:

$$Q(a, s) = \boldsymbol{\phi}(s, a)^T \boldsymbol{\omega}. \tag{1.6}$$

The weights $\boldsymbol{\omega}$ are obtained by solving the $(k \times k)$, system given by

$$\mathbf{A}\boldsymbol{\omega} = \boldsymbol{b}, \tag{1.7}$$

with k being the number of basis functions. $\mathbf{A}$ and $\boldsymbol{b}$ can by learned from samples by

$$\mathbf{A} = \frac{1}{N}\sum_{t=1}^{N}\Big[\boldsymbol{\phi}(s_t,a_t)\big(\boldsymbol{\phi}(s_t,a_t)-\gamma\boldsymbol{\phi}(s_{t+1},\pi(s_{t+1}))\big)^T\Big] \tag{1.8}$$

and

$$\boldsymbol{b} = \frac{1}{N}\sum_{t=1}^{N}[\boldsymbol{\phi}(s_t,a_t)r_t]. \tag{1.9}$$

The policy in all cases is given by $\pi(s) = \max_{a_{t+1}} Q(s_{t+1},a_{t+1})$.

1.2.2 Glest

Glest[1] is a typical real time strategy (RTS) game with different factions and collectible resources. In addition, it is open source software and therefore comes with many useful community-made tools and access to game data. Therefore, it is an ideal platform for benchmarking RL games. It features two different factions with different advantages and disadvantages. There is the tech faction, which is strong in melee combat and easy to learn; and the magic faction, which is a bit more complex and weak in melee combat, but strong in ranged combat. For this game the SARSA and the least-squares policy iteration (LSPI) algorithm are used. The states of the game have been abstracted for this learning scenario since maintaining all information would have been too costly. The actions have been divided into three groups:

1. Low-level actions such as checking if a worker has a task and, if not, giving him one
2. Mid-level actions which consist of grouped low-level actions such as building units or structures
3. High-level strategy and tactics made up from groups of mid-level actions

To create initial examples for the algorithms to learn from, the mid-level actions have been executed depending on the game timer. Normal, early, and very early timing intervals (compared to the standard game artificial intelligence (AI) agent) were used. The reward was calculated by the change in kills(x100), units(x50), and resources(x0.1) which are the same weights the game uses to calculate the final score. The SARSA algorithm learned the timing on the mid-level actions. The algorithm changed the timing over the course of a game. The least-squares policy iteration algorithm, being an offline learning algorithm, could not learn during a match. The data for the offline learning was taken from random games and from games played by the SARSA algorithm. Two game modes were used to evaluate the performance and to improve the policies.

1. Duel mode, a game against the standard artificial intelligence with a medium difficulty level
2. Tough duel mode, a game against an artificial intelligence on hard difficulty which had cheaper, better units, gathered resources faster; etc.

Ten games of learning were followed by one game to evaluate the policy.
In duel mode, all algorithms succeeded in beating the standard game artificial intelligence agent. In tough duel mode they did not, which is not surprising. Concerning the rate of policy improvement, SARSA did best in the duel mode with a peak of 129% improvement after 20 test games. The least-squares policy iteration algorithm with random samples also peaked after 20 games with an improvement of 123%. The least-squares policy iteration algorithm using SARSA game samples peaked after 150 test games with an improvement rate of only 86%. In tough duel mode SARSA reached its peak improvement after 50 games, with an improvement of 22%. Least-squares policy iteration with random samples only achieved an improvement of 2%. This improvement was reached after 40 games. The best performance was achieved by least-squares policy iteration with SARSA samples after 160 games with an improvement of 36%. The policies did not converge towards a good policy, however. This could be due to the fact that not all information available was processed by the algorithms, e.g., positioning. As mentioned before, maintaining all state information would have been too costly.

[1] www.glest.org

1.2.3 FPS

For reinforcement learning (RL) in first person shooter games, we take a look at [12]. For their experiments, they created their own first person shooter (FPS) game so they were able to control the testing environment. They used the SARSA reinforcement learning (RL) algorithm, a modified Q-Learning algorithm. SARSA allows current, successful actions to improve the reward of past actions. Therefore it is possible to learn action sequences and it also speeds up the learning process.
They used the tabular version instead of function approximators. Since the state space was too complex, sensor input abstraction was applied. Visual inputs were abstracted to either near or far; enemy or item; and left, middle, or right. Actions the agent could take were: strafe left/right, turn left/right (5°), move forward, move backward, and shoot. The learning was divided into two stages. The first stage was learning to navigate in the given maps, collecting power ups. In the second stage, the agent learned how to attack and kill its opponents.

The learning goals for the movement stage were:

1. minimize collision
2. maximize distance traveled
3. collect power ups

To achieve this, a high reward was given when the bot collected an item. Small rewards and penalties were given for moving and colliding with walls, respectively. Multiple agents with different values for λ and γ were tested. With λ representing the exploration rate, which governs how often random actions are taken over the best action according to the policy. The best results were achieved by bots with either a high (0.8) or very low (0) value for λ, meaning the bot either "planned" a lot or not at all. Overall the reinforcement learning approaches were better then total random actions, but not as good as A* with item finding rules.
During the combat stage, a high reward was given for hitting and for killing an enemy. A low penalty was given for shooting and missing an enemy as well as for getting hit, and a big penalty was given for dying. As enemies for the agents, two passive bots that only navigated the map and a state machine bot that also attacked the agents were added. Again, either very low or high values for λ were the most successful. Furthermore, they added a bot that got high reward for collecting items, hitting an enemy, and killing an enemy and a low penalty for colliding with the map geometry.
For the final agents, the best result from the navigation and combat stage were fused together. This happened by either Hierarchical reinforcement learning, that learned when best to use the navigation controller and when to use the combat controller; or rule-based reinforcement learning, that used the combat controller when an enemy was in sight and the navigation controller when not. The Hierarchical and the rule-based reinforcement learning both performed better than the agent that did not learn navigation and combat separately. The Hierarchical agent was the best considering accuracy, the rule-based reinforcement learning agent performed better in the other objectives. Still, observations showed that the hierarchical reinforcement learning agent had a wider set of behaviors, such as fleeing when confronted with too many enemies. The plain reinforcement learning agent however outperformed the state machine agent in the item collection task. Which is interesting since the state machine was explicitly programmed to collect items in its vicinity. All in all the rule-based reinforcement learning and the hierarchical reinforcement learning approach showed behavior comparable to advanced first person shooter players. The plain reinforcement learning approach showed behavior comparable to a beginner level first person shooter player.

2 Fitted Q-Iteration with Extra trees

The following algorithms were chosen based on the results of the paper by Guerts et al. [9], where extra trees (ExT) have proven to be robust with regard to irrelevant data and work well with fitted Q-Iteration (FQI). FQI, as most reinforcement learning (RL) algorithms, is suitable since once the algorithm is implemented, changing the reward function is not complicated and only needs minor domain knowledge concerning the game and close to no knowledge concerning the algorithm. One reason for this is that FQI does not need a defined feature vector to learn a policy. Defining a suitable feature vector is one of the big problems of RL. This enables game designers to use the algorithm without knowledge of machine learning. They only need to identify features of the game which should be maximized to improve as a player. Most games have resources that are important for the player's advancement and therefore present prime candidates for features. The algorithm's robustness towards unimportant features, which was shown in [9], also reduces and often even eliminates the negative effects unimportant features would have, with some exceptions that were found during the tests for this paper. Another advantage of FQI over standard Q-Learning is its sample efficiency due to the fact that data points are reused in every iteration. Higher sample efficiency leads to faster learning which again allows for faster policy optimization.

2.1 Extra trees (ExT)

Let us quickly recap decision trees as a data structure. In a decision tree, there are three kinds of nodes connected by edges: the root node, the internal nodes, and the leaves. Starting at the root node, a decision tree can be defined recursively. The root node and the internal nodes can have up to two child nodes, whereas leaves have no child nodes. Every node has exactly one parent except for the root node. Every leaf additionally contains a label. A root node or an internal node contains a test, that splits the search space and is used to traverse the tree when searching for a solution. Every test, also called "cut", consists of a cut direction, which determines which features are tested, and a cut value, which provides the value against which the feature is checked. The edges connecting the node to its children correspond to the possible outcomes of the test.
Extra trees (ExT), as presented in the paper "Extremely randomized trees" by Guerts, Ernst and Wehenkel [9] and in the thesis "Feature Extraction for Policy Search" by Amend [2] uses an ensemble of multiple regression trees, called forest. In a forest, the solution of a given decision problem is averaged over all its trees. Since every tree is trained individually, each tree represents a different model of the same problem. By averaging over its random trees the algorithm is robust towards locally inaccurate solutions of a single, overfitted tree. In order to be able to poll a tree for a solution, that tree needs to be built or trained from a given data set. Training a tree, according to the ExT algorithm, is done recursively as follows. First, a number K of cuts is generated randomly and the existing data set S is split into two new sets (S_1, S_2) per cut according to it. Every entry is a state, action, value triple. Now the score of every cut is determined by computing the variance of the values in the two new sets, weighted by their respective size. The cut with the best score s, which is the lowest combined variance as seen in Equation 2.1, is chosen and the corresponding cut is saved in the current node.

$$s = -\frac{|S1|}{|S1|+|S2|}var(S1) - \frac{|S2|}{|S1|+|S2|}var(S2) \tag{2.1}$$

The associated subsets S_1 and S_2 are used to create two new child nodes. For both nodes the process is repeated until the size of the data set in a node deceeds a set value n_{min}. In this case, the average states' value over the remaining entries is set as label for the leaf.
We now will illustrate the algorithm with an example. The number of random cuts K is set to two. The value n_{min}, the minimal number of data points required in a node to perform another split, is set to three in this example. The pseudocode for the generation of ExT is shown in Algorithm 1. For this example, let us assume a one dimensional environment, where the rewards for starting in a space are, from left to right, 3,3,2,2,1,1 as seen in Table 2.1. The algorithm works with two inputs and one output, the inputs being state and action and the output being the reward. The possible actions are moving left or right, represented by the numbers one and two respectively. As data we have gathered every possible state and action pair once, resulting in 12 data entries since moving out of the world results in standing still.
This example follows only the left hand side of the nodes to the leaf. The first two random cuts to be evaluated are (1,4) and (2,1), which translates to every entry with the current position being lower than 4 and every entry with the movement order being "left". The set will be split as follows and sorted according to the cut dimension chosen. The first cut (1,4) results in two new sets with equal size and the second cut (2,1) results in a new set containing all the data and

position	1	2	3	4	5	6
reward	3	3	2	2	1	1

Table 2.1: The world in which the ExT example is computed.

one empty set as seen in table 2.2.

(1,4)

position	movement	reward
1	1	3
1	2	3
2	1	3
2	2	2
3	1	3
3	2	2
4	1	2
4	2	1
5	1	2
5	2	1
6	1	1
6	2	1

(2,1)

position	movement	reward
empty set		
1	1	3
2	1	3
3	1	3
4	1	2
5	1	2
6	1	1
1	2	3
2	2	2
3	2	2
4	2	1
5	2	1
6	2	1

Table 2.2: The horizontal line in each table represents the two subsets of the data according to the cut noted above the table. With the first number representing the cut dimension and the second value representing the cut value.

Now we need the score of both cuts, which is defined by the combined variance of both new sets, weighted by their respective size

$$-\text{abs}\left(\text{var}\left(3,3,3,2,3,2\right)\frac{1}{2}+\text{var}\left(2,1,2,1,1,1\right)\frac{1}{2}\right)=-0.2667 \quad \text{and} \quad -\text{abs}\left(0+\text{var}\left(3,3,3,2,2,1,3,2,2,1,1,1\right)1\right)=-0.72\,. \tag{2.2}$$

The next cuts will be (1,2) and (2,2) resulting in the splits as seen in table 2.3.

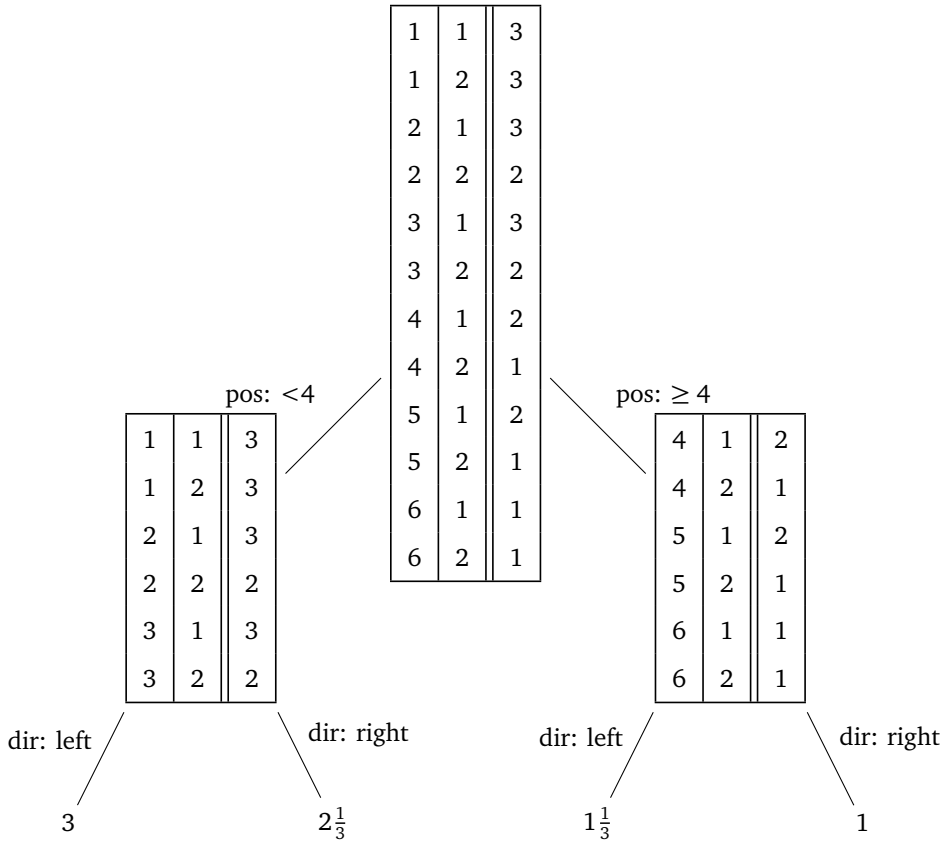

Figure 2.1: The resulting ExT from the example

(1,2)

position	movement	reward
1	1	3
1	2	3
2	1	3
2	2	2
3	1	3
3	2	2

(2,2)

position	movement	reward
1	1	3
2	1	3
3	1	3
1	2	3
2	2	2
3	2	2

Table 2.3: The horizontal line in each table represents the two subsets of the data according to the cut noted above the table. With the first number representing the cut dimension and the second value representing the cut value.

The new sets now have less than four elements and will be converted into leaves. The reward in the leaves will be $\frac{3+3+3}{3} = 3$ and $\frac{3+2+2}{3} = 2\frac{1}{3}$. The cut scores are

$$-\text{abs}\left(\text{var}(3,3)\,\frac{1}{3} + \text{var}(3,2,3,2)\,\frac{2}{3}\right) = -\frac{2}{9} \quad \text{and} \quad -\text{abs}\left(\text{var}(3,3,3)\,\frac{1}{2} + \text{var}(3,2,2)\,\frac{1}{2}\right) = -\frac{1}{6}\,. \tag{2.3}$$

Applying the same cut the rest of the tree yields (2,2) as the better cut. The resulting sets are so small, that they will be converted into leaves and the algorithm terminates since there is not data left to process. The resulting tree will look like Figure 2.1.

Input: data,M,K,n_{min}
Result: forest
for *1:M* **do**
  create a node containing data;
  for *every node* **do**
    if *entries in node* $> n_{min}$ **then**
      choose K random features from data;
      for *every feature choose a random cut value* **do**
        divide data into two sets (S1,S2) according to the cut;
        set the score of the cut to $\left(-\frac{|S1|}{|S1|+|S2|}var(S1) - \frac{|S2|}{|S1|+|S2|}var(S2)\right)$;
      end
      choose the cut with the highest score;
      create 2 new nodes from the 2 sets;
    else
      set leaf value to average reward of the remaining entries;
    end
  end
end

Algorithm 1: Pseudocode for the creation of extra trees.

Initial parameters for extra trees

In the paper "Extremely randomized trees" Guerts et al. [9] examine the performance of ExT in different regression and classification problems. Especially the effects of the number of trees M, the minimum number of entries for a node n_{min} and the number of cuts K were analyzed. A good value for K was shown to depend on the problem at hand. Regression problems had best results for $K =$ (number of features) and classification problems had best results for $K = \lceil \sqrt{\text{number of features}} \rceil$. Problems with discrete outputs have shown to behave like regression problems concerning the best setting for K. For M, 10 has proven to be a solid initial choice. A good value for n_{min} depends on the noisiness of the data. Good values for noiseless data have proven to be 2 for regression and 5 for classification problems. However bigger values should be chosen if the data is noisy.
In chapter 3 we will see that Heroes of Newerth (HoN) does not use deterministic rewards. This makes 20 a good initial choices for n_{min}. Most of the data taken from HoN and all outputs from the algorithm are discrete, making $K = \lceil \sqrt{\text{number of features}} \rceil$ a good initial setting.

2.2 Fitted Q-Iteration (FQI)

Fitted Q-Iteration (FQI) is a modified version of standard Q-Learning introduced in "Introduction to Reinforcement Learning" by Sutton and Barto [17]. It is superior to most standard reinforcement learning (RL) approaches concerning feature efficiency and does not require a feature vector. Finding a feature vector can pose a big problem, thus eliminating the need to find it makes the algorithm much easier to apply to different problems. This is one reason why it is applied to various RL problems in combination with various tree based function approximators in "Tree-Based Batch Mode Reinforcement Learning" [8]. Q-learning as introduced in chapter 1.2.1, is a simple off-policy algorithm which does not require a model of the system to be learned. It relies on the Markov property to reduce the RL problem to a sequence of regression problems on the whole data available. The target values are given by

$$Q_{t+1}(s_t, a_t) = r_t + \gamma \max_{a+1} Q_t(s_{t+1}, a_{t+1}). \tag{2.4}$$

The resulting algorithm is introduced in "Tree-Based Batch Mode Reinforcement Learning" by Ernst, Guerts and Wehenkel [8] and is called fitted Q-Iteration (FQI). In contrast to standard Q-Learning, FQI is a batch mode algorithm. This means that it uses all data gathered during all past evaluation phases and uses it to compute a new policy. Computing a new policy also uses a different approach than Q-Learning. FQI computes the Q-Function from which the policy is derived recursively during a learning phase. The Q-Function is computed multiple times using the same data, iteratively improving on the most recent Q-Function. The number of repetitions defined by the factor n, governs how much the final policy looks into the future when making decisions. Of course more repetitions take more time to be completed and therefore selecting the correct value n is a trade off between desired planning and computation time. In contrast to standard Q-Iteration, FQI is able to use arbitrary function approximators. In this paper ExT is chosen as function approximator because it has proven to be effective in "Extremely randomized trees" by Guerts, Ernst and Wehenkel [9].

One additional change had to be made for the algorithm to work with the data from Heroes of Newerth (HoN). Since actions are not equally long, rewards given for the actions have to be discounted accordingly. Therefore the length of an action is taken into account by multiplying the expected reward with γ^{Δ_t} where Δ_t is the duration of the action. Algorithm 2 describes the FQI algorithm as pseudo code. D represents the sampled data. The variable n governs how many iterations per data set are performed when learning a new policy π and can be adjusted as needed. A higher number of repetitions results in a bigger look ahead when planning an action.

Input: Q_m, D, γ
Result: Q_{m+1}
Repeat n times
  foreach *6-tuple of* $a_t, s_t, r, s_{t+1}, v, \Delta$ *in* D **do**
    Get action a_t and states s_t, s_{t+1} from D
    Get duration Δ of a_t and reward r of state s_t
    Choose action a_{t+1} at state s_{t+1} according to policy π derived from Q_m
    Get value v of the state and action tuple (s_{t+1}, a_{t+1}) from Q_m
    Build new Q-function Q_{m+1} according to
    $Q_{m+1}(a_t, s_t) = r + \gamma^{\Delta} v$
  end
end

Algorithm 2: Pseudocode describing fitted Q-Iteration (FQI).

3 Experiments

3.1 Heroes of Newerth

Heroes of Newerth[1] is a free-to-play multiplayer online battle arena (MOBA) game by S2 Games[2], a development studio based in Kalamazoo, USA. The game is controlled with keyboard and mouse from a top down view. The most common setup is 5 vs. 5 players on the map shown in 3.1.

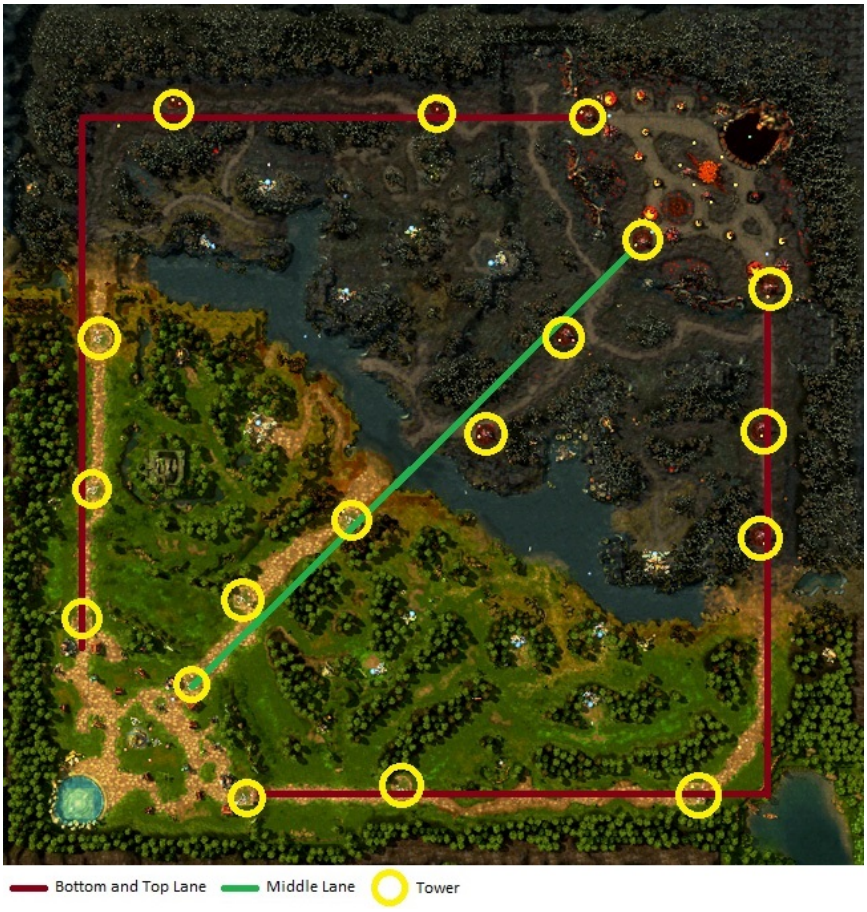

Figure 3.1: Standard 5 vs. 5 players map in HoN

Every player in the game controls only one hero. Heroes are unique units with four unique abilities and there are more than 150 different heroes. Abilities can be active or passive and usually cost a set amount of mana points (MP) to use. Aside from their abilities, heroes can use a standard attack which does not require MP to use. Heroes can gain levels by collecting experience points (XP). When they gain a level, they can improve/learn one of their four abilities. XP are gained by either being present when an enemy unit is killed or by killing them. A unit is killed by reducing their health points (HP) to zero. Players usually divide themselves into three groups per team, two players per team play on top and bottom lanes and one player per team on middle lane. Players are free to roam the whole map, usually with the aim to ambush enemies on a different lane, in order to help allied players there. The ultimate goal is to destroy all towers on one lane in order to destroy the enemy base. A tower is a stationary unit with high attack damage and HP. Towers and lanes are highlighted in Figure 3.1.
Apart from player controlled heroes, every lane is populated by so-called creeps. Creeps spawn every 30 seconds on the three lanes at the two teams' bases. They follow the lane into the enemy base, attacking heroes, enemy creeps and towers they encounter. Killing them, also referred to as "last hitting", is the main method of gaining gold and XP. The amounts gained for both are not deterministic, they have a small range from which the amount is randomly chosen. Gold is also gained at a rate of one gold per 0.875 seconds. Both mechanisms add noise to the gathered data, making it more difficult to apply reinforcement learning (RL) algorithms. Gold can be used at shops to buy items which can improve the hero. Such items can grant the hero more MP, HP, magic armor or armor, and they can increase attack damage or give extra abilities. This makes gold and XP prime candidates to be included in the reward function.
Another game mechanic is denying. A deny is performed by attacking and killing an allied creep or tower. Attacking an allied creep is only possible if it has less than 50% of its maximum HP, attacking an allied tower is possible if it has less than 10% HP. A denied creep only gives half the XP and no gold to the enemy heroes in the vicinity. A denied tower only gives half the gold to the enemy team. Destroying a tower gives a set amount of gold to every player on the team.

[1] http://www.heroesofnewerth.com/
[2] http://www.s2games.com/

Towers are invulnerable as long as there are towers on the same lane in front of them. When all towers on a lane are destroyed, the barracks and the main base become vulnerable. Destroying enemy barracks improves a team's own creeps, destroying the enemy base wins the game.
In the base there is the so-called fountain, which regenerates allied units and easily kills enemy units that get in range. The fountain is also the place where heroes respawn after they were killed.

3.2 Experiments and results

For the experiments the game was reduced to a simpler task, compared to the normal game setup. The settings of the game itself and the configuration of the test system were identical in all tests[3]. The learning and test matches took place on the middle lane only. There were no enemy hero and the artificial intelligence (AI) agent did not use its hero's abilities. The game was terminated when the hero died or after one and a half minutes passed.
The AI agent's actions were limited to moving 100 or 200 units per direction (resulting in a 5x5 movement grid) and attacking an attackable unit. Attackable units are every enemy unit and every allied unit with less than 50% of its maximum HP. Attacking was limited to the allied, and the enemy unit closest to the hero, those with the least HP or a random allied or enemy unit. Special actions were moving to the fountain and return to the battle after being healed. This resulted in 25 movement, six attack and two special actions.
The state space included the hero's two-dimensional position, the position and HP of the allied and enemy creep closest to him and those with the lowest HP, as well as the total number of allied and enemy creeps visible to the hero. The hero's current HP, maximum HP (which indirectly represent its level and attack damage) and the number of towers attacking the hero were also included. This results in a total of 15 features.
The reward function used gold, experience points (XP) and denies. Different weights for gold, experience points (XP) and denies were used in different test setups. After every 25 matches, a new policy was learned, using the data from the two to four most recent iterations, again depending on the test setup. Evaluating the current policy and gathering data for learning a new policy were done simultaneously. Every setup was tested five times. Initial experiments with long maximum time per match showed that high rewards were achieved by simply increasing the survival time, instead of achieving a high reward-to-time ratio.
To encourage more efficient behavior, the maximum duration per match was set to 1.5 minutes. A comparison of both approaches' reward values and average match times can be seen in figure 3.2. On the one hand this forced the AI agent to improve its ability to score and on the other hand it helped in reducing the computational and memory load on the computer by reducing the amount of data gathered.

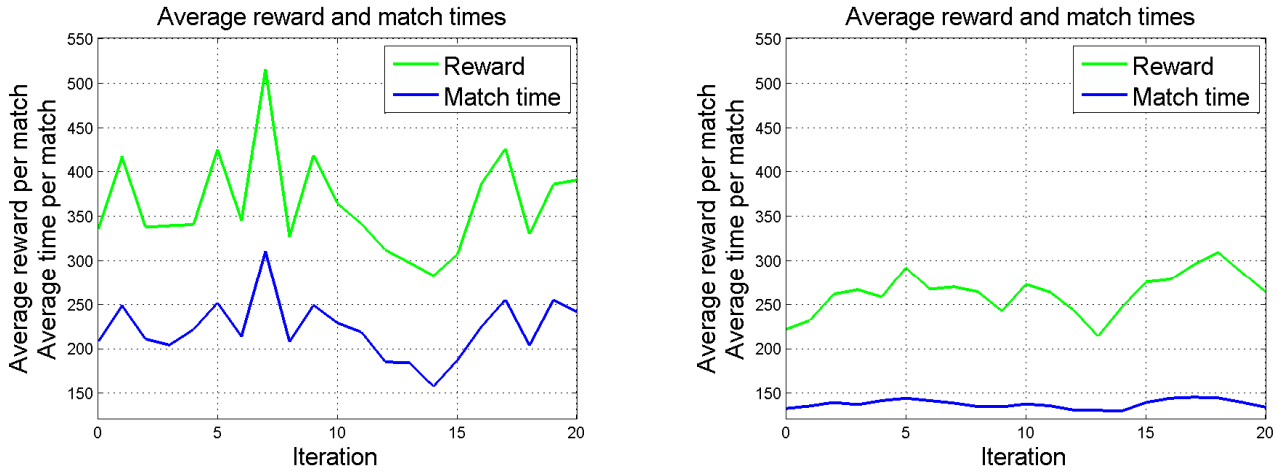

Figure 3.2: Comparison of typical reward and time values of a 5 minutes test run (left) and a 1.5 minutes test run (right).

Initial settings for reward function were XP * 0.4, gold * 1.2, deny * 4, and -0.04 * percent of HP lost. For extra trees (ExT) the initial settings were $K = 16$, $n_{min} = 20$ and $M = 10$, in addition the exploration probability was initialized to 20%. The exploration probability governs how often the AI agent takes random actions instead of using the current policy. By default, only the data from the last 50 games was used to compute new ExT.

[3] Heroes of Newerth (HoN) version: 3.3.0.6, Intel® Core™ i5-4670 CPU 3,4 GHz, 16GB RAM and AMD Radeon™ HD 7900 with 4GB RAM

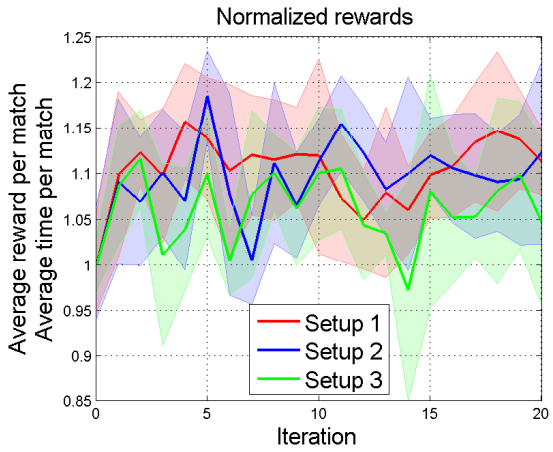

Figure 3.3: Averages of the relative performances of the first three setups.

Figure 3.3 shows the first three setups, which produced similar results. Setup 1 used the initial settings, in setup 2 the 100 most recent games were used to compute new ExT. In setup 3 K was set to 8 and only the 50 most recent games were used for training. Setup 1 and 2 reached their best performance while not using the data from the first, random iteration in five and four respectively. The third setup however reached its peak performance in the second iteration, which still uses the random iteration's data. Its peak performance is lower than the peak performance of setup 1 and 2. This suggests that choosing a value for K that is too small results in a sub-optimal policy. This is on par with the results of the paper by Ernst, Guerts and Wehenkel [8].

Another noticeable effect when comparing setup 1 and 2 is a significant drop in performance of the latter after the fifth iteration. Analyzing how many actions were taken during a match reveals that the number of actions greatly increases once ExT are used in the decision process instead of random action.

Up to 160 movement actions per match were recorded. A possible explanation for this behavior is that the AI agent prefers small movements over long ones and often even stands still. This would mean that long movements are only supplied by the data sets from the random actions from the first iteration which are not present after the fifth iteration. Such small and short movements increase the effective reward for the gold gained in regular intervals, since it means more gold per time. This could also be observed when watching the AI agent during play. The negative effect of this behavior is bloating of data sets since they contain one entry per action regardless of duration. This leads to bigger ExT which again leads to longer times when polling for the next action.

Since the hero is not the only unit attacking enemy creeps, last hitting requires exact timing. This timing is not possible when the agent's current reaction time differs from the reaction time during the last iteration. Using the last 100 games leads to the situation that data is present in the training set which was based on vastly different reaction times. When only the last 50 games are used, the observed reaction times are more recent which reduced scatter which led to a better policy.

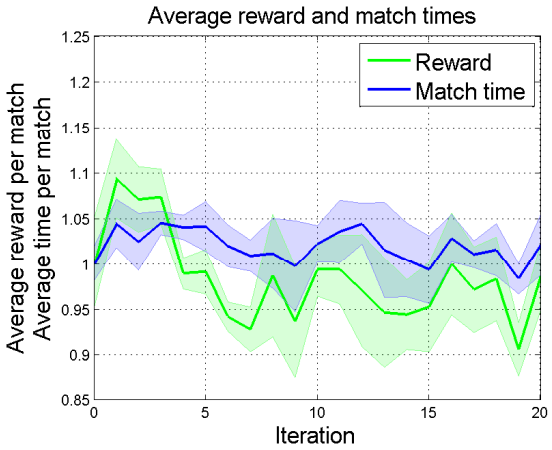

Figure 3.4: Averages of the relative performances of the games with exchanged weights for XP and gold.

Changing the reward function to reward experience points (XP) stronger than gold led to a worse performance overall. Experience points (XP) are gained when the hero is present while an enemy unit is being killed, while gold is only awarded for actually last hitting an enemy. This might have caused passive policies where the agent tries to minimize penalty by avoiding taking damage. By avoiding damage the agent would also not be in a position to score last hits, missing out on the now lower reward presented by earned gold.

Another setup introduced a penalty for returning to the fountain based on the amount of HP the hero was missing. Less HP missing meant higher penalty since returning to the fountain takes time and should only be done when necessary. This setup showed a minor improvement over previous tests without the additional penalty term concerning reward as seen in figure 3.5. Considering that an added penalty term should decrease the reward gained, this increase in performance shows that the penalty term worked as intended. With it the hero stayed on the lane longer before returning to the fountain for regeneration, earning more gold and XP.

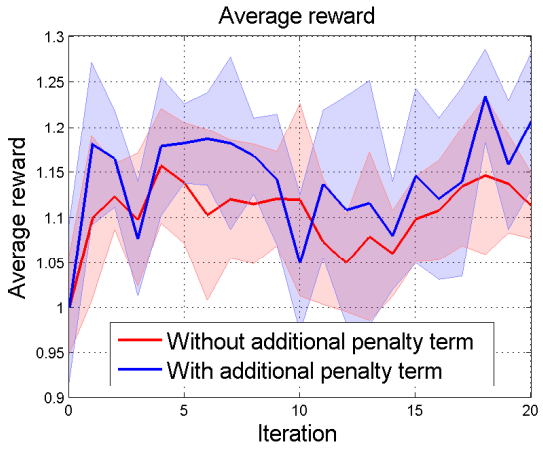

Figure 3.5: Average reward of the standard setting and the setting with additional penalty term for high HP when returning to base.

The last two setups posed significant problems for the ExT algorithm, because both modified which features were used.

The first setup exchanges the creeps' distance to the hero for two features containing the x and y coordinate of the creeps. This setup fails to improve on its performance. The reason most likely to cause this problems is that the position conveys maximum information only when the x and the y coordinate are evaluated. Due to the random sampling of ExT this is not always guaranteed, which leads to the agent not taking optimal actions.
The second setup adds two features which both convey the same information. Those features are the minimal and maximal damage the hero inflicts. Since the AI agent does not buy items, the damage inflicted only changes when the hero gains a level and therefore is redundant to the hero's maximum HP. This setup fails to improve or even retain a good level of performance. The most probable cause for this are the random cuts used in the learning phase. Since there are three redundant features in this setup, the probability for dividing according to this information increases. This leads to degenerate trees and ineffective policies. The performance of those two setups can be seen in Figure 3.6.

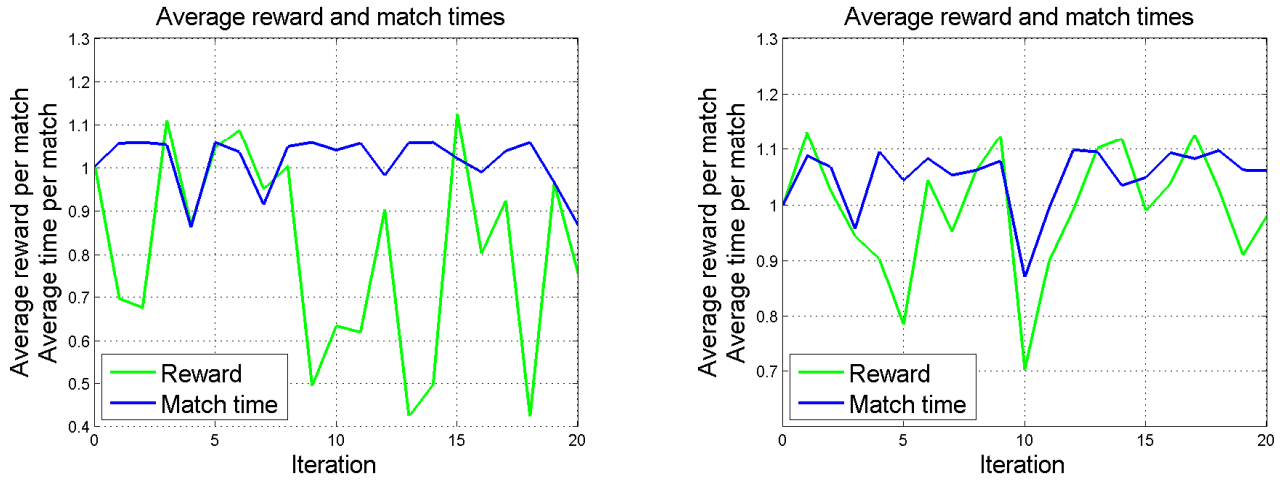

Figure 3.6: The reward and match time of the setup using x and y coordinates (left) and the second setup using additional features conveying redundant information(right).

3.3 Discussion

The results of the experiments show that the algorithms do not work perfectly. Ideal results would be six last hits and denies, which is every last hit and deny possible. The average number of last hits scored by random actions was around two, with a learned policy this value increased slightly for every successful setting tested. This proves a general suitability of the algorithms for learning Heroes of Newerth (HoN).
There are multiple possible reasons for the sub-optimal performance. For one, the support presented by the game itself is aimed at user-written Lua scripts evaluated by the game itself. Lua itself is not suitable for machine learning problems; Matlab on the other hand is suited for processing big amounts of data as is the case for extra trees (ExT) with fitted Q-Iteration (FQI). Adding an interface for Matlab was only possible via the given Lua support slowing down the data exchange between algorithm and game. This added some extra delay to the reaction times of the algorithms which might have been critical for successful last hitting. Another problem might be that since FQI does not build a model of the learned task, it does not learn the consequences of actions taken. Since it takes a set duration before the hero can attack again after an attack was triggered, it is not beneficial to attack without scoring a last hit. In the time it takes for the hero to be able to attack again, creeps might be killed and the agent loses the gold that could be gained from the last hit.
There are two possible solutions for this problem. The first is to simply add a penalty for attacking, adding some domain knowledge to the reward function. This would reduce the agent's use of attack actions, thus increasing the chance of the attack action being ready when a last hit can be scored. The second possibility is to use an algorithm which learns a model. This would be more effort than simply adding domain knowledge to the reward function but it would make the algorithm more versatile. A problem with this approach would be that learning a model requires more resources which could make it more difficult to find a sufficiently fast algorithm. All in all the approach used in the thesis looks promising but needs more fine tuning to provide satisfactory results.

Bibliography

[1] C. Amato, G. Shani, and B.-s. Israel. High-level Reinforcement Learning in Strategy Games. 2010.

[2] S. C. Amend. Feature extraction for policy search, 05 2014.

[3] M. Buro. Call for AI Research in RTS Games. pages 2–4, 2004.

[4] H. Chan, A. Fern, S. Ray, N. Wilson, and C. Ventura. Online Planning for Resource Production in Real-Time Strategy Games. pages 65–72, 2007.

[5] B. E. Childs, J. H. Brodeur, and L. Kocsis. Transpositions and move groups in Monte Carlo tree search. *2008 IEEE Symposium On Computational Intelligence and Games*, pages 389–395, Dec. 2008.

[6] M. Chung, M. Buro, and J. Schaeffer. Monte Carlo Planning in RTS Games.

[7] V. K. Dimitriadis, A. M. G. Lagoudakis, A. K. Mania, and A. N. Vlassis. Reinforcement Learning in Real Time Strategy Games Case Study on the Free Software Game Glest. (September), 2009.

[8] D. Ernst, P. Geurts, and L. Wehenkel. Tree-Based Batch Mode Reinforcement Learning. *Journal of Machine Learning Research*, 6(1):503–556, 2005.

[9] P. Geurts, D. Ernst, and L. Wehenkel. Extremely randomized trees. *Machine Learning*, 63(1):3–42, Mar. 2006.

[10] M. G. Lagoudakis. Least-Squares Policy Iteration. 4:1107–1149, 2003.

[11] M. A. A. Llen, K. R. D. Irmaier, and G. A. R. Y. P. Arker. Real-Time AI in Xpilot using Reinforcement Learning. 2010.

[12] M. McPartland and M. Gallagher. Reinforcement Learning in First Person Shooter Games. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(1):43–56, 2011.

[13] P. Perick, D. L. St-Pierre, F. Maes, and D. Ernst. Comparison of different selection strategies in Monte-Carlo Tree Search for the game of Tron. *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 242–249, Sept. 2012.

[14] M. Pfeiffer. Reinfrocement Learning of Strategies for Settlers of Catan Graz University of Technology. 2004.

[15] F. Sailer, M. Buro, and M. Lanctot. Adversarial Planning Through Strategy Simulation. *2007 IEEE Symposium on Computational Intelligence and Games*, (Cig):80–87, 2007.

[16] D. Silver, R. Sutton, and M. Martin. Reinforcement Learning of Local Shape in the Game of Go. pages 1053–1058, 2007.

[17] R. S. Sutton and A. G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.