Imlementation of an A.I. Player for the videogame League of Legends based on Image Recognition, Discretization and Gradient Optimization

664064 Oliver Struckmeier
663641 (Eric) Tsz Hin Heung

School of Electrical Engineering
Aalto University

December 15, 2017

Abstract

The topic of this project is to create an agent that is able to play 3rd person massive multiplayer online battle arena games (MOBA) like League of Legends, Dota 2 and Heroes of the Storm with the same input as a human player. Image recognition is used to detect objects in the game environment. The environment is then simplified by discretizing into a grid. The state vector $\vec{x}$ contains information about location of the player, location of the enemies, health points and game time. A stochastic policy gradient learning algorithm is used to determine which area of the screen should be clicked or how the agent has to position in order to get a maximum reward that leads to winning the game.

1 Introduction

This report discusses the results of creating an agent that can execute simple tasks in a MOBA game (multiplayer online battle arena game). The game chosen is League of Legends due to its, relative to similar games, lower focus on micro gameplay and fewer macro tactical decisions compared to similar games in the genre. It is inspired by the Dota 2 Bot created by OpenAi [1]. The first objective will be to determine the state and action space as well as the environment in which the agent is positioned. This is achieved with image recognition using OpenCV and TensorFlow is used to detect objects in the game. It would be possible to read game data like positions from the RAM of the computer but this action is considered an illegal action in the game as it provides a significant advantage over a human player. At first it will be presented how the image acquisition and recognition works. Then it will be discussed how we can create a representation of the game using discretization. The discretized environment allows the agent to use learning methods and decision making processes on the environment in order to learn behaviour policies to achieve certain goals by following a policy to maximize the reward for an actions. In the last part it will be discused how succesful and efficient the method was and how it could be improved.

GitHub repository: `https://github.com/Oleffa/LeagueAI`

2 An Introduction and League of Legends

First, some basics and certain terms that will be later used to describe actions, states and the game process have to be introduced.

Figure 1: The game level

2.1 League of Legends

is an online multiplayer game created by Riot Games in 2009. It is usually played between two teams of human player consisting of 5 members on each side. Each player can select one of more than 120 player characters, so called ”Champions” with a unique set of abilites and different roles.

The game takes place in a square shaped static environment or level. Each team has it’s own base either at the bottom left or top right corner of the map. In the base is the main structure called the ”Nexus”. Losing this structure means losing the game. The two bases are connected by three ”Lanes” originating of each base and have neutral territory in between them. When the game starts so called ”Minions”, computer controlled soldiers, swarm from one team’s Nexus to the enemy team’s nexus and attack every enemy minion, character or structure they meet on the way. On each Lane are several towers which are strong deffensive structures and can’t be attacked

by a single player because of their high damage output. It is the minions purpose to serve as a distraction for the player to take down these objectives. Without a human's interaction to the game the waves of Minions would collide in the center of the lanes and attack each other but never make any significant progress due to the fact both sides are balanced and towers provide deffensive capabilites and can only be taken down by clever actions through human players. It is the players task to influence this equilibrium in a way that the enemy base will be conqured and the enemy nexus destroyed. Figure 1 shows a zoomed out overview of the map with the red lines displaying lanes and the Nexus in a red rectangle.

2.2 Evaluation of the environment

This static environment provides a static scenario with static obstacles. Furthermore enemy agents are challenging the agent. The agent can interact with the environment by for example attack minions and thereby changing the balance of the game to his favor. Of course there are many more challenges to the game, especially considering the presence of other players who also try to win the game. For now we focus only on one player, the agent, trying to to win a game without enemies by influencing only the computer controlled environment. Furthermore macro decision making is ignored and the agent is trained to only detect his player character enemy minions and enemy towers to see if the system can be implemented to achieve satisfying results. Once this is the case other game elements can be added step by step.

2.3 Defining a Goal

The goal is to create a basic agent for the videogame League of Legends and teaching the agent to kill enemy minions in order to push through one lane and kill the enemy nexus within a reasonable amount of in-game time. This involves moving in a part of the game environment, determining where enemies and allies are and assessing their threat level or reward toward winning the game.

A long term goal could be to introduce the official game A.I. to the game and see if the agent can beat it in a 1 versus 1 game. But this will require additional features and improvements in the object detection and environment perception.

3 Agent perception and Environment Representation

3.1 Environment Detection

As mentioned in the introduction the agent reads pixels from the screen to capture the game view in real-time and TensorFlow to perform image recognition on the data. The data preparation for Tensorflow and the output of the agents view is handled by OpenCV For this purpose the TensorFlow Object detection API was used [2]. Since League Of Legends is not a realistic game the predefined models for object detection did not work. For this reason a custom model was trained to detect the player character as well as minions and towers.

Figure 2: The Player Character "Vayne" as seen by TensorFlow

As Figure 2 shows, the object detection of the player character works satisfying and the agent still provides roughly 1 update per second which is barely enough to interact with the game in a reasonable time.

Performance The performance of this system is heavily depending on the image recognition as this is the most performance heavy task for the system apart form running the game itself. The python loop that performs the image detection runs about 1 time per second which is still a satisfying number of actions we can execute per second. This

calculation time gets worse the more aditional calculations we execute and the more complex the decision making and learning algorithms are. The performance could be of course improved by using a computer with stronger hardware or moving the object detection process to other hardware and just pass information to the computer running the game.

3.2 State Representation of the Environment using Discretization

The environment in which the player character is placed is limited by the screen and the size of a single pixel on it. It is difficult to define states and position in this continuous environment. A solution for this problem is to discretize the environment. The game does not recognize clicks with single pixel precision and the player character is approximately 40 by 60 pixels large, so it is sufficent to divide the screen into a grid of possible positions relative to the size of the player character while still providing reasonable clicking precision.

Figure 3: The Player Character in a discrete grid

In Figure 3 the red dot symbolizes the position after processing the box (marked in green) around the player figure. Based on the position of the player a grid with sizes based on the detected player character is layed over the whole screen, each representing one unit of the discretized environment with the player being the origin of the grid.

If we now introduce other game objects like minions into this environment we can determine in which cell in the grid the minion is by matching its detected position into the grid. The same works for obstacles which make a cell/position occupied from the perspective of the agent.

Figure 4: A sample game situation from the perspective of the agent

Figure 5: The corresponding game view

Figure 13 shows a sample situation from the game with marked champions, tower and enemy minions. Their positions are marked as red dots and the states they occupy are framed by blue rectangles for the enemy minions and white for the enemy tower. The player state which is also the origin (0|0) of the state space is marked with a red rectangle.

3.3 Interacting with the Environment

The agent makes use of the grid generated by the discretization. As the grids are divided in small enough units, each state can be directly translated into a position that can be clicked by the agent. An action A is defined as a click on the center of a certain sector in the grid: $A(x, y) = mouse\_click(S_{x,y})$

To increase the effectivity of the clicking process we make use of one of the games features called attack move and normal attack.

If a normal click hits terrain it makes the player character move to this position ignoring everything on the way to the target position. If the normal click is executed on an enemy the character stays in place and attacks the target if it is in range or moves towards it and attacks the target when it gets in range. This mode is used for moving the player character around in the game environment. An attack move on the other hand is similar to the normal click but if applied to a position without an enemy it automatically attacks the closest target to the click position. Thereby we can eliminate the lack of precision when an enemy minion just barely is in a certain state but we try to attack it by clicking in the middle of the state and would miss otherwise.

This approach is still not ideal especially when many minions pile up or the object detection fails to detect some objects. For example if we decide to move to a new position but there is already an enemy minion and we accidentally click it instead of moving to the position.

4 Decision Making and Learning Process

The decision making process is a stochastic policy of four possible actions:

- Attack a minion [AM]
- Attack a tower [AT]
- Approach the enemy base (Move to Goal) [MG]
- Retreat [RT]

Each of the possible action will have a corresponding probability to be chosen ranging from 0 to 1, based on the current state $\vec{x}$ and the policy parameter $\vec{\theta}$.

$$\text{Prob}_{[\text{action}]}(\vec{x}, \vec{\theta}) = [0, 1]$$

The individual probability for each action will be normalized and the action is selected randomly based on the normalized probability P.

$$P_{[\text{action}]} = \frac{\text{Prob}_{[\text{action}]}}{\sum \text{Prob}_{[\text{all action}]}}$$

The learning process is a policy improvement process using gradient ascend. Finite difference gradient approximation is used for its simplicity. Every time after a small set of actions is taken (5 actions), the policy gradient is updated and we improve our policy by moving towards the positive gradient in a small magnitude δ. We only update the policy gradient after a small set of actions in order to mitigate the effect of random fluctuations.

The construct of probability of individual action will be described below. In the following text, distance to closest minion, d_{minion} and distance to closest tower, d_{tower}, are normalized into range [0,1].

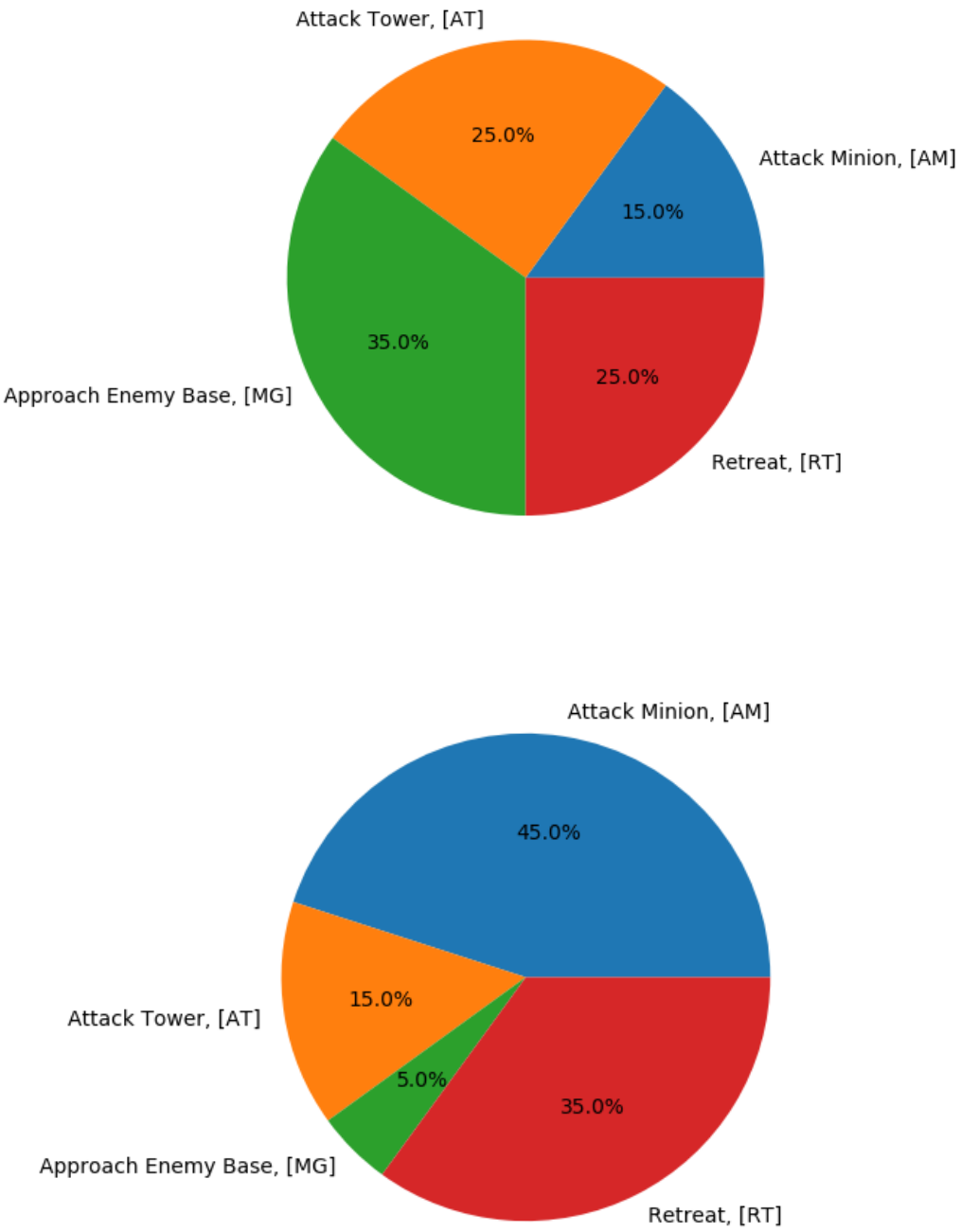

Figure 6: Examples of normalized probability

4.1 Probability to attack a minion

The closer minions are, the higher is their threat level. Therefore the probability to attack a minion is a directly proportional to the inverse distance to the closest minion. The proportionality constant is the product of current percentage HP (%HP) and θ_{AM}. This captures the idea that with higher HP a player is more confident to attack, whereas with lower HP player is likely to be more careful. The θ_{AM} acts as a regularizer or sensitivity coefficient, which will be adjusted by the machine learning algorithm.

$$P_{AM} = (\theta_{AM} \cdot \%\text{HP}) \frac{1}{d_{minion}}$$

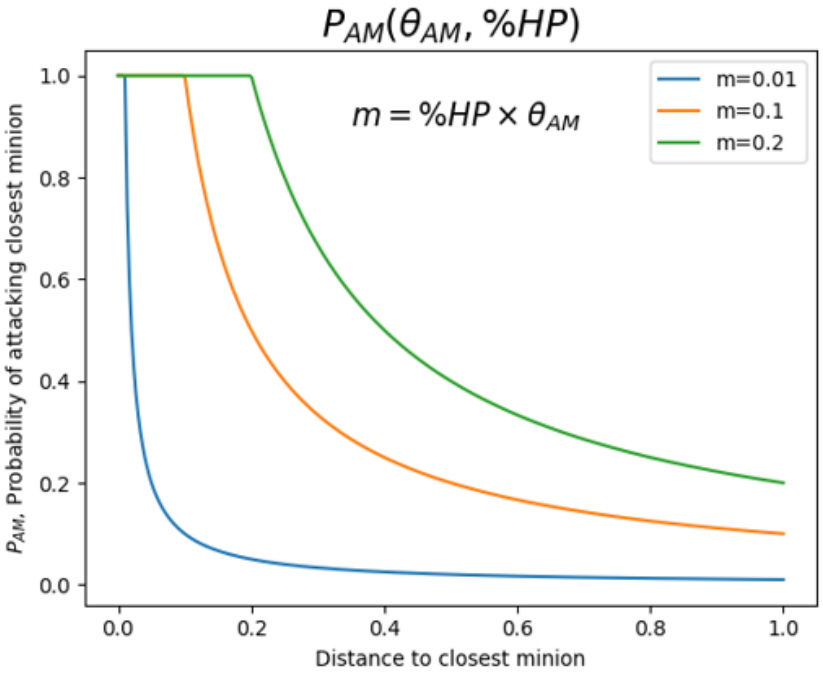

Figure 7: Probability to attack a minion

4.2 Probability to attack a tower

The tower attacks targets in its proximity, prioritizing minions over champions. They inflict heavy damage and thus their danger rise rapidly as the champion approach their attack range and remains a great threat until we are out of their range. Due to the high damage they inflict, champion should not get too close to the tower and instead should remain in the boundary of their attack such that they can engage and retreat agilely and use allied minions as a distraction for the tower.

To capture this behaviour, a linear combination of 2 logistic function is used. One logistic function is responsible for capturing the rapid rise of attack probability and the other one captures the sharp fall of probability. Both functions have the same peak value which is the product of percentage HP (%HP) and θ_{AT}. This captures the idea that with higher HP a player is more confident to attack whereas with lower HP player

is likely to be more careful. The θ_{AM} act as a regularizer or sensitivity coefficient, which will be adjusted by the machine learning algorithm.

$$P_{AT} = \frac{\theta_{AT} \cdot \%\text{HP}}{1 + e^{-5(d_{\text{minion}} - x_{\text{close}})}} - \frac{\theta_{AT} \cdot \%\text{HP}}{1 + e^{-40(d_{\text{minion}} - x_{\text{far}})}}$$

where x_{close} and x_{far} are the location of the rising and falling edge.

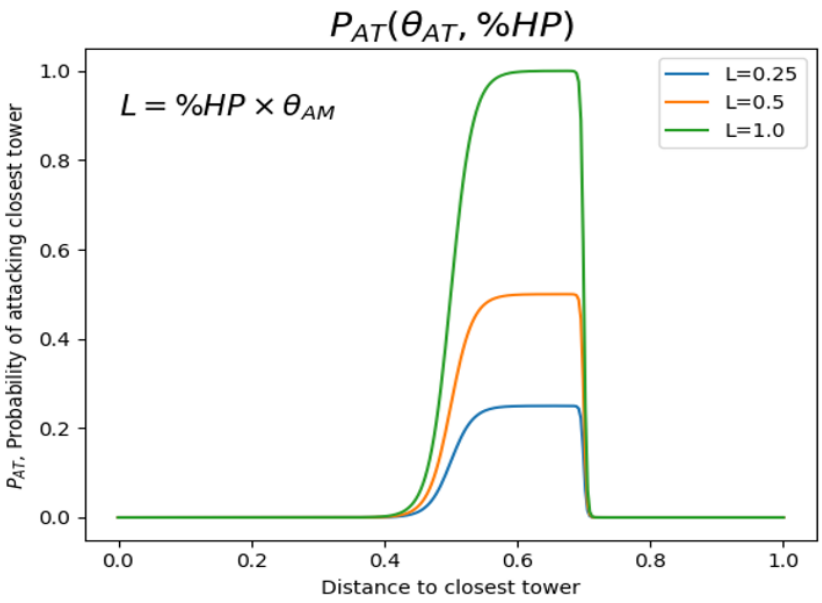

Figure 8: Probability to attack a tower

4.3 Probability to approach the enemy base

The Probability to approach the enemy base is constant in the beginning of the game. After a short period change into a linear relationship which increase from zero as the game time progresses.

This is because in the beginning phase of the game, the champion should move towards the enemy base in order to encounter enemies. After the initial phase, the champion should already be in the battle field, thus the demand to go toward the base will reset to zero, and increase linearly as the game time in order to drive the champion deeper into the enemy force and preventing the champion from standing still.

The θ_{MG} in this case serves again as a regularizer or sensitivity coefficient.

$$P_{MG} = \begin{cases} 1.0, & t_{\text{game}} < t_{\text{cutoff}} \\ \theta_{MG} \cdot t_{\text{game}}, & t_{\text{game}} \geq t_{\text{cutoff}} \end{cases}$$

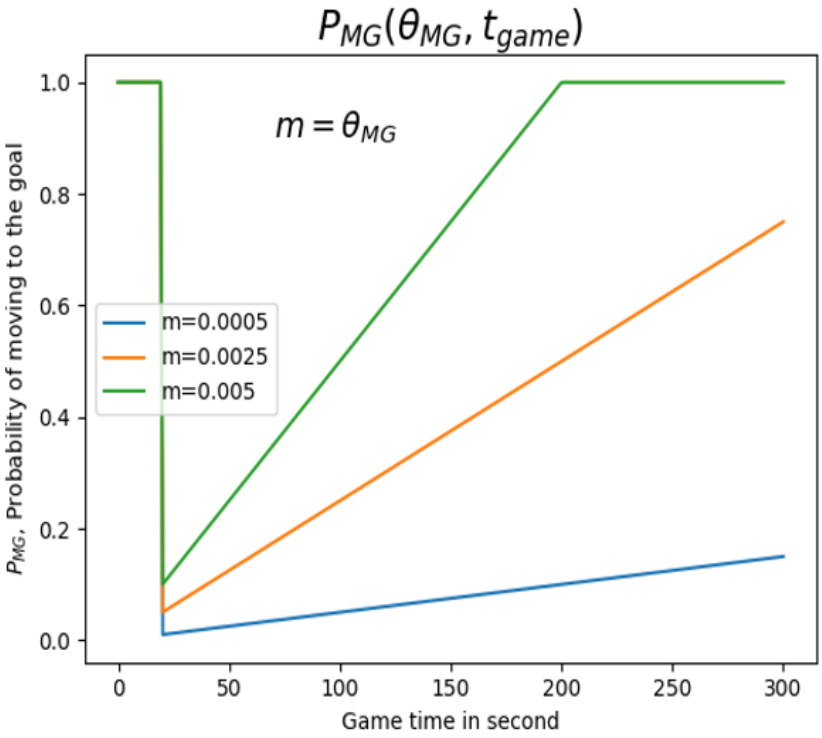

Figure 9: Probability to approach the enemy base

4.4 Probability to retreat

The probability to retreat is based on the simple principle that with lower HP one should step back instead of engage. The model equation is an exponential function of a fraction. The denominator of the fraction is the percentage HP and the numerator is θ_{RT}, which again acts as regularizer or sensitivity coefficient.

$$\exp\left(\frac{\theta_{RT}}{\%\text{HP}}\right)$$

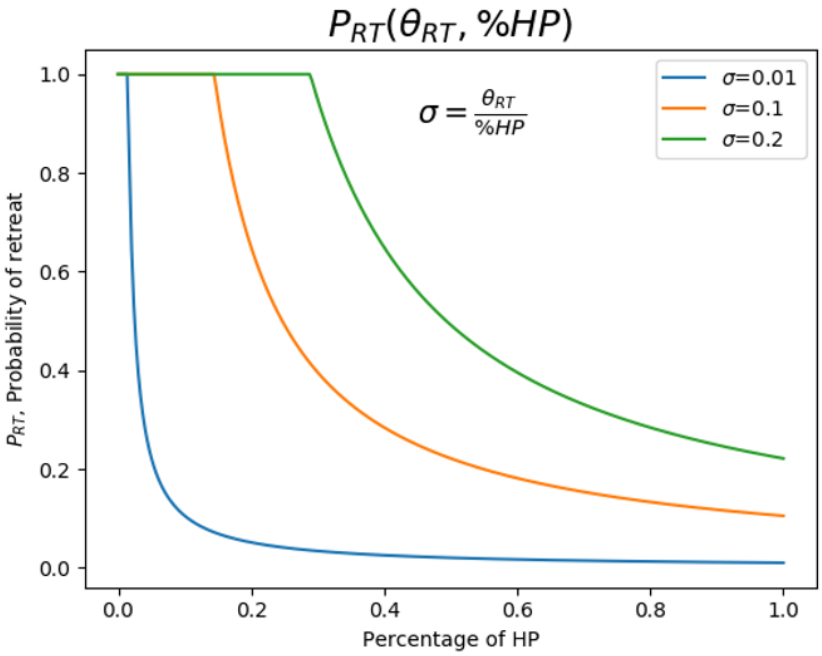

Figure 10: Probability to retreat

4.5 Policy Learning Progress

The learning is implemented using policy ascend. The policy gradient is approximated by finite difference on the reward and the reward is averaged over 5 iteration to reduce fluctuation.

The expected reward, which is required for the finite difference algorithm, is approximated for simplicity reasons by an exponentially weighted moving average.

After every new policy gradient is approximated, the $\vec{\theta}$ is updated by a small amount δ towards the positive gradient direction, which increases the expected reward for any subsequent action selection.

$$
\begin{aligned}
\text{step reward} &: 1 - (\Delta\%\text{HP} + \beta_{textATK} \cdot \#\text{ATKs}) \\
r &: \text{step reward averaged over 5 steps} \\
\nabla_\theta \pi_\theta &: \text{policy gradient approximated by finite difference} \\
R_{\text{ref}} &= (1-\alpha) R_{\text{ref}} + \alpha r \\
R_{\text{ref}} &= (1-\alpha) R_i \quad + \alpha \nabla_\theta \pi_\theta \cdot \delta\vec{\theta} \\
\Delta R &= R_i - R_{ref}
\end{aligned}
$$

The new policy gradient is approximated by:

$$\nabla_\theta \pi_\theta = \frac{\Delta R}{\delta\vec{\theta}}$$

Two runs of the learning progress are plotted below for illustration. Each run consists of multiple games with each being played for 10-15 minutes.

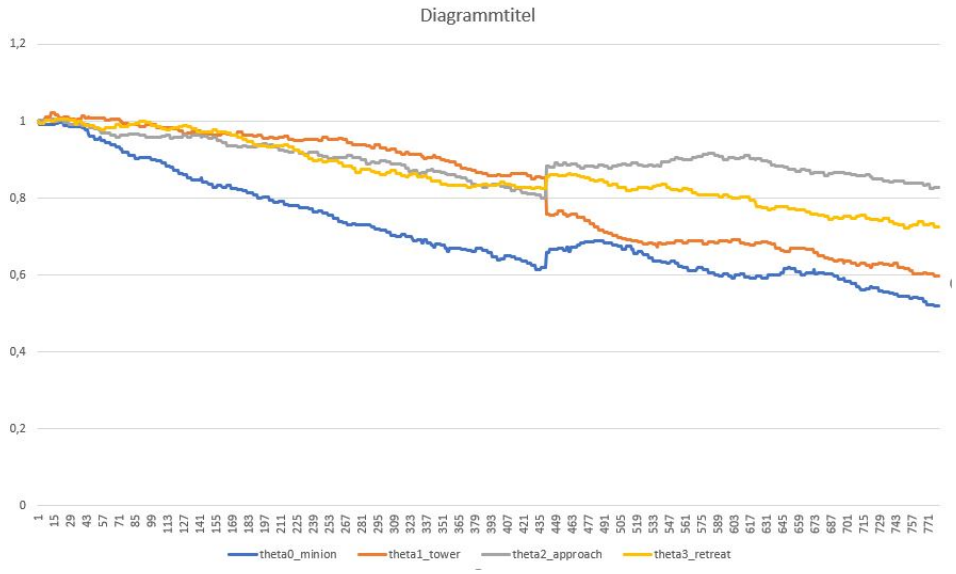

Figure 11: Policy learning progress run 1

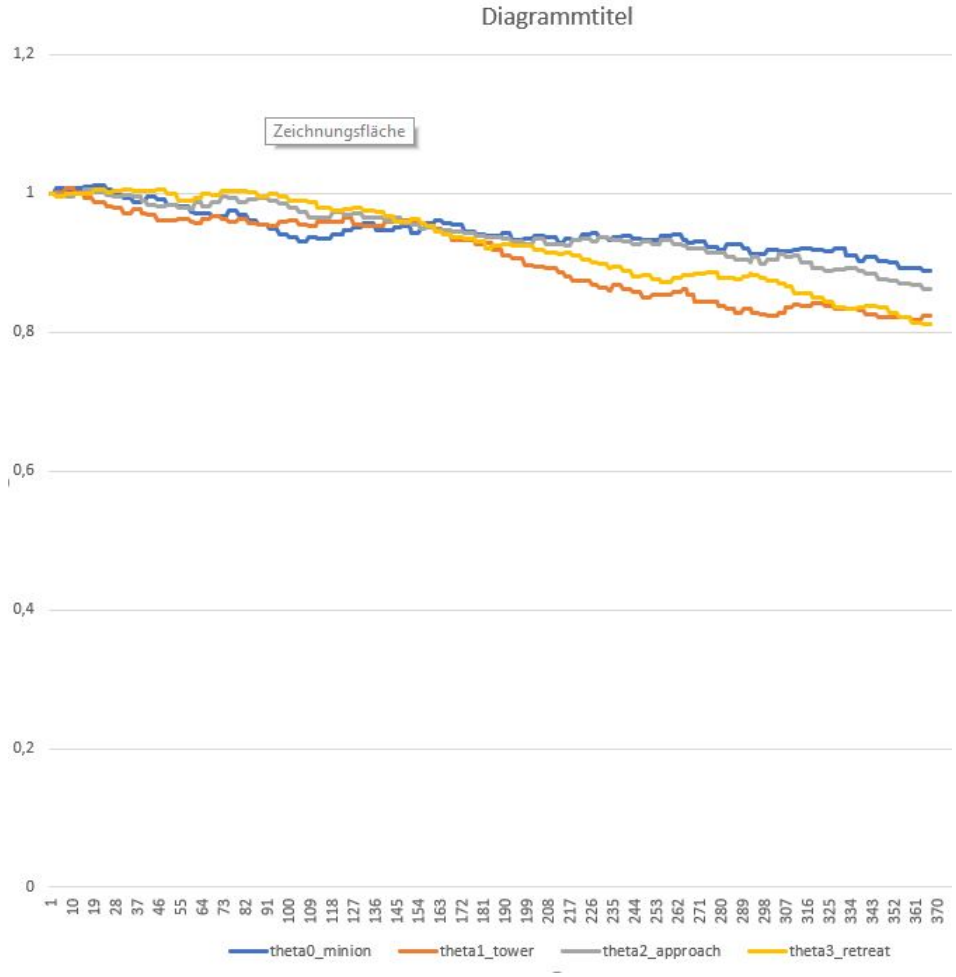

Figure 12: Policy learning progress run 2

Figure 11 shows that all 4 parameters are declining. This was a result of the reward

for attacking being to low and the agent learning that doing nothing and just surviving and having full health yields the highest reward. This is further stressed by the fact that theta0, the probability to attack an enemy minion, is also declining the feastest. The fault around step 450 is explained by a manual parameter change.

After some more experiments and a rework of the rewards for attacking the second run in figure 12 shows a better behaviour. The overall trend of all thetas is still declining, but it can be seen that the thetas for attacking minions and approaching the enemy are declining slower. Also noteworthy is the fact that the probability to attack towers is declining fast. This is a result of the problems mentioned in chapter 5.1. The agent learned that towers are high threat targets and it usualy dies to them due to the slow reactin time which results in high loss of healthpoints and thereby a very low reward. The probability to retreat is also declining fast, as the reward for attacking is making agressive behaviour now more attractive.

5 Rewards, Feedback and learning

The rewards are used to improve the agents decision making. Since there is no interface to the game we have to play every game in real time and thus it is not possible to use many episode for learning. For this reason Q-learning or Deep Learning is not possible and we have to optimize the decision making while playing the game. For this to work the reward is calculated using the following parameters:

- Player Hit Point Change
- The number of attacks executed
- The time the agent survived without dieing

This means the reward is higher the less hit points the agent loses while maximising the number of attacks and surviving as long as possible. In order to get a more reliable reward value we calculated one reward averaged over 5 iterations of the agents logic. This allows us to also take actions into account that are not directly punished, for example moving to a position over the course of a few seconds and then ending up in a bad spot.

5.1 Towers

One problem when calculating the rewards was the delayed reaction to a tower. Since it takes about 1.5 seconds on the test system to react to a change and another 1.5 seconds to perform an action like running away the agent often gets hit a few times by a tower. This results in the agent learning to stay further away from them. Death by tower is also the most frequent reason, as the second threat, minions, are kept at a safe distance properly.

5.2 Evaluation

The following figure shows the reward during multiple runs. For each run the reward starts around 0 and the upwards trend for each episode is the reward for staying alive. So for a few test episodes of about 5-10 minutes it is not possible to make a prediction if the learning is actually improving the reward but as explained in chapter 4.5 the thetas show that some adaption is taking place. Much longer testing as well as running the tests without the constant upwards trend caused by the reward for staying alive are needed. Other improvements like using more parameters for calculating the reward are discused in chapter 6.

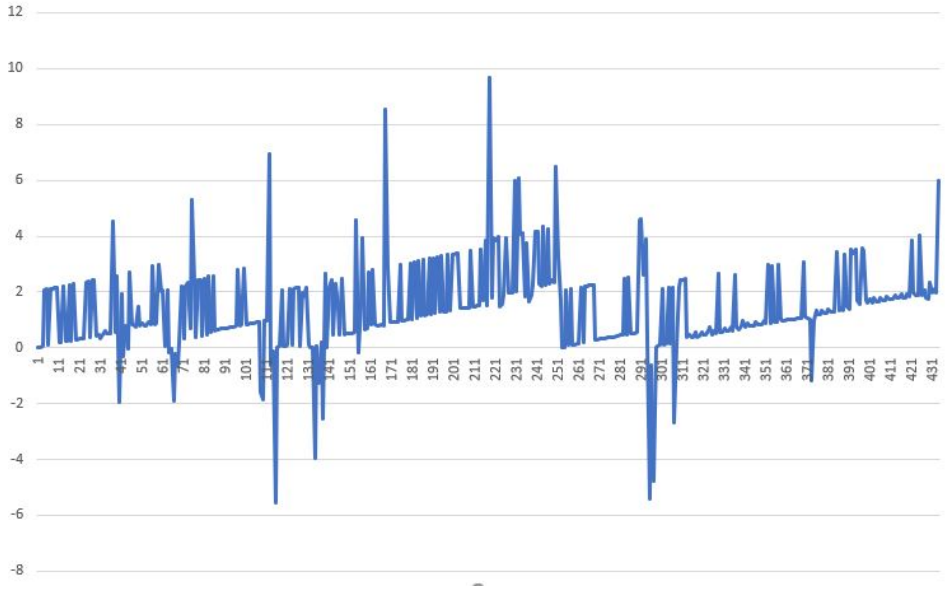

Figure 13: Rewards during run 2

6 Conclusions

The conclusion compares the outcomes to the more advanced Dota2 Bot made by OpenAi[1].

6.1 Evaluation

The goal defined in 2.3 was fullfilled and implemented succesfull. The agent can survive a reasonable amount of time in the game and it has been shown that it possible to use this as the base for a much more elaborate system that could also fulfill the mentioned long term goals. With more development it would most likely be possible to take on the built in bots of the game. The possible improvements are discussed in the following.

6.2 Object Detection

A first step would be to make the object detection more powerful by teaching it more objects like enemy champions, allied minions and structures. The precision of the object detection model is crucial for the precision of the decision making and thus expanding its capabilites will also improve the overal performance. This is a time consuming task as it requires to take and label a lot more aditional pictures and retrain the custom tensorflow model.

6.2.1 Performance

Another big issue is the performance. As stated in 5.1 the reaction times are too slow to prevent damage in the first place and in general more actions per minute means better performance in the game. This requires to accelerate the execution of the agents logic. The most time consuming task is the object detection which is based on tensorflow. This could easily be optimized by using better hardware and running TensorFlow on a GPU instead of a notebooks CPU. We estimate an increase in actions per minute of at least factor 10 resulting in over 500 actions per minute. For comparision good player have about 100-200 actions per minute.

6.2.2 Learning

The problem with learnign was that it is not possible for us to let the agent practice a lot because we have no possibility to speed up the game process. The Dota2 Bot by OpenAI cooperated with the producers of the game and had an interface to the game which allowed it to play millions of games against itself and learn from human players replays. Since we do not have these means the learning capabilities of our agent are limited and we have to rely on learning by doing. This would be improved as soon as the bot is in a state where it can play and win a game on its own and then automate the process of starting new games in order to let the agent train for longer times without supervision.

6.3 Other Challanges

Another challenge is to make the agent aware of the overal state of the game outside of its limited view into the game environment. For example the agent has to know where

the enemy base is and how to get there and react to events that happen outside of its field of view somewhere else on the map. This will be one of the most challenging aspects as the agent can not yet interpret the information of the miniature map included in the HUD of the game or us the itemshops interface to purchase items. Also it is still possible that the agent gets stuck in certain positions. This has to be handled as well to provide a fully autonomous functioning. These challanges will require new approaches and probably a second layer of machine learning/object detection for macro decisions.

References

[1] OpenAi Dota 2 bot,
`https://blog.openai.com/dota-2/`

[2] TensorFlow Object Detection API,
`https://github.com/tensorflow/models/tree/master/research/object_detection`