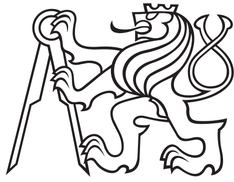
Master Thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Detection of disadvantageous individual decisions for a game with fantastic elements

Bc. Štěpán Műller

Supervisor: Mgr. Pavel Jakubec
Field of study: Open Informatics
Subfield: Artificial Intelligence
May 2022

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **M ller Št pán** Personal ID number: **474557**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Science**

Study program: **Open Informatics**

Specialisation: **Artificial Intelligence**

II. Master’s thesis details

Master’s thesis title in English:

Detection of disadvantageous individual decisions for a game with fantastic elements

Master’s thesis title in Czech:

Detekce nevýhodných individuálních rozhodnutí pro hru s fantastickými prvky

Guidelines:

The student will research state-of-the-art deep artificial intelligence (AI) approaches of decision making influenced by the economical factors such as opportunity cost and its implementability for a game with non-realistic (fantastic) elements and its combination with models for making the decision under risk and uncertainty. Afterwards, the student will implement the AI model and regularization method for detection of disadvantageous individual decisions and their contribution to the team effort (winning / losing the game).
Review different AI approaches for detection of individual decisions suitable for applying within the chosen game rules.
Implement / derive the chosen AI solution and combine it with a chosen classical method.
Test implementation of the solution with artificially generated data and compare them to the real world data.
Evaluate experimentally the solution and compare it to existing techniques used for the prediction in electronic sports for a chosen game.

Bibliography / sources:

de Palma, Andre, Moshe Ben-Akiva, David Brownstone, Charles Holt, Thierry Magnac, Daniel McFadden, and Peter Moffatt et al. 2008. "Risk, Uncertainty And Discrete Choice Models". Marketing Letters 19 (3-4): 269-285. doi:10.1007/s11002-008-9047-0.
Jeong, Yonghyun, Hyunjin Choi, Byoungjip Kim, and Youngjune Gwon. 2020. "Defoggan: Predicting Hidden Information In The Starcraft Fog Of War With Generative Adversarial Nets". Proceedings Of The AAAI Conference On Artificial Intelligence 34 (04): 4296-4303. doi:10.1609/aaai.v34i04.5853.
Wang, Shenhao, Baichuan Mo, and Jinhua Zhao. 2021. "Theory-Based Residual Neural Networks: A Synergy Of Discrete Choice Models And Deep Neural Networks". Transportation Research Part B: Methodological 146: 333-358. doi:10.1016/j.trb.2021.03.002.
Wong, Melvin, and Bilal Farooq. 2021. "Reslogit: A Residual Neural Network Logit Model For Data-Driven Choice Modelling". Transportation Research Part C: Emerging Technologies 126: 103050. doi:10.1016/j.trc.2021.103050.

Name and workplace of master’s thesis supervisor:

Mgr. Pavel Jakubec Fantasy AI Solutions s.r.o.

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **28.01.2022** Deadline for master's thesis submission: **20.05.2022**

Assignment valid until: **30.09.2023**

Mgr. Pavel Jakubec Supervisor’s signature	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt

Student's signature

© VUT v Praze, Design: VUT v Praze, VIC

Acknowledgements

I would like to thank Mgr. Pavel Jakubec for supervising my research. I am also grateful to my partner Vendy for her support.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, May 18, 2022

Abstract

The electronic sports industry has been growing rapidly in the last years and is expected to grow further in the following years. Players are interested in getting better at games, which leads players to watch professional players or even hire coaches. We propose a method to automatically detect mistakes made in games based on artificial intelligence. Our method is going to help players gain more insight into how individual game events affect the outcome of the game and notify them about what they could have done differently to improve the probability of their team to win the game. We trained a neural network on matches of professional players to predict which actions players are going to perform. This allowed us to automatically detect situations where the difference between the predicted and acted behavior was high and the player probably could have performed a better action.

Keywords: neural networks, multi-task learning, game ai, esports analytics

Supervisor: Mgr. Pavel Jakubec
Fantasy AI Solutions, s.r.o.,
Cihelná 718,
411 08 Štětí

Abstrakt

Odvětví elektronických sportů se v posledních letech rychle rozrůstalo a podle odhadů bude růst nadále. Hráči mají zájem se v hraní zdokonalovat, někteří se proto kromě hraní dívají na profesionální hráče nebo si najímají kouče. Navrhujeme metodu pro automatické detekování chyb ve hrách na základě umělé inteligence, která pomůže hráčům získat lepší vhledu do toho, jak jednotlivé herní události ovlivnily výsledek hry a upozorní je na situace, kdy se mohli zachovat jinak, aby zvýšili šanci svého týmu na výhru. Natrénovali jsme neuronovou síť na zápasech profesionálních hráčů aby předvídala, které akce hráč udělá. Díky tomu dokážeme automaticky detekovat situace, kdy je rozdíl mezi předpovězeným a vykonaným chováním velký a hráč mohl pravděpodobně vykonat lepší akci.

Klíčová slova: neuronové sítě, víceúkolové učení, umělá inteligence ve hrách, analytika v esportu

Překlad názvu: Detekce nevýhodných individuálních rozhodnutí pro hru s fantastickými prvky

Contents

1 Introduction **1**
1.1 League of Legends 1
1.2 Our work 4

2 Related work **7**
2.1 Game artificial intelligence 7
2.1.1 AI players 7
2.1.2 Representing knowledge 8
2.1.3 Planning and games 9
2.1.4 Decision making under uncertainty and risk and opportunity cost analysis 10
2.2 League of Legends analysis 11
2.3 Traditional sports analysis 12

3 Statistical models **13**
3.1 Supervised learning 13
3.1.1 Loss functions 13
3.1.2 Logistic regression 14
3.1.3 Tree classifiers 14
3.1.4 Neural networks 16
3.2 Reinforcement learning 19
3.2.1 Fully observable Markov decision processes 20
3.2.2 Value functions 20
3.2.3 Deep reinforcement learning . 21

4 Dataset **23**
4.1 Data structure 23
4.2 Data wrangling 24

5 Solution **27**
5.1 Predicting game outcome 27
5.1.1 Feature engineering 27
5.1.2 Results 30
5.2 Predicting macro decisions 33
5.2.1 State encoding 33
5.2.2 Prediction targets 34
5.2.3 Model 37
5.2.4 Results 38

6 Conclusion **51**

A Bibliography **53**

Figures

1.1 A schema of Summoner's Rift, the most popular League of Legends map. 2

4.1 2D histograms of champion positions in the data. 26

5.1 K-modes clustering of team champion compositions. 28
5.2 The relationship between the number of samples per game in the training split and the final test accuracy. 31
5.3 Prediction accuracy of win prediction models, depending on game time. 32
5.4 Probabilities of the blue team winning throughout sample games from the test set. 42
5.5 Processing of structured multi-modal data using a neural network. 43
5.6 Histograms of absolute differences between predicted and true values. 44
5.7 Histogram of probabilities of movements performed by players on the test dataset. 45
5.8 Champion movement and a 2D visualization of predicted positions at different game states. 47
5.9 Champion movements and predicted movements for all champions at different game states. 48
5.10 Champion movement and a 2D visualization of predicted positions at detected "bad decision" states. . . . 49

Tables

4.1 List of all information available at each game state. 25

5.1 Train and test accuracies of logistic regression. 41
5.2 Regression target means, variances and weights. 41
5.3 Model losses, depending on timestep length. 43
5.4 Ablation study of features. 45
5.5 Ablation study of targets. 46

Chapter 1

Introduction

The electronic sports(esports) market revenue and audience size have been growing rapidly and are expected to grow further in the following years. [1], [2]

Many players of competitive esports games want to get better at playing. Their motivations include a sense of achievement, the social status it brings among gamers [3] or intentions of becoming a professional player.

Some players are willing to pay coaches to watch them play and give them advice on how to improve. Another option to learn is to watch professional players play in tournaments to learn from the best. There are also educational streams where good players play and explain their decisions.

Even with all the options, getting better at the game is difficult. When players watch replays (recordings of the games they played) and have access to the full state of the game, part of which was hidden to them during the game, determining the best move to make in a given situation and identifying mistakes is still not straightforward. Even if some actions seem advantageous at a given moment, it is difficult to reason about their long-term consequences.

We propose an artificially intelligent (AI) method that automatically processes games played by players and gives them the tools and feedback required to improve and gain a better understanding of the game based on statistical machine learning.

1.1 League of Legends

League of Legends (LoL) is a multiplayer online battle arena (MOBA) game. It is played in real-time. Its most popular game mode is played by two teams, each consisting of 5 players, on a square, 2-dimensional map. The map is mirrored and consists of the bases of both teams, three lanes, called *top*, *mid* and *bottom*, the upper and the lower jungle, and the river. On each lane, there are 3 turrets and 1 inhibitor on both sides. In each base, there is a shop, a fountain where champions quickly replenish their health and mana, and the nexus. The schema of the map can be seen in Figure 1.1.

Shortly after the game starts, minions start appearing at each base at regular intervals, marching down each lane. Minions are simple characters controlled by simple game logic. They will attack any opponent they en-

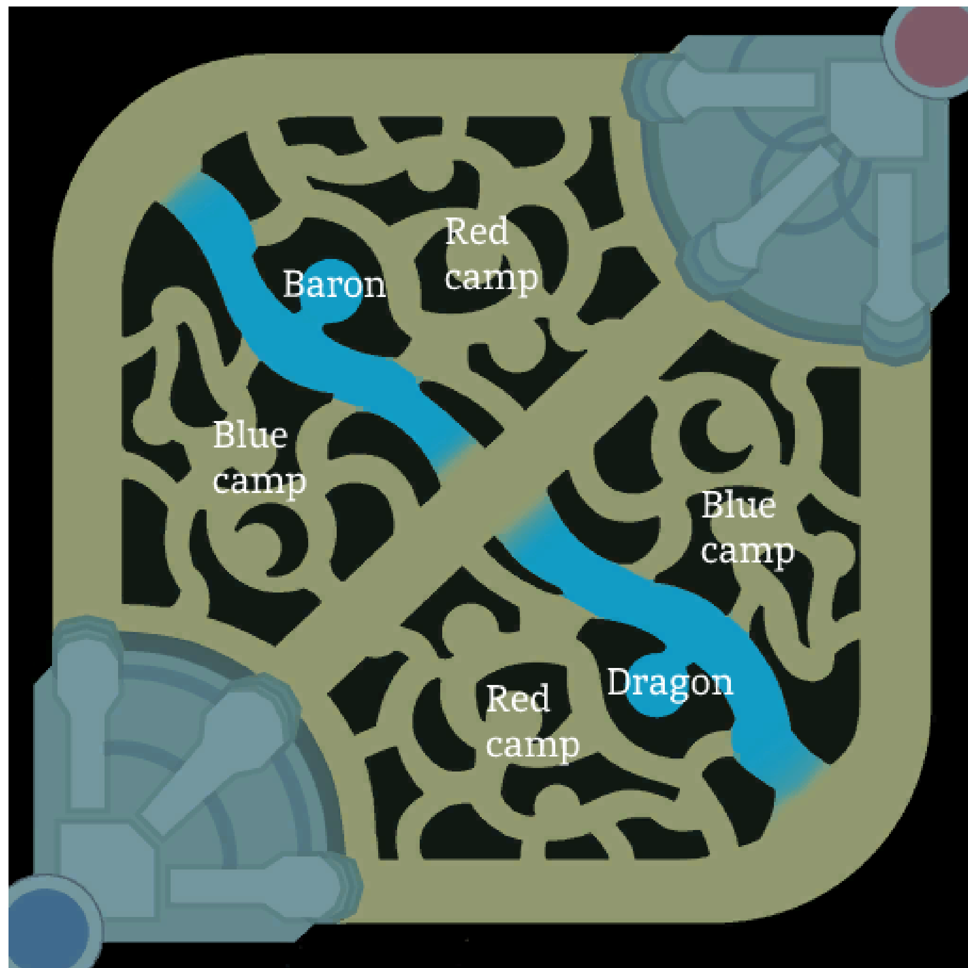


Figure 1.1: A schema of Summoner's Rift, the most popular League of Legends map. Spawn locations of important monsters are shown. The blue team has a base in the bottom left corner. The red team has a base in the right top corner. [4]

counter, eventually destroying the opponent team's nexus if not stopped. The game ends when one of the nexuses gets destroyed. Turrets attack any nearby enemy with powerful attacks, hindering the advancements of enemy minions until destroyed. An inhibitor is a passive structure that respawns five minutes after it is destroyed. While it is destroyed, the opponent team spawns an additional minion, called *super minion*, on the lane on which the inhibitor is destroyed, putting additional pressure on the lane.

Each player controls a unique character, called *champion*. Before the game starts, players choose their champion in the *pick phase*. Currently, there are more than 140 different champions available, each one having different looks, abilities, and stats.

All players have the *recall* ability which allows them to teleport back to their base after staying still and channeling for a few seconds. This ability is interrupted by any damage they receive so it cannot be used in combat. The

purpose of this ability is to quickly get home to replenish health and mana, buy items, or defend the base.

Before the game begins, players have to select 2 summoner spells. Summoner spells are special abilities with high cooldowns. They are called *summoner spells* because, in the game, the player is referred to as the *summoner* who summons the champion. Cooldown is the time for which the player needs to wait after using an ability before it can be used again. Summoner spells have cooldowns in the range of minutes. Summoner spells include but are not limited to *heal* which heals the champion and nearby teammates, *ingite* which ignites an enemy champion, causing them damage over a short duration, *smite* which deals a large amount of damage to neutral monsters and *teleport* which allows the player to teleport to a friendly minion or structure anywhere on the map.

Champions get stronger as the game progresses. If a champion is within a certain range (called “XP range”) of an opponent and the opponent dies, the champion will gain experience from that foe’s death. After accumulating a certain amount of experience, champions level up which increases their stats and allows the player to choose a skill to unlock or to improve the level of one of their already unlocked skills. If a champion dies, they respawn after a certain time. Respawn time increases as the game progresses and as champions level up.

Players also gain gold when they kill an enemy minion or a neutral monster, but they have to be the ones who dealt the killing blow, called *last hit*. The entire team gets a gold reward if they manage to slay some of the powerful neutral monsters, dragon and Baron Nashor, or when they destroy an enemy turret. Gold can be used to buy items in the game shops located on each of the team’s bases.

In earlier stages of the game, instead of Baron Nashor, Rift Herald spawns in his place. After killing the Rift Herald, the team that last hit him can pick up an item from the ground that allows them to spawn the Rift Herald somewhere on the map for their team to start marching and destroying enemy turrets on the nearest lane.

There are different types of elemental dragons and then there is the elder dragon. Slaying elemental dragons grants a permanent team-wide buff. Slaying Baron Nashor grants a powerful team-wide buff called “Hand of Baron” for 180 seconds. Slaying the elder dragon grants a powerful team-wide buff called “Aspect of the Dragon” for 150 seconds.

Besides the lanes, there is the jungle, where neutral monsters can be killed for experience and gold. Some neutral monsters also grant a temporary buff to the champion who slew them.

The strategy adopted by the majority of the players, usually referred to as the current *metagame*, is for each team to split their half of the map, assigning each team member a role. The *top* takes the top lane and is usually played by a durable champion. The *mid* takes the mid lane and is usually a mage, a character with powerful spells. The *attack damage carry* (ADC) takes the bot lane, along with the *support*, a character whose role is to grant

vision to the team by placing cameras on the map, called wards, to deny the opposite team's vision by destroying the opponent team's wards, and to help the ADC on the lane. Lastly, the *jungler*'s role is to clear the jungle. The jungler may also temporarily replace a teammate on a lane who died or needs to recall back to replenish health or mana or to buy an item. The jungler may also try to ambush the opponents on lanes. The move to go on another lane in an attempt to surprise and kill the opponents is called a *gank*.

We divide the wards placed by players into 3 types, sight wards, which are invisible but only last for a limited time, control wards, which reveal other invisible units, including sight wards, and farsight wards, which only have a small vision range but can be placed at a high distance. Sight wards include stealth wards and totem wards from the in-game items.

Each champion should place wards around the map, warding cannot be entirely delegated to a single-player. Firstly, wards have to be placed from a relatively small distance and the *support* cannot be expected to be present and ward on all parts of the map. Secondly, there are limits on the number of active wards of a certain type a single-player can have placed at the same time. If a champion places another ward while being at their limit, the oldest ward of that type placed by the player disappears.

Early in each game, all champions are relatively weak. They have to perform a lot of attacks to slay minions and they die to just a few turret shots. Mistakes in this part of the game do not matter as much because the respawn times are short. The main focus of the early game is to obtain gold and experience by last hitting enemy minions or neutral monsters. Later in the game, champions get stronger and can take on the Rift Herald, Dragon, or Baron Nashor and kill multiple minions by casting a single spell. A single death at this part of the game can lead to a long disadvantageous 4v5 situation which can result in a loss of the game.

The game rules and the map are constantly evolving. Additional content is being introduced, including new playable champions. Adjustments are also being made that balance the game. Players have to keep adapting their play styles to these changes.

1.2 Our work

The performance of players can be split into two parts, micromanagement or *micro* and macro-management or *macro*. Players have to make strategic high-level decisions about where to go on the map and what to do. These high-level decisions are called macro. Players also need to master the controls and combos of their characters in combat. This is called micro.

In our work, we focus on the macro part of playing games. Learning complete MOBA AI agents, that is both micro and macro, using machine learning is computationally very expensive [5] [6]. We simplified the task of learning to reason about LoL to save computation time but still get useful results.

The other reason we focus on macro is that we think it has a higher

potential to bring value to players. To have good control of the champion, players have to practice for hundreds or thousands of hours to automate certain action sequences. On the other hand, players have time to consider macro decisions during games and can improve their decision-making very quickly by changing their views about the game. The goal of our work is to deepen players' strategic understanding of the game. For this purpose, we developed two models.

One is an interpretable win prediction model. With it, we can show the player how did each game event affect the chances of both teams winning the game. We also show an example of how the visualization of the model output could look like on two selected games. This model should improve the players' intuition and decision-making by showing the effect of different game objectives and events on the expected win rate of both teams.

The second model we developed is a neural network that predicts the position of a champion in the next state and the expected rewards they accrue based on the current game state. By training this neural network on a dataset of games played by professional players, we get a behavior model of professional players. We then consider the model predictions as good decisions and realistic acquirable rewards. This allows us to compare the decision made by the player at each game state and the rewards they acquired with the predicted ones. We can then give positive and negative feedback based on whether the player was able to achieve more or less and where on the map they decided to go. In this work, we focus only on detecting disadvantageous decisions, even though the usage of this model for detecting advantageous (surpassing expectations) decisions is straightforward.

Our proposed AI method can be used as a submodule of a more complex system that would provide feedback to players after games in a form of post-game analysis. We also provide example visualizations of the outputs of our models as an inspiration for how they could be presented to players.

Chapter 2

Related work

2.1 Game artificial intelligence

2.1.1 AI players

Board games like Chess and Go were considered a grand challenge in AI. Playing these games can be viewed as a complex planning problem.

In 2016, Silver et al. published AlphaGo [7]. AlphaGo combines deep reinforcement learning (DRL) with Monte-Carlo tree search (MCTS), utilizing the discrete action and state-space of Go. First, they pre-trained deep neural networks using supervised learning on human expert games. Then, they trained the networks further using DRL and self-play. AlphaGo achieved a 99.8% winning rate against other Go programs and defeated the European Go champion Fan Hui in a five-game match 5 to 0.

In 2017, Silver et al. published AlphaGo Zero [8]. AlphaGo Zero achieved superhuman performance without human domain knowledge, except for the knowledge of the game rules, and also beat the previous AlphaGo 100 to 0. AlphaGo Zero was trained without any human expert games, using just self-play.

In 2018, Silver et al. published AlphaZero [9], a general algorithm able to master chess, shogi, and Go through self-play. AlphaZero can be used on games that can end in a draw, which is the case for both chess and shogi but not for Go. AlphaZero also does not use any game-specific knowledge, like exploiting the symmetries in a Go board. They used the same convolutional neural network architecture as AlphaGo Zero for chess, shogi, and go. AlphaZero can be viewed as a DRL algorithm with MCTS as a policy improvement operator.

Chess, Shogi, and Go are fully observable 2-player board games with deterministic rules in which players take turns playing actions. Compared to real-time computer games, they are shorter in the number of time steps each game lasts and their state and action spaces are discrete and relatively small.

In 2013, Mnih et al. [10] presented the first successful DRL model that learned to play ATARI games using the raw game screen as input. Their method even surpassed human experts on three of the games.

In 2020, Schrittwieser et al. generalized the AlphaZero algorithm further

and published MuZero [11]. The need to know the game rules was removed. During the tree search, MuZero uses a learned dynamics model instead of the game rules. MuZero also works on single-player games on top of two-player zero-sum games. MuZero achieved state-of-the-art performance on ATARI games, comparable results on Chess and Shogi to AlphaZero, and surpassed AlphaZero in Go with fewer parameters.

StarCraft II, Dota 2, LoL, HoK, and Quake III CTF are partially observable real-time computer games. StarCraft II is a real-time strategy game played 1 vs. 1 in which both players control multiple units. Dota 2, LoL and HoK are MOBAs. MOBAs are usually played 5 vs. 5 and each player usually only controls a single unit with unique abilities. Quake III CTF is a multiplayer first-person shooter game.

Recently, artificial agents that can play multiplayer computer games on a superhuman level have been developed. StarCraft II [12], Dota 2 [5], Quake III Capture the Flag (CTF) [13] and Honor of Kings (HoK) [14] have all been solved using large-scale DRL and self-play.

For games with imperfect information, which all of the above-mentioned multiplayer computer games are, the agents played against pools of previous versions of the agents to prevent the agent from only learning to counter its current strategy and getting stuck in a rock-paper-scissors-like cycle.

Besides reinforcement learning, an agent playing HoK at the level of top human players was developed using only supervised learning on high-quality ranked games. Ye et al. [6] chose games played by the top 1% of human players, in which the overall score of the player exceeded 90% of the scores of players using the same hero. They removed the other games from their training set.

2.1.2 Representing knowledge

When optimizing the AI agents using DRL to win games as the main task, the agents learned to represent important knowledge from raw game data.

In Dota 2, Berner et al. [5] evaluated the agents' understanding by predicting various game features from the agents' Long short-term memory (LSTM) [15] state:

- Win probability
- Net worth rank: Which rank among the team (1-5) in terms of total resources collected will the hero be at the end of the game?
- Team objectives/enemy buildings: Whether the hero will help the team destroy an enemy building.

They then used the win probability predictor to develop a drafting algorithm. During pick phase, in which heroes are drafted, they used it to decide which hero to pick next. Because the hero pool was limited to just 17 heroes, they could precompute the win rate of every possible lineup and then use the

minimax algorithm to draft optimally with respect to the predicted win rate at the start of the game.

Similarly, the HoK DRL agents [14] used win rate prediction for drafting. The authors managed to teach the agent to play games with a larger hero pool of size 40, therefore an algorithm based on Monte-Carlo tree search [16] was used instead of minimax.

In Quake III CTF [13], the agents' observations were the raw pixels of the game. Using logistic regression on the internal state of the agent, the authors were successfully able to answer more than 70% of binary questions like "Do I have the flag?", "Did I see my teammate recently?", and "Will I be in the opponent's base soon?". The agent even developed neurons whose activation corresponded to the answers to some of these questions. For example, a neuron that was active if and only if the agent's teammate was holding the flag.

Generative adversarial networks (GANs) [17] were used to develop Defoggan [18], a model which was able to successfully remove the "fog of war" and predict hidden information in StarCraft II from previously observed game states.

2.1.3 Planning and games

Let us consider the following classical planning state model:

$$P = (S, A, \gamma, s_i, s_g)$$

S is a set of states, A is a set of actions, γ is a transition function that maps each state-action pair to a state, $s_i \in S$ is the initial state, and $s_g \subset S$ is a set of goal states.

A sequence of actions is a plan if by applying the actions in order we get to a goal state from the initial state s_i.

Classical planning can be used to find plans for single-player games. However, we must extend it to play multiplayer games because it assumes a static environment. In multiplayer games, the state also changes because of actions performed by other players.

Let us take Chess as an example. If we had a model, that is, a probability distribution over the moves the opponent takes in each state, we could formulate the process of playing against this opponent as a Markov decision process and theoretically solve it optimally, obtaining a policy that would tell us what is the best action to take against this opponent in each state. However, there are too many states to compute this in practice.

Today, DRL is used to approximate policy functions and is the current state-of-the-art for both board games and video games. For board games and ATARI games, these policies have successfully been improved by planning, either using the known game rules or a learned model. [11] State-of-the-art DRL agents that play multi-player computer games do not plan ahead. They estimate the best action to take at each step using just the current state or a

small recent history of states. This can be viewed as reactive planning and is also the approach we use for planning where to go on the map in LoL.

2.1.4 Decision making under uncertainty and risk and opportunity cost analysis

Playing partially observable games against human opponents can be viewed as decision-making under uncertainty. We could try to discretize the actions a player can take in different parts of a game of LoL using domain knowledge and use a discrete choice model for decision-making under uncertainty to analyze the decisions taken by players. [19] [20] [21]

There are several problems with this approach. In our opinion, discretization would simplify the problem too much. State-of-the-art DRL agents and usually even amateur teams are able to beat rule-based hand-scripted agents in MOBAs. [5] As the state-space of League of Legends is structured and complex, it would be rather difficult and likely inaccurate to try to estimate the actions available to a player in a given state and the probabilities of their outcomes using handmade rules created by domain experts. We would constrain our reasoning to a set of predefined discrete actions. However, our model would not work well if the players played differently than we expected.

For example, in the early game, it is important that both teams have champions on each lane to collect the gold and experience that minions provide on death. We could compute the opportunity cost of the ADC leaving the bottom lane to try to kill the enemy mid laner. Let us assume that it would not be economically worth it to do at the start of the game. But when exactly would it start being worth it? How likely is the ADC to succeed in killing the enemy mid laner if they decide to do it? This depends on many things, including the combinations of all involved champions, whether there is an enemy ward on the path from bottom to mid, the exact position of the enemy mid laner, and more.

If we could compute the economical aspect of gold and experience, would it be enough to consider only these? If the enemy mid laner gets killed, maybe the team can kill the dragon or the Rift Herald. Maybe the enemy jungler is going to go on mid to kill minions and that will relieve the pressure on the bottom and top lanes and allow our team's jungler to successfully gank one of these lanes. Or, if the ADC does not succeed, maybe the enemy ADC is going to push the bottom turret. While the ADC is absent, the support would be last hitting minions on the bottom lane. What is the economic cost of the fact that the support, rather than the ADC, gets some amount of gold from minions?

Our model predicts rewards while considering the entire game state. We can use our model in any state in any game and consider the predicted expected reward as an opportunity which we then compare with the opportunity chosen by the player and the rewards he received. We can then analyze the opportunity cost of the player's decision.

2.2 League of Legends analysis

Data from games played by real players was mined to solve various game sub-problems like item recommendation (picking the final 6 items to build during the game given a lineup) [22] [23], champion drafting [24] or game outcome prediction [25] [26].

Kim et al. [27] extracted game information from games using image processing and used custom hand-crafted metrics, namely Split Score and Rotation Score, and used them to cluster different playstyles. They compare their system to the popular website op.GG [28].

We are interested in identifying important moments in which a player influenced or could have influenced the final result of the game.

Maymin [29] extracted a large amount of data from live LoL matches using computer vision. Based on simple features, namely game time, team kills, turrets destroyed, dragons killed and Barons killed, they created an interpretable in-game win probability prediction model.

The usefulness of the basic end game statistics, namely the kills, deaths and assists of each player, are limited when it comes to analyzing an individual's impact on the outcome of the game. Some kills only give the killer some amount of gold and their team is not able to get anything else out of them while other kills can lead to capturing an objective leading to a victory, for example later in the game. Some deaths are worthless and some can be worth it, for example, if the player managed to destroy the opponent team's inhibitor before they died.

The authors propose so-called *smart kills* and *worthless deaths*. Smart kills are kills that increase your team's estimated win probability, worthless deaths are deaths that do not increase your team's estimated win probability. The authors show that, contrary to regular kills and deaths, there is a much higher correlation between an individual's performance and a team's performance when the individual performance is measured using smart kills and worthless deaths.

Using this in-game win probability prediction model, they developed several advanced features which correlate highly with how does an individual's performance influence the overall winning probability:

- Abilities
- Gold
- Survivability
- Time management
- Deaths
- Favorable fights

There are also commercial tools available that provide after-game analysis. These tools include the website OP.gg and Blitz [30]. These tools however

only provide feedback through visualization of transpired in-game events, handmade rule-based advice, and comparison of individual metrics to those of other players. To our knowledge, there is currently no tool available that uses machine learning for after-game analysis.

2.3 Traditional sports analysis

The problem we are trying to solve for LoL is similar to traditional sports analysis. Ronald et al. [31] used an LSTM model to predict how many yards the ball-carrier is expected to gain from their current position. Quarterbacks in the national football league were evaluated using tracking data. [32] The advantage of esports is that millions of games are played daily, producing a large amount of data that can be analyzed.

Chapter 3

Statistical models

3.1 Supervised learning

In supervised learning, we have a dataset

$$D = \{(x_1, y_1), ..., (x_n, y_n)\}$$

consisting of n input-output pairs. The goal is to approximate the true, unknown function f^* which maps each input in the input space to its correct output in the output space with an approximating function f. To do this, we usually formulate the problem as an optimization task where the goal is to minimize the difference between the correct output y_i and the output of the approximating function $f(x_i)$. This difference between the two is quantified to a single number using loss functions.

3.1.1 Loss functions

Regression

Different loss functions serve different purposes. For regression, where the output is a quantity or a tensor of quantities, the mean-squared error (MSE) loss function is usually used:

$$l_{MSE}(f(x_i), y_i) = (y_i - f(x_i))^2$$

Classification

For single-label classification, the output is a vector of probabilities. The j-th element of the vector indicates the probability that the input x_i belongs to the j-th class and y_i is the correct class. We then commonly use the cross-entropy (CE) loss:

$$l_{CE}(f(x_i), y_i) = -log(f(x_i)_{y_i})$$

.

3.1.2 Logistic regression

Logistic regression (LR), in its simplest form, is a statistical model that uses a logistic function to model the probability of a binary dependent variable.

Let vector x be the observation and β the parameters of the model, consisting of the bias β_0 and the feature weights β_1. The probability that X belongs to class k is then

$$P(k|x, \beta) = \frac{1}{1 + e^{-k(\beta_0 + x\beta_1)}}, k \in \{-1, 1\}$$

We learn the parameters β of the LR model by using the maximum likelihood estimation. The optimal β^* maximizes the log-likelihood $l(\beta)$ of the dataset D:

$$\beta^* = argmax_\beta(l(\beta))$$
$$l(\beta) = \sum_{(x_i, y_i) \in D} ln(P(y_i|x_i, \beta))$$

It is not possible to compute the optimum using a closed-form expression but we can use an iterative method like gradient descent.

Logistic regression assumes that the observed variables x_i are independent. When working with real-world data, this assumption can be difficult to guarantee.

We use LR to predict the probability of which team in a game of LoL is going to win, given the current state of the game. We denote the binary result of the game as classes -1 and 1. The logit of the probability that team 2 wins is then modeled as a linear combination of the input variables.

3.1.3 Tree classifiers

Decision tree

Decision trees are a non-parametric method that uses a tree-like structure of decisions. In machine learning, they can be used for both classification and regression.

Decision trees are constructed from a dataset. We start by creating the root of the tree with all samples x_i, y_i from D in it. The goal of the decision tree is to group samples from the same class together. To expand a node, we apply a binary rule which splits the samples present in the node into two parts, one part for which the rule holds and the other for which the rule does not hold.

There are many ways to create a decision tree for a given dataset. We want the decision tree to be small to save memory and have fast inference. A common greedy approach, where locally optimal decisions are made at each node, is to choose the split which yields the largest information gain at each node.

Let D_N be the set of samples at node N and S a split which splits D_N into D_N^{left} and D_N^{right}. Let the probability that a sample in a set D has class y be

$$P(y|D) = \frac{1}{|D|} \sum_{(x_i, y_i) \in D} I(y_i = y)$$

The entropy of a set of samples D is then

$$H(D) = -\sum_y P(y|D) log(P(y|D))$$

The information gain of a split S at node N with dataset D is then

$$G(D_N, S) = \frac{|D_N^{left}|}{|D_N|} H(D_N^{left}) + \frac{|D_N^{right}|}{|D_N|} H(D_N^{right})$$

A commonly used algorithm called Classification and regression trees (CART) splits samples based on a single variable x_i and a single threshold c:

$$x_i \leq c$$

CART uses the best split among all variables and thresholds.

Some of the advantages of decision trees are that they are interpretable and that they do not require much data preprocessing. Numerical features do not need to be normalized because the scale does not affect possible splits. They can also theoretically be used directly on categorical data with rules like $x_i = c$. In practice, to be able to use implementations similar to CART, categorical data is in one-hot encoded. The possible splits are then equivalent.

The interpretability of decision trees comes from the fact that we know which set of rules holds for the current observation and why it was assigned a given class. Decision trees can also be visualized.

Disadvantages of decision trees include that they are piecewise-constant and not smooth. Decision tree learners can also create overly-complex trees that classify the training dataset well but do not generalize well to unseen data. This is called *overfitting*. [33]

Random forest

Random forest is an ensemble learning method that creates multiple decision trees. For classification, the class predicted by a random forest is the class selected by most trees.

Random forests utilize the idea of bootstrap aggregating or bagging. Each sub-tree is trained on a different subset of the training dataset, created by sampling randomly from the original dataset with replacement. This technique attempts to reduce the variance of the model without increasing its bias.

Random forests create trees that only consider a random subset of features when considering the best split for a node during node expansion as another bagging technique.

Random forests usually outperform raw decision trees. An advantage of random forests is that their decision trees can be constructed in parallel.

Gradient boosted decision tree

Gradient boosted decision trees are another ensemble learning method.

They use a different approach to ensembling than random forests. Instead of reducing the variance of low-bias models, gradient boosted decision trees ensemble low-variance high-bias weak decision trees of fixed-size in an attempt to reduce their bias.

Gradient boosted trees are constructed iteratively, each following tree attempting to compensate for the error of the current ensemble.

A disadvantage of gradient boosted decision trees is that their construction is difficult to parallelize because the prediction targets for each decision tree depend on the ensemble of all of the previous decision trees.

3.1.4 Neural networks

Neural networks are a parametric method. They are usually used to approximate functions. They usually consist of multiple layers of neurons with trainable parameters. The simplest layer is the fully-connected layer.

A fully-connected layer has trainable parameters W, b takes as input a vector X and outputs another vector $Y = WX + b$ where W is the matrix of the weights of the neural connections and b is the bias vector.

Training neural networks

The key to training neural networks is that their layers are differentiable. Let us have a neural network with parameters θ that we want to optimize with respect to a dataset D and a loss function l.

We sample repeatedly input and output tensor pairs x and y, the network output $f(x|\theta)$ is computed, a loss $l(f(x|\theta), y)$ is computed with respect to the supplied true output y, and a gradient of the loss is computed with respect to the network parameters:

$$\frac{\partial}{\partial \theta} l(f(x|\theta), y)$$

This gradient, multiplied by a learning rate hyperparameter, is then subtracted from the network weights θ. This is one training step that lowers the loss of the network with respect to the provided input-output pair.

If we do this repeatedly with input-output pairs randomly sampled from D, this algorithm is called stochastic gradient descent.

Activation functions

To make it possible for neural networks to represent something other than an affine transformation $y = Wx + b$, non-linear activation functions are used on the outputs of neurons. Common activations include the rectified linear unit (ReLU):

$$ReLU(x) = max(x, 0)$$

leaky rectified linear unit (LReLU):

$$LReLU(x) = max(x, 0.01x)$$

and sigmoid:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

.

ReLU is a very popular activation function used in both convolutional and fully connected deep neural networks. Its advantage is that it does not cause vanishing gradient problems in deep neural networks, as its derivative is either 1 or 0:

$$ReLU'(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases}$$

A disadvantage of ReLU is that it may lead to dying neurons. The derivative for neurons that did not contribute to the output is 0. If a neuron never contributed to the output, it stays inactive forever. Proper weight initialization and data normalization are key to preventing neurons from dying. For fully connected layers, He weight initialization is recommended. [34]

Leaky ReLU does not cause neurons to die as the derivative is never 0 and the neurons have a chance to “fix” themselves:

$$LReLU'(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0.01, & \text{otherwise} \end{cases}$$

Sigmoid squishes the output to the interval $[0, 1]$. Its disadvantage is that the derivative is always smaller than 1, causing vanishing gradients in deeper networks:

$$\sigma'(x) = \frac{-1}{(1 + e^{-x})^2} \frac{d}{dx}(e^{-x}) = \frac{-1}{(1 + e^{-x})} \frac{-e^{-x}}{(1 + e^{-x})}$$
$$= \frac{1}{(1 + e^{-x})} \frac{e^{-x}}{(1 + e^{-x})} = \sigma(x)(1 - \sigma(x))$$

Convolutional neural networks

Convolutional neural networks (CNNs) consist of convolutional layers followed by activation functions.

Convolutional layers consist of kernels that are applied to parts of the input tensor, regardless of the relative position of the part. Because kernels are applied to small neighborhoods of each position, the edges of the input are often padded with zeros so that the output has the same shape as the input. Convolutional layers also have a stride which is the distance between applied convolutions. Usually, a stride of 1 is used.

For a convolutional layer with kernel size NxM with K kernels, stride 1 and bias, the output for for an input with C channels is computed as the following:

$$y_{i,j,k}(x) = b_k + \sum_{n=1}^{N} \sum_{m=1}^{M} \sum_{c=1}^{C} x_{i-1+n,j-1+m,c} w_{i,j,c,k}$$

CNNs are used widely in computer vision. For example, a dog in a photo should be detected, no matter where in the photo it appears. As kernels are applied to parts of the input independently, the computations of CNNs can be greatly parallelized.

In CNNs, pooling layers are often used to downsample the tensor. When working with a 2-dimensional input, pooling layers split the input into rectangles of equal size and output a single number for each sub-region. The purpose of pooling layers is to expand the receptive field of the following layer and to reduce the number of parameters of the network and the number of features needed to be processed by the following layer, saving computational time.

The intuition behind pooling layers is that the exact location of features might not be that important. If there are multiple pooling layers in a CNN, it may extract local features like "is there an edge?" in the first layer and gradually start applying convolutions over these features, outputting high-level features which span a larger portion of the original image like, "is there a nose?".

Examples of pooling layers are min pool, average pool and max pool, max pool being the most commonly used. Here is the computation of a 2D max pool layer with a filter size of $2x2$ and stride 2:

$$y_{i,j,k}(x) = max_{a=0,b=0}^{1} x_{2i+a,2j+b,k}$$

For training deep CNNs, skip connections are often used. Even learning a simple identity mapping can be difficult for randomly initialized regular deep CNNs. When using skip connections, CNNs are split into blocks and the output of each block $y(x)$ is added to its input x, together creating an output $f(x) = y(x) + x$. This way, the layers have to learn just the deviation from the identity mapping. This also helps with the vanishing gradient problem as there is a short path between the output of each block and the calculation of the loss function. [35]. Deep CNNs with skip connections were used for the deep value and policy networks of AlphaGo Zero, AlphaZero, and MuZero.

CNNs were used for the original DRL agents that played Atari games [10]. The last 4 frames of the game, stacked together, were used as the input.

CNNs were also used for the Dota 2 agent and HoK agents to encode part of the game state. They applied multiple convolutional layers to spatial features extracted from the global map [6], [5] and the player hero's local view [14] [6].

Recurrent neural networks

Recurrent neural networks (RNNs) can take an ordered sequence of vectors as an input and produce a series of outputs. The input sequence can be of any length. After passing each input vector to an RNN, its hidden state changes, affecting both the current and following outputs.

RNNs are usually used to process time-series data. They have also been successfully used in natural language processing. In DRL, RNNs are used to solve non-Markovian decision processes where the history is also important or problems where the current state can be represented as a sequence.

Specific types of RNNs include gated recurrent unit (GRU) [36] and LSTM [15].

LSTM was developed to solve the vanishing gradient and exploding gradient problems of traditional RNNs. LSTM uses an input, output and a forget gate which determine what gets stored in the hidden state, what gets forgotten, and what gets outputted.

3.2 Reinforcement learning

Each instance of a reinforcement learning (RL) problem consists of an environment and an agent. The agent performs actions in the environment to which the environment responds by updating its hidden state based on the performed action and returning an observation to the agent, upon which the agent decides the next action. This loop is repeated until the episode terminates. The objective of the agent is to perform the actions in a way that maximizes the cumulative rewards acquired throughout the episode.

RL is very general and can find solutions for both fully and partially observable Markov decision processes (MDPs), thus it can easily be applied to playing games and many real-world problems.

RL framework can be used to find the optimal policy which maximizes the expected cumulative reward or to learn the value of states. When we want to find the optimal policy, we call it *control*. An example of a control task would be to determine how to play Blackjack optimally. Example usage of state values is that we might be interested in knowing the expected value in a given state of a Blackjack game, for example, once we are dealt our cards. As we are mainly interested in learning how to play games, we are going to focus on control.

More formally, the task of RL is to find the optimal policy which maximizes the expected cumulative reward. A policy π is a function that maps the list of previous observations $o_0, ..., o_n$ to a probability distribution over the action space A. After each action a_i, the environment returns a reward r_i. updates its state from s_i to s_{i+1} and produces the next observation o_{i+1}. There is also usually a discount factor $\delta \in [0, 1]$. The smaller δ is, the more the agent puts emphasis on closer rather than distant rewards.

The discounted cumulative reward R for a trajectory

$$\tau = \{s_0, o_0, a_0, r_0, s_1, o_1, a_1, r_1, ...s_n, o_n, a_n, r_n\}$$

is defined as the discounted sum of all rewards: $R_\tau = \sum_{i=0}^{n} r_i \delta^i$. The optimal policy π' is the policy that maximizes the expected discounted cumulative reward

$$\mathbf{E}[R|\pi] = \sum_{\tau} R_\tau p(\tau|\pi)$$

where the probability of each trajectory is defined as the product of the probability of the initial state and observation, the state transition and observation probabilities given the current state and action, probabilities of the rewards given the current state and action and the probabilities of performing actions given the policy and previous observations:

$$p(\tau|\pi) = p(s_0, o_0) \prod_{i=0}^{n} p(a_i|\pi, o_0, ..., o_i) p(r_i|s_i, a_i) \prod_{i=0}^{n-1} p(s_{i+1}, o_{i+1}|s_i, a_i)$$

Note that it is not necessary for RL algorithms to work to know the initial state probabilities $p(s_0, a_0)$, reward probabilities $p(r_i|s_i, a_i)$ or transition probabilities $p(s_{i+1}, o_{i+1}|s_i, a_i)$. Some RL algorithms attempt to first learn these probabilities. These are called model-based algorithms.

In practice, even though the real state may be hidden, the list of all previous observations is usually deterministically converted to a “state” to make the process Markovian. For example, the Dota 2 agent [5] used the last 16 observations as the current state, enhanced with some data from the previous observations, including the respawn times of monsters which could only be known by observing the death of those monsters in the past.

3.2.1 Fully observable Markov decision processes

In the case of fully observable MDPs, observations are equal to the states.

$$\forall i = 0, ..., n : o_i = s_i$$

Because of the Markov property which states that the state transition and reward probabilities only depend on the latest state, the policy π can then depend only on the latest state instead of all of the previous observations: $p(a_i|\pi, o_0, ..., o_i) = p(a_i|\pi, s_0, ..., s_i) = p(a_i|\pi, s_i)$.

Example of games formulated as fully observable MDPs are chess and shogi if we assume that the probabilities of the opponent’s moves are fixed, for example during self-play against a previous version of the agent, as is the case in AlphaZero.

3.2.2 Value functions

Let us now assume that we have estimated the current state s_i from the list of previous observations $o_0, ..., o_i$. There are 2 types of value functions that RL algorithms may try to learn. The value function of states (s) and the value function of state-action pairs (s, a). These functions are usually

denoted as the V and Q function respectively. These functions should return the expected discounted cumulative reward acquired.

As the expected reward sum $E[R|\pi]$ depends on a policy π, so do the value functions depend on a policy. The V^π function should return the expected value when starting in state s and following policy π. The Q^π function should return the expected value when starting in state s, executing action a, and then following policy π.

$$V^\pi(s) = \mathbf{E}[R|s_0 = s, \pi]$$

$$Q^\pi(s, a) = \mathbf{E}[R|s_0 = s, a_0 = a, \pi]$$

A common class of RL algorithms attempts to learn just the Q values of state-action pairs. The deterministic optimal policy for an estimated Q function can be computed using the following:

$$p(a_i|\pi, s_i) = \begin{cases} 1, & \text{if } a_i = argmax_{a \in A} Q^\pi(s_i, a) \\ 0, & \text{otherwise} \end{cases}$$

Another approach is to learn the state value function V together with the policy π, this is done for example in AlphaZero and MuZero.

3.2.3 Deep reinforcement learning

RL problems with small state and action spaces can be solved by storing the value function and the policy in memory as tables. For larger problems, which most real-time games are, the tables would not fit into memory. Many real-life problems are also continuous. It is possible to discretize continuous variables by assigning them to a discrete number of bins, but then we need to make a trade-off between representation quality and memory consumption.

In these cases, we usually approximate the value function and the policy instead. Let us take the V value function as an example. We choose a function V_θ with parameters $\theta \in \Theta$ where Θ is the parameter space. The task of the reinforcement learning algorithm is to then find the optimal parameters θ^* which minimize the distance between the expected value function $E[V]$ and V_θ.

One of the simplest parametrizable functions is the linear function. The value function would then be $V_\theta(s) = \theta_0 + \sum_{i=1}^{n} s_i \theta_i$ with coefficients $\theta_0, ..., \theta_n$. If the parametrizable function is a deep neural network, we call it deep reinforcement learning.

DRL recently achieved many breakthroughs. It was behind most superhuman AI players. An example real-life problem that was successfully solved using DRL is chip floorplanning. Mirhoseini et al. [37] formulated the task of planning component placement as a sequential MDP. The process consisted of placing circuit components on a chip canvas, one at a time. The goal was to minimize the wire length, congestion, and density of the final placement. Using a deep neural network as a joint parametrizable policy and value function, the authors were successfully able to match or outperform manual

placements created by human experts. On top of that, the manual placements their method was compared to took human experts months to develop in an iterative process. Their method was used to design the next generation of Google's AI accelerators.

Chapter 4

Dataset

Initially, we thought we would need to create artificial data but we were able to get access to real data and decided it was not meaningful anymore.

The experiments we conducted were on matches from professional tournaments. The dataset is not public.

To avoid variance caused by training our models on different versions of the game, all of the matches we have chosen were from patch 11.15 as it was the version with the most available games.

4.1 Data structure

The raw data of each game consists of a series of events. The most important event is the “state update event” which describes the current state of all champions including their items and global game statistics like total team gold, team kills, deaths, dragons killed and barons killed. Other events specify a single game event, like the purchase of an item, the death of a champion, or the destruction of a building.

The game states consist of structured multi-modal data. We have scalar global data, which includes statistics, states of buildings, and epic monsters. Then we have sets of champions for both teams. Each champion owns a set of items and has sets of active sight, control, and farsight wards placed on the map and a set of summoner spells. From the raw data, we also created spatial data, which we obtained by splitting the square global map into a grid of NxN sectors. We set $N = 16$ and define 6 features for each team, resulting in a tensor of size 16x16x12. The channels specify how many champions, turrets, inhibitors, and wards of each type are present in each sector of the map for each team.

Information about the current state of minions and neutral monsters was missing in the data, as well as basic ability cooldowns of champions and the atomic actions they took, including when they issue a move command, perform an attack, or cast an ability. The fact that a champion cast its ultimate ability or a summoner spell can be deduced from the fact that it goes on cooldown, which is available for these abilities, but precise time and usage details, including the target of the ability, are missing.

All available data can be seen in Table 4.1

4.2 Data wrangling

In total, we were able to retrieve 664 games played on version 11.15. Of the 664 available games, 4 contained no events and 1 contained multiple events that shared the same sequence index, making it unclear how to process them. We left these games out, leaving 659 games to analyze.

First, we checked the distribution of values for each feature. For visualization of graphs, we use the matplotlib python library [38]. Visualizations of the distribution of positions of champions, positions where they die and positions where wards are placed during different parts of the game can be seen in Figure 4.1.

Events like "dragon kill" specified the indices of assistants participating in the kill. They would sometimes contain certain indices multiple times. This was likely caused by an error and we chose to ignore the duplicate values.

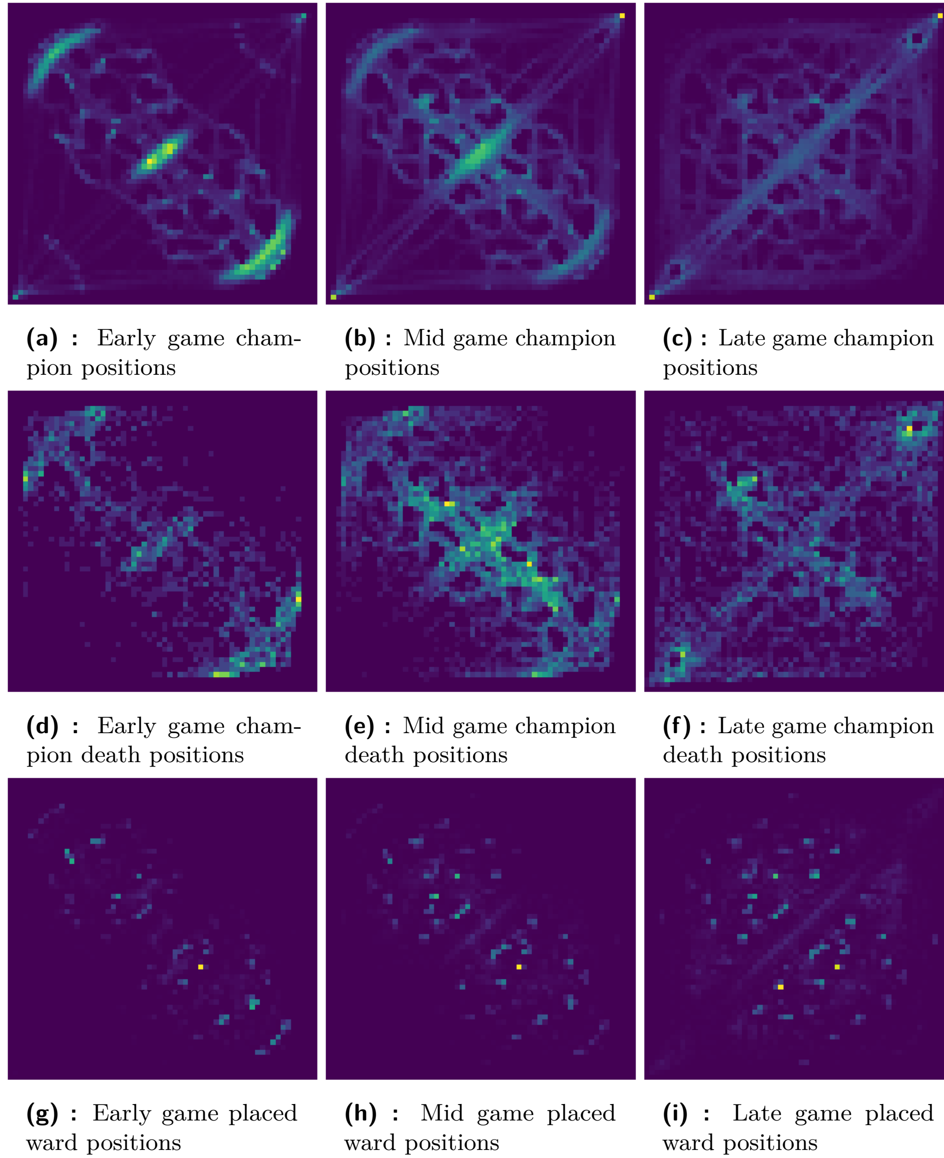
We aggregated events of each game into a series of game states with a constant time difference of 2.5 seconds, 5 seconds, and 10 seconds between subsequent states.

We have access to team statistics and know when epic monsters got killed and buildings got destroyed. From the events and respawn times of monsters [39], we calculated which monsters are currently alive and tracked Baron and dragon buffs for both teams and the blue and red buffs for each champion. For each champion, we have information about their state, statistics, items, levels of skills, currently active wards, chosen summoner spells, and their cooldowns.

For the baron buffs, elder dragon buffs, champion buffs, and the ward features, we had to manually keep track of the current state because nor the remaining time of buffs, nor the remaining time of wards was present in the data. The fact that wards disappeared due to exceeding the limit on the number of active wards of that type for that player was also not recorded in the data. We had to implement the game logic in order to determine which wards are currently active.

state	**champion_state**	**champion_stats**
minion respawn	**per champion:**	**per champion:**
scuttle crab respawn	alive	minions killed
dragon respawn	respawn timer	neutral minions killed
Rift Herald respawn	level	neutral minions killed your jungle
Baron respawn	xp	neutral minions killed enemy jungle
per team:	position x	champions killed
3x inhibitor respawns	position y	num deaths
raptor respawn	health	assists
wolf respawn	max health	wards placed
gromp respawn	health regen	wards killed
krug respawn	magic penetration	vision score
red camp respawn	magic penetration percent	total damage dealt
blue camp respawn	magic penetration percent bonus	physical damage dealt player
baron buff remaining	armor penetration	magic damage dealt player
dragon buff remaining	armor penetration percent	true damage dealt player
air dragon buff level	armor penetration percent bonus	total damage dealt to champions
earth dragon buff level	current gold	physical damage dealt to champions
water dragon buff level	total gold	magic damage dealt to champions
fire dragon buff level	gold per second	true damage dealt to champions
skills	shutdown value	total damage taken
4x per champion:	primary ability resource	physical damage taken
skill embedding	primary ability resource max	magic damage taken
level	primary ability resource regen	true damage taken
is ultimate	attack damage	total damage self mitigated
evolved	attack speed	total damage shielded on teammates
wards	ability power	total damage dealt to buildings
control wards	cooldown reduction	total damage dealt to turrets
2x per champion:	lifesteal	total damage dealt to objectives
position x	spell vamp	total time crowd control dealt
position y	armor	total heal on teammates
sight wards	magic resist	time ccing others
4x per champion:	cc reduction	**stats**
position x	ultimate cooldown	**per team:**
position y	red buff remaining	tower kills
remaining time	blue buff remaining	assists
farsight wards	team	inhib kills
20x per champion:	is ally	total gold
position x	**champion_embedding**	champion kills
position y	**per champion:**	deaths
map (16x16 tiles)	champion embedding	dragon kills
per team:	**gameTime**	baron kills
champions	game time	**items**
turrets	**summoner_spells**	**7x per champion:**
inhibitors	**2x per champion:**	item embedding
sight wards	summoner spell embedding	cooldown remaining
control wards	cooldown remaining	
farsight wards		

Table 4.1: List of all information available at each game state. Features with a blue background are learned categorical embeddings of size 2. The rest are numerical features. There is one feature that is different depending on which champion's behavior we are predicting and that is the "is ally" feature in **champion_state**



(a) : Early game champion positions

(b) : Mid game champion positions

(c) : Late game champion positions

(d) : Early game champion death positions

(e) : Mid game champion death positions

(f) : Late game champion death positions

(g) : Early game placed ward positions

(h) : Mid game placed ward positions

(i) : Late game placed ward positions

Figure 4.1: 2D histograms of champion positions in the data. We defined mid-game to start when either a turret gets destroyed or the dragon is killed and late game when an inhibitor is destroyed or the baron is killed.

Chapter 5

Solution

5.1 Predicting game outcome

We developed an interpretable in-game win probability model based on simple features like the total kills of each team and the objectives accomplished by each team. After finishing a game, the graph of the win rate throughout the game can be shown to players with important in-game events highlighted. From this, players can gain more insight into how much each game event contributed to the final result of the game.

To model the probability, we tried logistic regression, small multi-layer dense neural networks, random forests and gradient boosted decision trees.

As the baseline, we chose the in-game probability model developed by Maymin et al. [29]. They used logistic regression on the game time in minutes, the number of kills of both teams, the number of turrets destroyed by both teams, and the number of epic monsters slain by both teams.

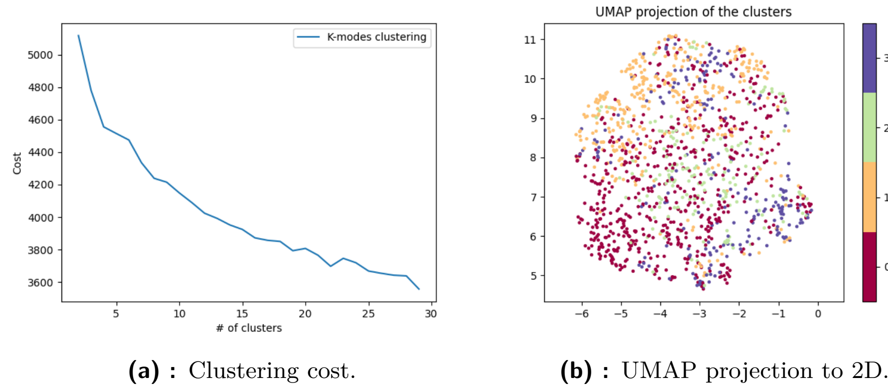
5.1.1 Feature engineering

Because our dataset was so small, we had to worry about overfitting. We started with the features used in the baseline and developed our own features with different levels of specificity.

We tried adding important statistics, namely team gold for both teams and champion levels. We also tried being more specific with epic monster kills, splitting them into dragons and barons killed. To be even more specific, we tried splitting dragon kills further into the specific types of dragons killed as they all grant different buffs. We added a feature that states how much time is remaining for the baron and elder dragon buffs to end for both of the teams. We also tried different levels of aggregation for turrets. Finally, we added information about the number of inhibitors destroyed and the number of currently active wards.

To make the input space represent the current game state, we also suggest specifying the current number of champions alive per team. To get more specific, we also tried adding the remaining respawn times of champions as a feature. We also tried adding the remaining respawn times of inhibitors.

The champion pick phase greatly influences the outcome of games. Due to



(a) : Clustering cost.

(b) : UMAP projection to 2D.

Figure 5.1: K-modes clustering of team champion compositions. (**a**), Shows the cost of clustering depending on the number of clusters using k-modes. (**b**), The UMAP projection of n-hot encoded teams of champions to 2 dimensions, colored by the 4 clusters as discovered by k-modes.

the specific ways the abilities of different champions interact, some champions are good against others. We then say that a champion "counters" the champions it is good against. Some champions and team compositions win games more often than others. In esports, competitive teams research what champions and team compositions other teams like to play and draft and ban champions during the pick phase accordingly.

There are websites that collect match data and report individual win rates for each champion. We propose a champion feature, defined as the difference between the 5-hot encoded vectors of champions of both teams.

We try to identify team compositions and model their win rates by clustering champion teams using the k-modes [40] algorithm. We used the implementation from the k-modes python library [41]. We used the "Huang" initialization with 5 random starts. Using the elbow method on the costs of clusters of teams from the entire dataset, we chose $k = 4$ clusters. The plot of the cost together with an unsupervised 2D projection of the n-hot encoded teams using UMAP [42] reduction can be seen in Figure 5.1. During the 5-fold cross-validation, the clusters were determined using the training part of each split.

We implemented a slightly modified version of the pseudocode for automatic categorization of champion roles proposed by Maymin et al. [29]. Because we did not have access to minion positions and deaths, instead of assigning the mid role to the player who killed the most minions near the map's center, we assigned the mid role to the champion who has spent the most time near the center of the map throughout the first 10 minutes of the game.

The role categories were used to order per-champion features, namely level and the remaining respawn time. They were also used to order the champions in teams to 5-dimensional categorical vectors for the k-modes clustering.

Overall, we created the following features:

- time - In minutes.

- gold - Total gold per team in thousands.
- kills - Total number of kills per team
- level - Level of each champion.
- level mean - The mean champion level per team.
- respawn - Remaining time to respawn for each champion in seconds, 0 for champions that are alive.
- alive - Number of team members alive for each team.
- champions n-hot - A vector of dimension equal to the number of unique champions, where each value indicates whether the champion is present in either of the teams. The value 1 indicates that the champion is present in the blue team, -1 indicates that the champion is present in the red team.
- champion clusters - Represented as the difference between one-hot encoded vectors of team cluster of both teams.
- dragons - How many times did each team kill each type of dragon.
- dragons total - How many dragons did each team kill.
- barons - How many barons did each team kill.
- epic buffs - How much time remains for the global buffs awarded for slaying Baron Nashor and the elder dragon for both teams, in seconds.
- monsters - How many epic monsters (dragons or Baron Nashor) did each team kill.
- turrets - Boolean indicator for each turret for each team, indicating whether the turret was destroyed.
- turrets per lane - How many turrets were destroyed on each lane for each team.
- turrets per tier - How many turrets of each tier (outer, inner, base, nexus) were destroyed for each team.
- turrets total - How many turrets have, in total, been destroyed by each team.
- inhibitors total - How many inhibitors have, in total, been destroyed by each team.
- inhibitors per lane - How many inhibitors on each lane have been destroyed by each team.
- inhibitors respawn - How much time remains for the inhibitor on each lane to respawn for both teams, in seconds.

- wards total - How many wards are active in total per team.
- wards per type - How many sight wards, control wards, and farsight wards are currently active per team.

5.1.2 Results

We tested our models with different features using 5-fold cross-validation. First, we split the games into 5 subsets of roughly equal size. We chose one of these subsets as a testing set, trained our model on the other 4 subsets, and evaluated our model on the test subset which was not part of the training set. We repeat this 5 times, each time choosing a different test subset from the 5 subsets and average the results over these 5 runs.

K-fold cross-validation allows us to see how the model generalizes to unseen data and is usually used when the dataset available is small. It gives a better estimate than just splitting the dataset once because, in k-fold cross-validation, we get an average result over k splits instead of just one.

Unlike Maymin et al. [29], we did not have the luxury of having a large amount of data. They only used 1 random sample from each game in the training split which resulted in independent rows for training. Because our dataset was small, we found that increasing the number of samples used per game increased the performance of our model. The relationship can be seen in plot Figure 5.2. In the following experiments, we used all of our data in the train split for training.

Both gradient-boosted trees and random forests had a very high training accuracy but they did not generalize well to unseen data. Multi-layer dense networks had similar results as logistic regression but took longer to train. Logistic regression performed the best in the cross-validation so we report only results for logistic regression with different combinations of features. The results can be seen in Table 5.1.

For our experiments, we did not use the time feature as, in our opinion, it does not make sense to assume that the logit of the probability that either of the team wins would depend linearly on the number of minutes passed. Other than that, in our base model, we used the same features as in baseline, namely kills, turrets total, and monsters.

We started with the base features and tried all the possibilities of either removing one of the base features, adding one additional feature, or making one of the features more informative by replacing it with a more specific one.

By removing one feature from the base model at a time, we did an ablation study of the base model. We can see that kills is the most influential feature, while the monsters feature is the least influential feature.

From this initial experiment, we found that adding either gold, level mean or epic buffs led to an improvement. Replacing the monsters feature with barons and dragons total, barons and dragons or epic buffs, barons and dragons total also increased the test accuracy.

Features that performed poorly include level, inhibitor per lane, inhibitor total, more specific turret features, champion clusters, and champions n-hot.

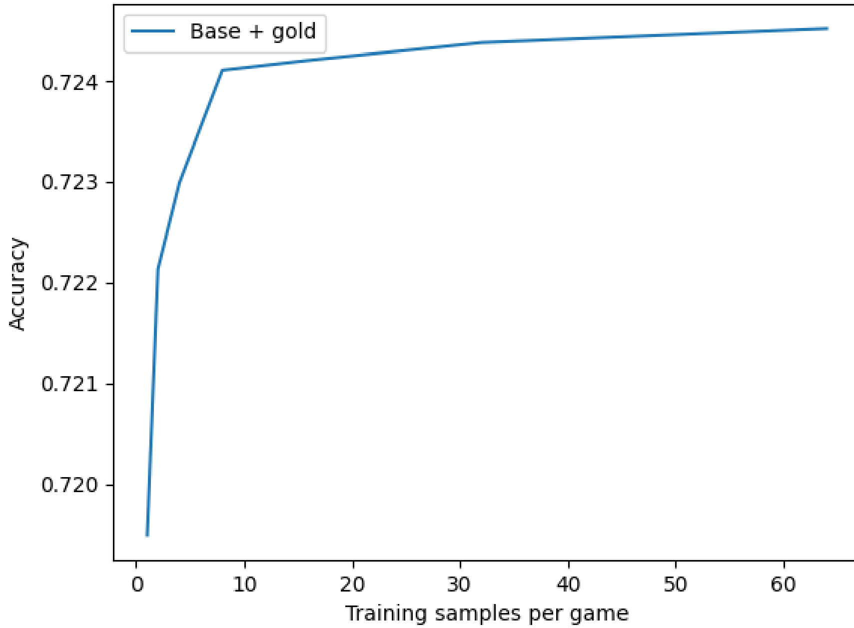


Figure 5.2: The relationship between the number of samples per game in the training split and the final test accuracy. Measured using 5-fold cross-validation, averaged over 10 runs. The Base + gold model was used.

Features that did not influence the result much were respawn, alive, inhibitors respawn, wards total, and wards per type. Adding them to the base model changed the test accuracy by less than 0.1%.

We also tried removing the kills feature as it represents the history of the game, rather than the current state. As we already mentioned, removing kills from the base features hurt the performance a lot (Base - kills). We successfully remedied this by adding both the gold and level mean features (Base - kills + gold, level mean). It seems logical as all the advantages gained by killing opponents, namely gold and experience advantage and being able to destroy turrets and kill epic monsters, are already included in the other features. The only thing missing is then the history of how successful were members of both teams at killing champions from the other team. We also tried adding the alive feature but that did not change the result on the test dataset.

We then tried combining the features that succeeded in improving the base model. Gold was the most prominent feature that we added. This is again confirmed by the Base + level mean + epic buffs model, which combines 2 useful features but the result is still worse, both in train and test accuracy, than the one achieved by just adding gold.

As an extreme, we tried including as much information in the inputs as possible. We included each feature at the most specific level, without the champion features. This gave rise to the All features model, which used the

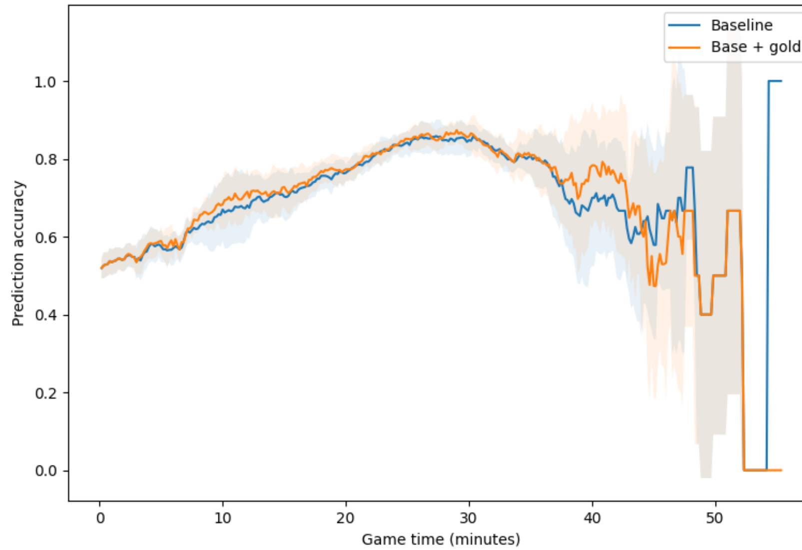


Figure 5.3: Prediction accuracy of win prediction models, depending on game time. The mean and standard deviation were calculated from the 5 splits. At each time t, the mean μ_t and standard deviation δ_t across the 5 splits is calculated. The interval $[\mu_t - \delta_t, \mu_t + \delta_t]$ is highlighted for each time t for both models. On the right, the standard deviation starts increasing because the number of games that lasted that long is decreasing.

following features: kills, turrets, barons, dragons, epic buffs, gold, respawn, level, inhibitors per lane, inhibitors respawn, wards per type. As expected, it achieves the second-best train accuracy behind Base + champions n-hot. It also surpassed the base model in test accuracy.

For logistic regression, the best working features were “Base + gold”, “Base + gold, epic buffs”, “Base - monsters + gold, epic buffs, dragons total” and “Base - monsters + gold, barons, dragons total”. Because all of the models achieve similar accuracy on the test set, we can use either model. If we had access to more data, which of these features are the best could become more clear.

We chose to use the “Base + gold” model in the rest of this Thesis because it achieved one of the best test accuracies and the very small improvements of the other more complex models did not seem to justify the added complexity.

The accuracy of Baseline and our proposed “Base + gold” over time can be seen in Figure 5.3. Both models can most accurately predict the winner at about the 26-minute to 31-minute interval. This makes sense as at this time, one team has probably been able to gain an advantage over the other. Earlier, the game is still undecided, and later in the game, both teams are eventually able to buy the best items and the situation becomes more balanced, as both teams become similarly strong and the winner of a single team fight is likely going to be able to win the game.

Examples of how our “Base + gold” model trained on one of the cross-validation splits works on unseen games can be seen in Figure 5.4.

5.2 Predicting macro decisions

5.2.1 State encoding

Previous works have successfully used neural networks to encode the current state of the game [5] [13].

To use neural networks, we vectorized the multi-modal game state data, converting it into a single vector per time step. We then used a neural network, either an LSTM network on the last N game states or a fully connected network on the last game state to encode the game history into a single vector.

We split the games in our dataset into train, test, and valid splits in a ratio of 80 to 10 to 10. We used a similar approach to encoding the game state as the one used by Berner et al. [5] for Dota 2. All numerical values, including booleans and floats, are normalized so that they have a mean of 0 and a standard deviation of 1. To do this, we obtain the mean and standard deviations of each feature from the training dataset and then modify each feature i in our datasets according to the following equation:

$$feature_i = \frac{feature_i - \mu_i^{train}}{std_i^{train}}$$

The way we process each type of data to obtain game state vectors can be seen in Figure 5.5.

In case a feature takes only a single value, which is the case for some features in the champion state, namely magic penetration percent bonus, armor penetration, armor penetration percent bonus, and cooldown reduction, we set the standard deviation for that feature to 1 to prevent division by zero, therefore setting the feature in all samples in the dataset to 0.

For categorical data, namely the different types of champions, items, summoner spells, and champion skills, we learn embedding vectors of fixed size during training.

To encode the spatial map features, we use a standard 2-layer convolutional neural network.

Game states contain data of varying sizes. Each champion can have a different number of currently placed wards and a different number of items in its inventory. To enable batch processing, we zero pad these sets to the maximum theoretical value allowed by the game rules. The limits were 4 for sight wards, 2 for control wards, 20 for farsight wards, and 7 for items. There is no hard boundary on the number of farsight wards a player could place but we chose 20 as that is a value that should hardly ever be reached in real games. The other limits for wards are only reachable by owning a specific game item, otherwise, a player can only have 3 sight wards and 1 control ward placed at a time.

We treat items, wards of all types, summoner spells, skills, and champions as sets. For each set, we apply a set processing procedure. We apply a two-layer fully connected neural network with ReLU non-linearity to transform each item in the set independently, including the zero-padded items. Both layers have the same number of neurons. Then, we perform 1-dimensional max-pooling over the set, converting the set of items into a single vector. This processing is independent of the order of the items in each set.

By projecting champions through the two fully connected layers, we obtain another, more complex champion embedding, which includes information from items, summoner spells, skills, wards, champion state, champion stats, and champion type embeddings, if all of these are provided.

For each champion c in a game x at time t, the state $x_{t,c}$ is a little bit different. We append the processed champion embedding to the game state before it is encoded so that the network knows for which champion to predict macro decisions. The *champion state* feature is also slightly different, as the *is ally* feature also depends on the current champion c.

5.2.2 Prediction targets

We attached additional heads to the encoder. Their purpose is to predict various information about the game from the game state. That way, we forced the encoder to encode multiple useful pieces of information from the raw game state data into a single vector.

Movement intention

To capture the decision of where the player wants to go on the map, we split the square map into course-grained NxN regions, similar to the supervised learning auxiliary macro target used by Ye et al. [6]. We decided for $N = 12$, resulting in $N^2 = 144$ sectors.

For each champion c and each time step t (except for the last time step) in a game x, we define the prediciton target as $Y_{t,c}^{pos}(x) = i$ where i is the index of the sector where the champion c is present in the next game state x_{t+1}.

One head is going to predict the probability of each sector being the sector where the champion c is present in the next time step $f(x_{t,c})^{pos}$ from the encoded game state $x_{t,c}$.

Minion and monster farming

One of the most important things to do in the early game is to slay enemy minions. Being able to last-hit enemy minions or prevent your opponents from last-hitting yours is a great way to get an advantage. On the other hand, respawn times are short in the early game which means killing champions at this stage has a lower impact on the game than killing them later. Leaving your lane in an attempt to kill an opponent in another part of the map might not be worth it even if you do manage to secure the kill because of the minions you miss on your lane. This is the reason why it is mostly only the jungler

who ganks other lanes in the early game, once they kill all of the neutral monsters in their jungle.

However, later in the game, it could be more important to participate in a team fight or in capturing an objective than to last-hit minions on your lane.

Minions and neutral monsters are together called *creeps*. The total number of minions and monsters slain is together called *creeps slain* (cs). This is an important statistic that most tools track to report performance, as it is important to slay as many creeps as possible throughout the game.

We had two of the heads predict the discounted number of minions and monsters slain. For each champion c and each time step t (except for the last time step) in a game x with N timesteps, we define the prediction target with time horizon T and discount factor $\delta \in [0, 1]$ as

$$Y_{t,c}^{minions}(x) = \sum_{i=t+1}^{min(t+T,N)} \delta^{i-(t+1)} minions_{i,c}(x)$$

$$Y_{t,c}^{monsters}(x) = \sum_{i=t+1}^{min(t+T,N)} \delta^{i-(t+1)} monsters_{i,c}(x)$$

where $minions_{i,c}(x)$ and $monsters_{i,c}(x)$ are the numbers of minions and monsters slain by champion c between time steps $i-1$ and i in game x, respectively.

Fights and map objectives

Another important part of macro is to decide whether to participate in a fight and whether to try to capture a map objective. We are going to model this by predicting the discounted number of champion kills, destroyed turrets, destroyed inhibitors, slain Rift Heralds, dragons, and Baron Nashors to which the champion contributed.

For each event type e of the above-mentioned events, we define the prediction target for each champion c and each time step t (except for the last time step) with timespan T and discount factor $\delta \in [0, 1]$ in a game x with N timesteps as the discounted number of times that the event e happened for champion c between time steps t and $t + T$:

$$Y_{t,c}^{e}(x) = \sum_{i=t+1}^{min(t+T,N)} \delta^{i-(t+1)} e_{i,c}(x)$$

where $e_{i,c}$ is the number of times the event e happened for champion c between time steps $i - 1$ and i.

We say that an event e happened for a champion if they participated in it. That is, we do not distinguish between kills and assists for any of the events. For example, if both teams tried to slay Baron Nashor, we are going to record this event for all champions that assisted in killing him, no matter which team ended up last hitting him. We do this because we try to model the intentions of players rather than the final outcomes of their decisions,

which have a higher variance and depend more on the micromanagement of players rather than their macro decisions.

Multitask loss

Prediction heads can be trained in parallel and jointly with the encoder. Our model is going to optimize the network parameters θ to minimize the following weighted sum of losses on our training dataset X:

$$L(\theta) = E[w_{pos} l_{CE}(f(x_{t,c}|\theta)^{pos}, Y_{t,c}^{pos}(x)) + \sum_{event_e} w_e l_{MSE}(f(x_{t,c}|\theta)^e, Y_{t,c}^e(x))]$$

where $f(x_{t,c}|\theta)$ is the network output for game state $x_{t,c}$ given parameters θ. w_{pos} and w_e are the loss weights and the possible events e are minions, monsters, kills, turrets, inhibitors, Rift Heralds, dragons and Barons.

Game timesteps x_t have a uniform distribution over the training dataset X. Champions c have a uniform distribution over the set of possible indices of champions in each game $\{1, ..., 10\}$.

The need to weigh the losses comes from the fact that each of the prediction targets has a different variance. While a champion can slay 10 minions during a single timestep, they can only participate in killing the baron a few times throughout the game. Most commonly, the amount of barons a champion slays during a timestep is 0.

Before weighing the targets, we found that the position, minion, and monster events were the largest contributors to the total loss, while rare events like Baron, Rift Herald, dragon, and inhibitor barely contributed.

We set the weight of the position loss w_{pos} to 1 empirically and the weight of each event e as the inverse variance of the prediction target.

$$w_e = \frac{1}{Var(Y_{t,c}^e)}$$

This way, we convert the losses to the same scale on which a very simple predictor that would predict the target to be the estimated mean from the training dataset

$$\hat{E}^e = E_{x_t,c}[Y_{t,c}^e] = \frac{1}{10|X|} \sum_{x_t \in X} \sum_{c=1}^{10} Y_{t,c}^e(x)$$

would obtain an average weighted loss of 1 on the training dataset:

$$E[w_e l_{MSE}(\hat{E}^e, Y_{t,c}^e)] = \frac{1}{Var(Y_{t,c}^e)} \frac{1}{10|X|} \sum_{x_t \in X} \sum_{c=1}^{10} (\hat{E}^e - Y_{t,c}^e(x))^2$$

$$= \frac{1}{Var(Y_{t,c}^e)} \frac{1}{10|X|} \sum_{x_t \in X} \sum_{c=1}^{10} (E_{x_t,c}[Y_{t,c}^e] - Y_{t,c}^e(x))^2 = \frac{1}{Var(Y_{t,c}^e)} Var(Y_{t,c}^e) = 1$$

Summary

To summarize, there will be a head for predicting the answer to each of the following questions:

- Where in the map is the player going to be in the next timestep?
- How many enemy minions is the player going to slay?
- How many neutral monsters is the player going to slay?
- How many times is the player going to participate in killing an enemy champion?
- How many times is the player going to participate in destroying an enemy turret?
- How many times is the player going to participate in destroying an enemy inhibitor?
- How many times is the player going to participate in slaying the Rift Herald?
- How many times is the player going to participate in slaying the dragon?
- How many times is the player going to participate in slaying Baron Nashor?

By training the model to answer all of these questions on games played by good players, we should get an interpretable probability model of player behavior. By applying this model to games of lower-ranked players, we hope to automatically detect game situations in which the difference between the predicted and actual behavior is high, meaning that the player did not pick an action that a good player would.

Note that this framework is very general and can easily be extended with additional heads and tasks.

5.2.3 Model

We implemented our model using PyTorch [43]. For many intermediate calculations, we used NumPy [44].

Hyperparameters

We set the discount factor $\delta = 0.7$ and time horizon $T = 6$ which covers 60 seconds considering our timestep length of 10 seconds. We trained our network with batch size of 64. Weights of each regression target, together with their statistics can be seen in Table 5.2.

We trained our models using the Adam optimization method [45] with a learning rate of $1e-3$.

Throughout our experiments, we were mostly dealing with overfitting. We started with a larger neural network. First, we made sure that the network was able to learn perfectly a small subset of the training data. Then, we iteratively reduced the number of weights in parts of the network. We did ablation studies of parts of the network and reduced the number of parameters in parts without which the model generalized better to unseen data. By doing this, we went from an initial number of more than 1 million trainable parameters to 40 340 trainable network parameters. When comparing our model to the model of the Dota 2 agent with approximately 159 million parameters, their model had more than 3 900 times more parameters. Complete training of our network took less than an hour on a single NVIDIA Geforce GTX 1060 with Max-Q Design GPU.

We found that increasing the number of recent states available to the model did not improve it but rather caused overfitting, so we used a history size of 1 for our predictions. This may have been caused by the low amount of data we have and the fact that our timesteps are rather large.

We proposed multi-task learning because we were interested in multiple predictions for each champion in each time step but also as a regularization technique. This however caused a problem during training as we had to compromise on the number of training epochs. The ideal number of training epochs differed for each objective. In our experiments, the position prediction loss on the validation dataset kept decreasing for at least the first 6 epochs, while the loss for the turrets target on the validation dataset started increasing after the first 2 epochs. We chose the best model for evaluation by looking at the weighted loss sum $L(\theta)$.

We ended up not using the stats and champion stats features in our model because they were causing overfitting in earlier experiments. While reducing the sizes of embeddings to stop overfitting, we stopped at 2 for all embeddings.

The first convolutional layer that we use for spatial data processing has 8 kernels and the second one has only 1 to reduce the number of features. We used kernels of size 3 with zero padding for both convolutional layers and each one is followed by a 2-dimensional max pool layer and LReLU activation function.

We used just 2 neurons in the fully connected layers of item, ward, summoner spell, and skill processors, 32 neurons in fully connected layers of champion processor, and 128 neurons to encode the game state.

We tried adding weight decay of $1e-4$ to the model as a regularization technique but that decreased the performance of the model on the validation dataset.

Our prediction heads are each just a single linear layer. As such, we combined them into a single linear layer so that their computation can be parallelized.

5.2.4 Results

We trained all models for 5 epochs. After each epoch, we calculated the total loss on the validation dataset. We then used the weights of the model that

achieved the lowest total loss on the validation dataset and evaluated it on the test dataset.

We show the influence of timestep length by comparing the results of models trained with timestep lengths of 10 seconds, 5 seconds, and 2.5 seconds in Table 5.3. For these models, the same discount factor of 0.7 and a time horizon of 6 time steps were chosen. This means that for the regression tasks, the models with interval lengths of 10 seconds, 5 seconds, and 2.5 seconds predicted the discounted sums of events happening in the next 60 seconds, 30 seconds, and 15 seconds respectively.

As expected, the models tasked to predict information about a closer future achieved smaller losses on many of the prediction tasks. The largest improvement was observed in predicting champion positions. This is likely also caused by the fact that there are fewer options as champions are not likely to travel far from their current positions in shorter time intervals. Minions and monsters were an exception, the model with an interval length of 10 seconds achieved the lowest losses on these tasks. Our theory is that because we are missing minion and monster states in our data, the shorter interval models struggled to pinpoint the exact moments when champions would be able to kill them and the longer prediction time horizon of the model helped it.

The models with interval lengths of 10 seconds, 5 seconds, and 2.5 seconds achieved position prediction accuracies of 27.81%, 39.72%, and 54.61%, respectively.

In the rest of this work, we worked with an interval length of 10 seconds. All visualizations are done with the model from this first experiment.

During training, our model learned embeddings for different types of entities in LoL. The visualizations of the learned champion, item, skill, and summoner spell embeddings were too large to show in this Thesis, they can be found in the attached folder.

From regression targets, our model was able to achieve the smallest loss on minions and monsters. The more rare events, namely kills, turrets, inhibitors, Rift Heralds, dragons, and Barons were harder for the model to predict. Kills were the hardest for our model to predict.

Distributions of the deviations of our predicted target from the real targets on the test dataset can be seen in Figure 5.6.

A histogram of predicted probabilities of actual movements of champions from the test dataset can be seen in Figure 5.7.

To understand the influence of each feature and prediction target, we did ablation studies of features and targets. The results of our model and the results when adding or removing a feature or a target can be seen in Table 5.4 and Table 5.5.

The feature ablation study demonstrates that it holds for many of our features that removing them increases the performance of the model on unseen data. On the other hand, removing “champ_state” greatly increases position loss because the current position of the champion is present in this feature. On this run, adding “champ_stats” or “stats” features slightly improved

the model, even though they caused overfitting in earlier experiments when the network was slightly larger. Removing the wards feature improved the performance the most.

The fact that simplifying the model by removing important features like wards and items reduces the loss of the model on unseen data suggests that we are having problems with overfitting and the model would likely greatly benefit from training on a larger dataset.

In the ablation study of targets, we show that even though the base model is not the best in any of the individual losses, it is able to achieve a lower total loss on 3 out of 9 loss subsets than models trained only to predict the targets present in each subset.

Note that the base models in both ablation studies have different losses even though they have the same hyperparameters and were trained and tested on the same datasets. This is because these were results from 2 different runs.

We have chosen a single game from the test set randomly. In this game, we show visualizations of position predictions for a specific champion and the predicted movements for all champions in different game states in Figure 5.8 and Figure 5.9.

In this game, we also show deviations from the expected behavior. We found these moments automatically by finding the largest deviation from the predicted targets among all champions and game time steps in the game. We would call these moments "bad decisions" if they were found in a game of regular players and not performed by professional players. They can be seen in Figure 5.10.

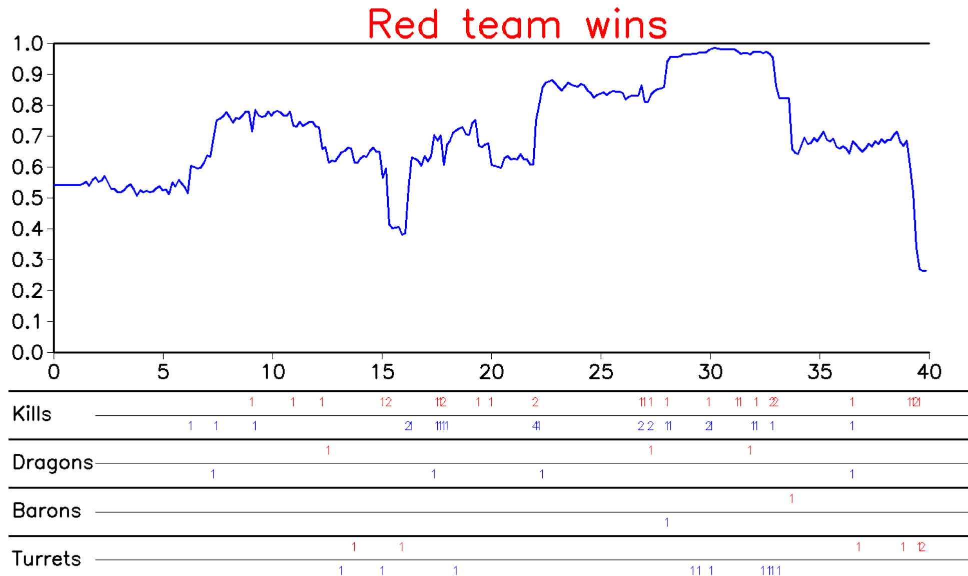
By visualizing the model predictions and disadvantageous decisions, we have empirically evaluated that it was able to understand some of the key concepts of LoL strategy. The model was able to successfully predict the positions of champions at the start of the game. It also has a general idea of when it might be possible to capture objectives like epic monsters, inhibitors, and turrets.

Features	Train accuracy	Test accuracy
Baseline	71.31%	71.27%
Base	71.31%	71.27%
Base - kills	66.66%	66.42%
Base - turrets total	70.26%	70.10%
Base - monsters	71.01%	70.88%
Base + gold	72.59%	72.31%
Base + level	71.95%	71.14%
Base + level mean	71.76%	71.45%
Base + respawn	71.49%	71.34%
Base + alive	71.46%	71.28%
Base + champions n-hot	81.80%	66.10%
Base + champion clusters	71.43%	70.70%
Base + epic buffs	71.61%	71.44%
Base + inhibitors per lane	71.42%	71.16%
Base + inhibitors total	71.35%	71.12%
Base + inhibitors respawn	71.30%	71.34%
Base + wards total	71.32%	71.19%
Base + wards per type	71.38%	71.28%
Base - monsters + barons, dragons total	71.63%	71.59%
Base - monsters + barons, dragons	71.95%	71.50%
Base - monsters + epic buffs, barons, dragons total	71.72%	71.59%
Base - turrets total + turrets	71.51%	70.65%
Base - turrets total + turrets per lane	71.42%	71.21%
Base - turrets total + turrets per tier	71.33%	71.01%
Base - kills + gold, level mean	72.44%	72.00%
Base - kills + gold, level mean, alive	72.46%	72.00%
Base - kills + gold, level mean, epic buffs	72.47%	71.98%
Base + gold, level mean	72.52%	72.07%
Base + gold, epic buffs	72.63%	**72.33%**
Base + level mean, epic buffs	71.89%	71.61%
Base - monsters + gold, epic buffs, dragons total	72.64%	72.29%
Base - monsters + gold, barons, dragons total	72.62%	72.32%
All features	72.89%	71.51%

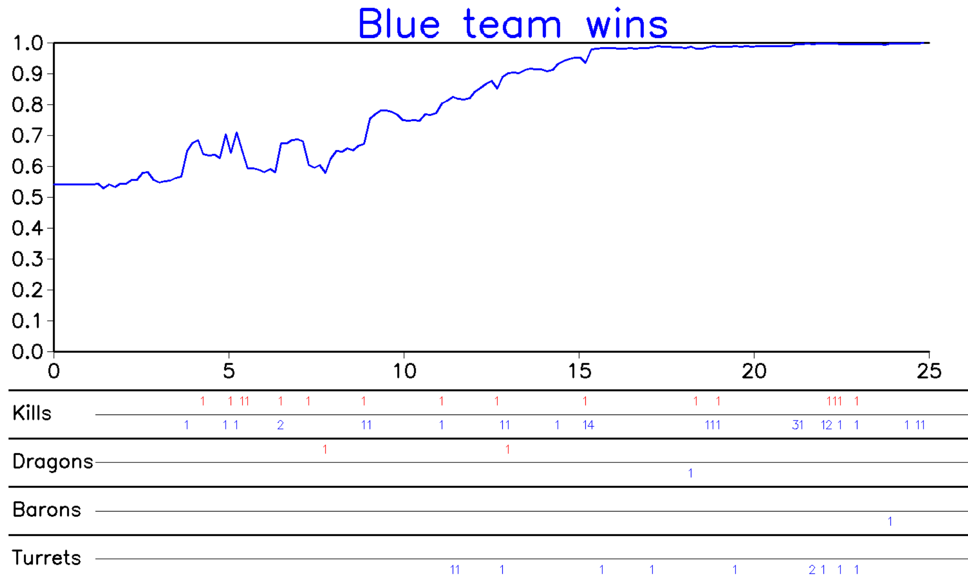
Table 5.1: Train and test accuracies of logistic regression obtained from 5-fold cross-validation averaged over 10 runs.

	Minion	Monster	Kill	Turret	Inhib	Herald	Dragon	Baron
Mean	2.4697	0.6278	0.1439	0.0358	0.0082	0.0073	0.0189	0.0092
Var	7.2554	2.2297	0.1686	0.0325	0.0059	0.0047	0.0121	0.006
Weight	0.1378	0.4485	5.9312	30.769	169.49	212.77	82.645	166.67

Table 5.2: Regression target means, variances and weights.



(a) : The game starts evenly with both teams being able to secure some kills, turrets, and dragons. At around 22 minutes, the blue team is able to kill all members of the red team and their third dragon, granting them a large advantage. At 27 minutes, the blue team is able to slay the baron and destroy a lot of turrets. Then, though they are at a disadvantage, the red team is able to kill all members of the blue team at 33 minutes and 39 minutes and win the game.



(b) : This is a game in which the blue team gradually gains advantages and ends up winning the game. The first two dragons end up being taken by the red team but the blue team is able to get more kills and destroy more turrets. Then, the blue team is also able to kill a dragon and the baron. The blue team ends up winning this game without losing a single turret.

Figure 5.4: Probabilities of the blue team winning throughout sample games from the test set. At the bottom, important events and the number of times they happened at each state, if non-zero, are shown.

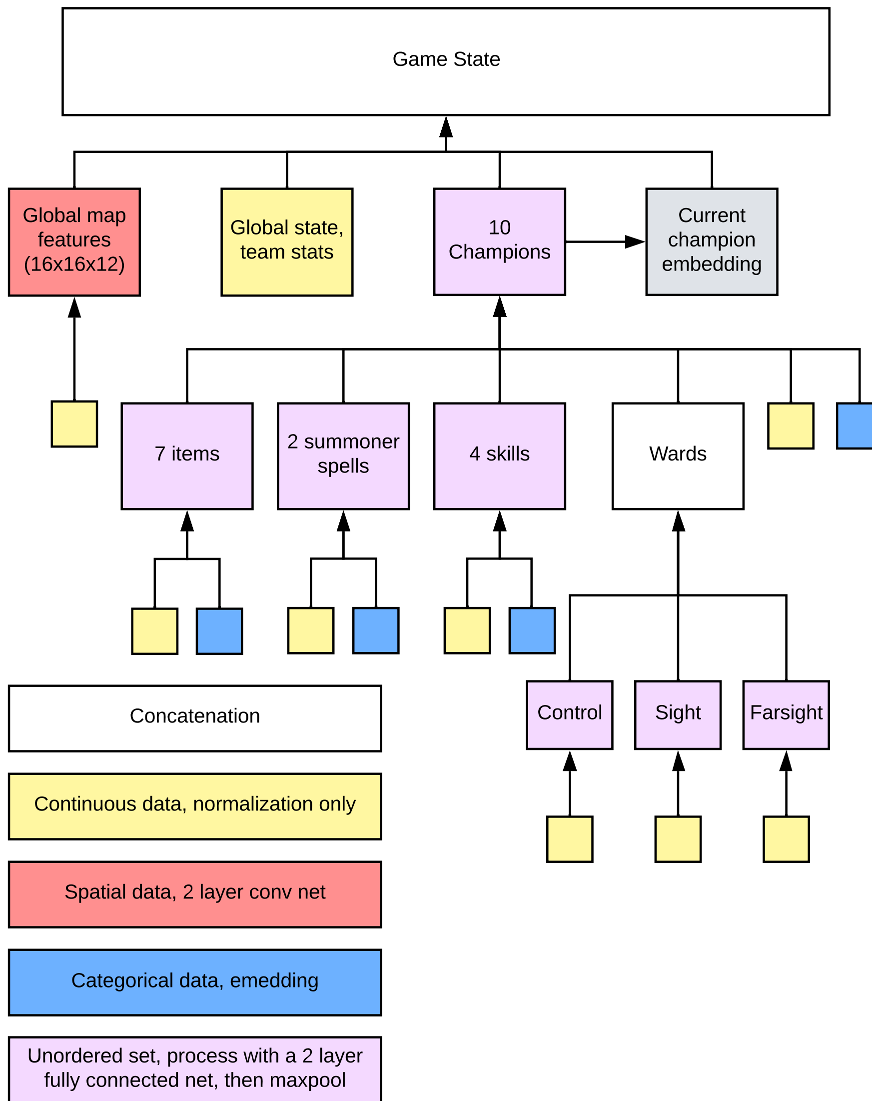


Figure 5.5: Processing of structured multi-modal data using a neural network.

	Pos	Minion	Monster	Kill	Turret	Inhib	Herald	Dragon	Baron	Sum
10s	2.542	**0.5305**	**0.5435**	1.082	0.9349	0.898	0.8379	0.8745	0.8936	9.1369
5s	1.869	0.6403	0.6382	1.045	0.8285	0.7997	**0.7498**	0.8317	0.7459	8.1481
2.5s	**1.291**	0.739	0.699	**1.027**	**0.8087**	**0.7775**	0.7697	**0.7404**	**0.687**	**7.5393**

Table 5.3: Model losses, depending on timestep length.

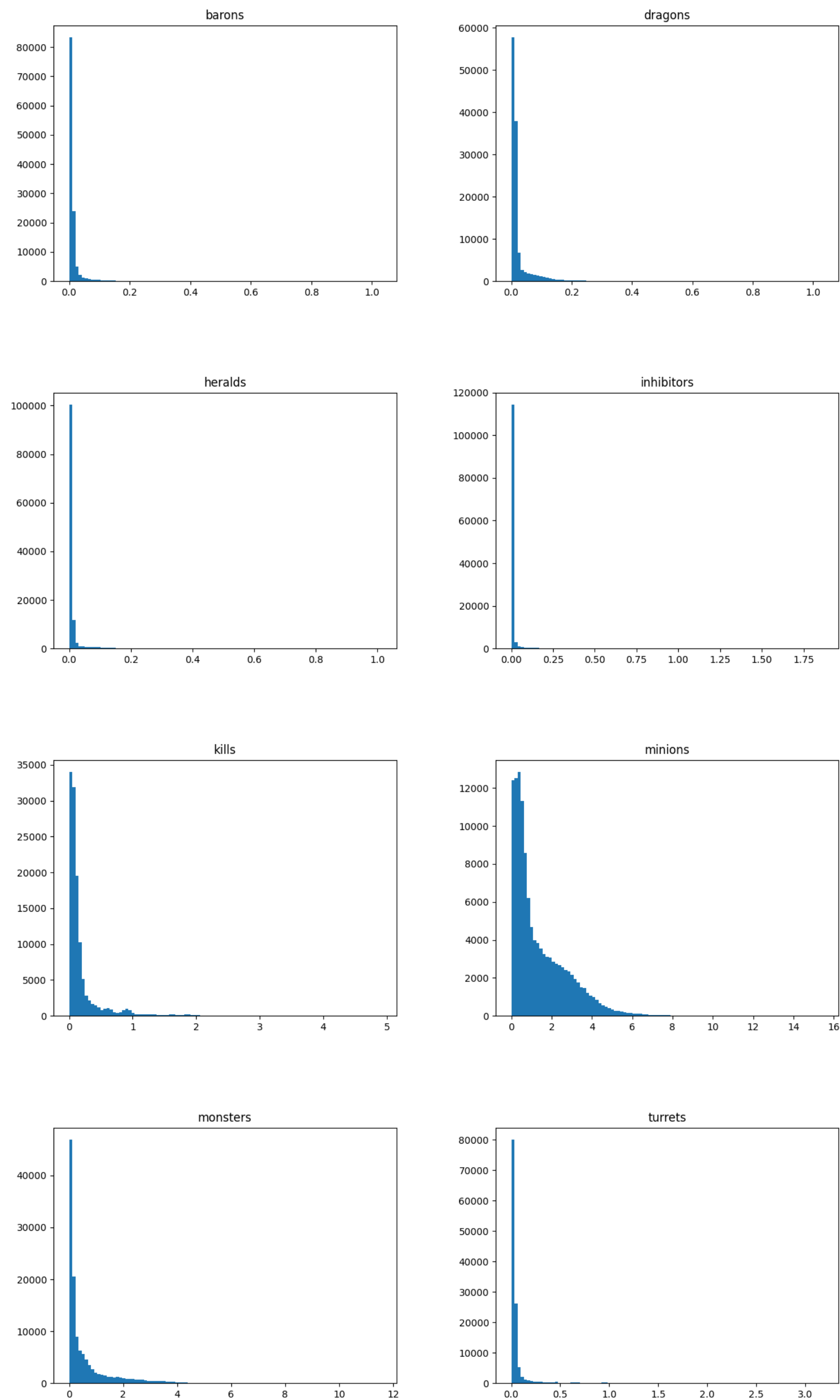


Figure 5.6: Histograms of absolute differences between predicted and true values.

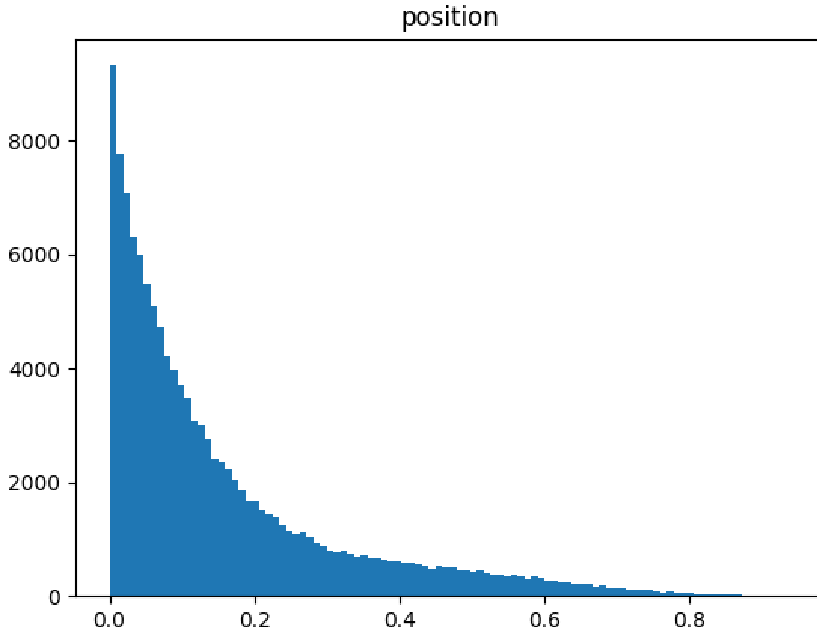


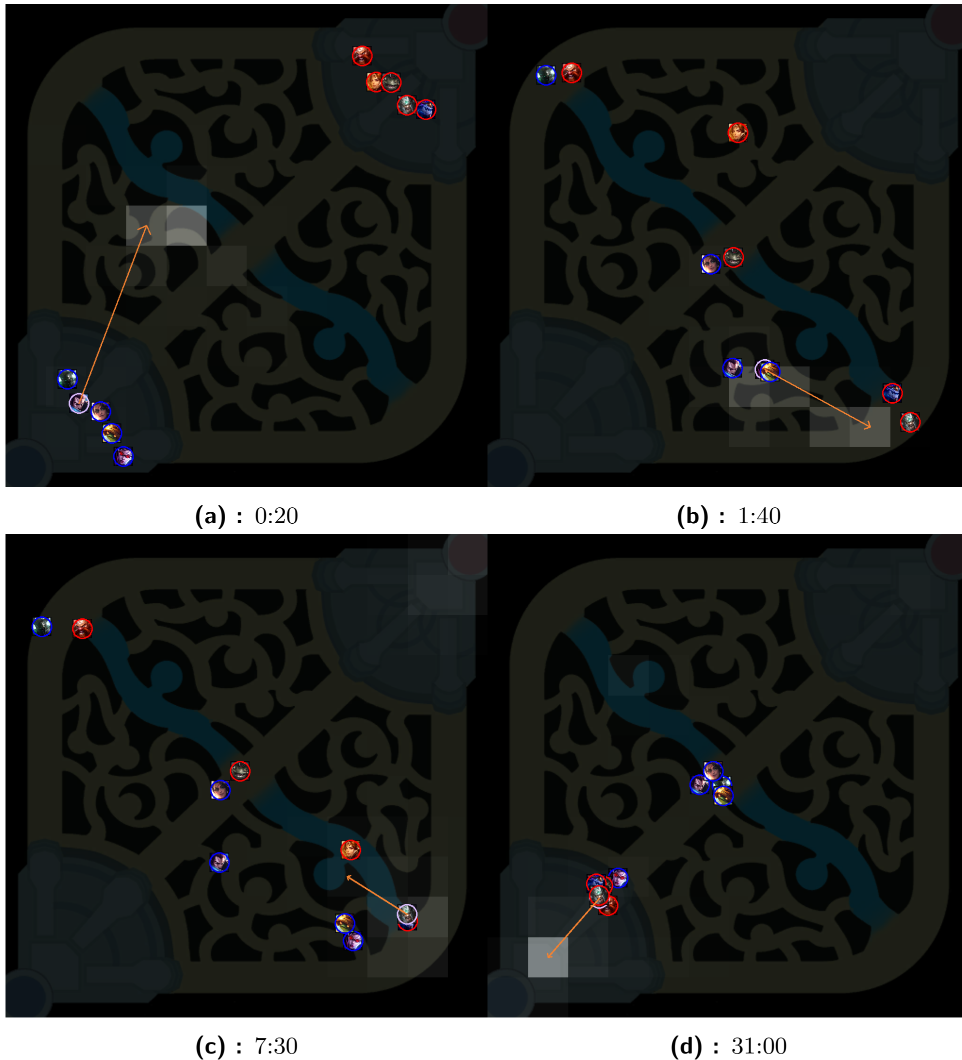
Figure 5.7: Histogram of probabilities of movements performed by players on the test dataset.

	Pos	Minion	Monster	Kill	Turret	Inhib	Herald	Dragon	Baron	Sum
Base	2.541	0.5354	0.5391	1.043	0.9565	0.9185	0.87	0.8576	0.8575	9.1186
+ champ_stats	2.567	0.5307	0.5418	1.059	0.975	0.8487	**0.8388**	0.8729	0.8805	9.1144
+ stats	2.536	0.5289	0.532	1.059	0.9466	0.8778	0.8521	0.8751	0.8805	9.088
- champ_emb	2.518	0.5292	0.5306	1.057	0.9611	0.9115	0.8531	0.8702	0.8984	9.1291
- champ_state	3.416	0.5589	0.5591	1.087	1.004	0.9046	0.8881	0.9003	0.8826	10.2006
- gameTime	2.55	0.5317	0.542	1.05	0.9562	0.8605	0.8429	**0.8572**	0.8649	9.0554
- items	2.533	**0.5272**	**0.5309**	1.05	0.944	0.8739	0.8441	0.8631	0.9036	9.0698
- map	2.567	0.53	0.5462	**1.04**	**0.9265**	0.8792	0.8581	0.8786	0.8695	9.0951
- skills	2.555	0.5294	0.5382	1.047	0.9317	**0.8374**	0.8603	0.8686	0.8677	9.0353
- state	2.541	0.5372	0.5701	1.055	0.9566	0.8945	0.9195	0.9433	0.8642	9.2814
- summoner	2.549	0.5318	0.5452	1.044	0.9405	0.882	0.87	0.8645	0.8686	9.0956
- wards	**2.513**	0.5357	0.5321	1.065	0.9635	0.8526	0.8479	0.8642	**0.8554**	**9.0294**

Table 5.4: Ablation study of features.

	Pos	Minion	Monster	Kill	Turret	Inhib	Herald	Dragon	Baron	Sum	Base Sum
Base	2.551	0.531	0.5395	1.065	0.9449	0.8749	0.8516	0.871	0.8956	-	9.1245
- pos	-	0.5373	0.545	**1.043**	0.9161	**0.813**	0.8552	0.8718	**0.8561**	**6.4375**	6.5735
- minion	2.553	-	0.5378	1.058	0.9675	0.9587	0.8524	**0.8663**	0.8563	8.65	**8.5935**
- monster	2.537	**0.5282**	-	1.063	0.925	0.8616	**0.8414**	0.8687	0.9105	**8.5354**	8.585
- kill	2.543	0.5343	0.5379	-	0.9433	0.9217	0.8597	0.8763	0.8641	8.0803	**8.0595**
- turret	2.545	0.5372	0.546	1.058	-	0.878	0.8577	0.8757	0.8771	**8.1747**	8.1796
- inhib	**2.516**	0.5271	**0.5278**	1.055	0.9628	-	0.8594	0.8734	0.8796	**8.2011**	8.2496
- Herald	2.534	0.5339	0.5345	1.052	0.9216	0.8603	-	0.8931	0.8838	**8.2132**	8.2729
- dragon	2.565	0.5323	0.5404	1.045	**0.9025**	0.8719	0.849	-	0.9057	**8.2118**	8.2535
- Baron	2.526	0.5433	0.5583	1.05	0.9599	0.8989	0.851	0.8619	-	8.2493	**8.2289**

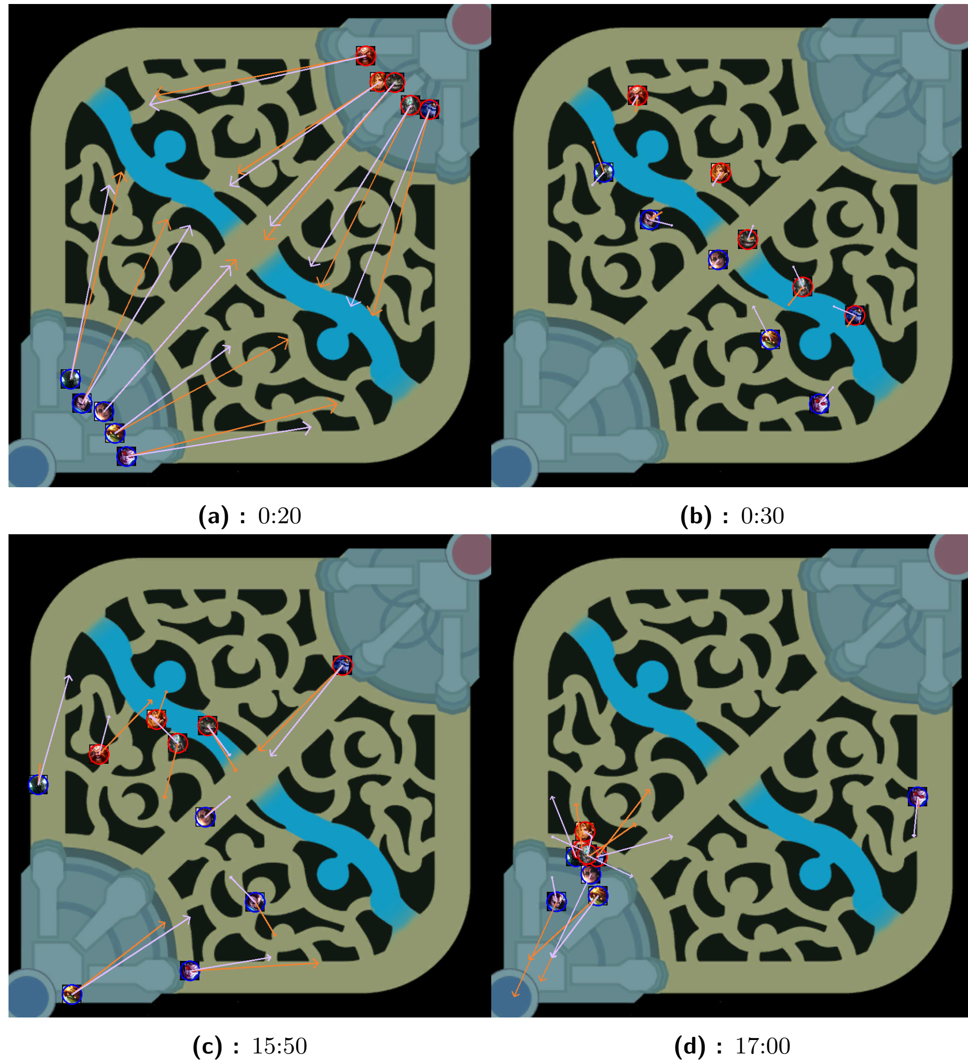
Table 5.5: Ablation study of targets. On the right, the sum of the losses, compared with the sum of the same subset of losses from the base model is shown. The result with the lower loss is in bold text.



(a) : 0:20 (b) : 1:40

(c) : 7:30 (d) : 31:00

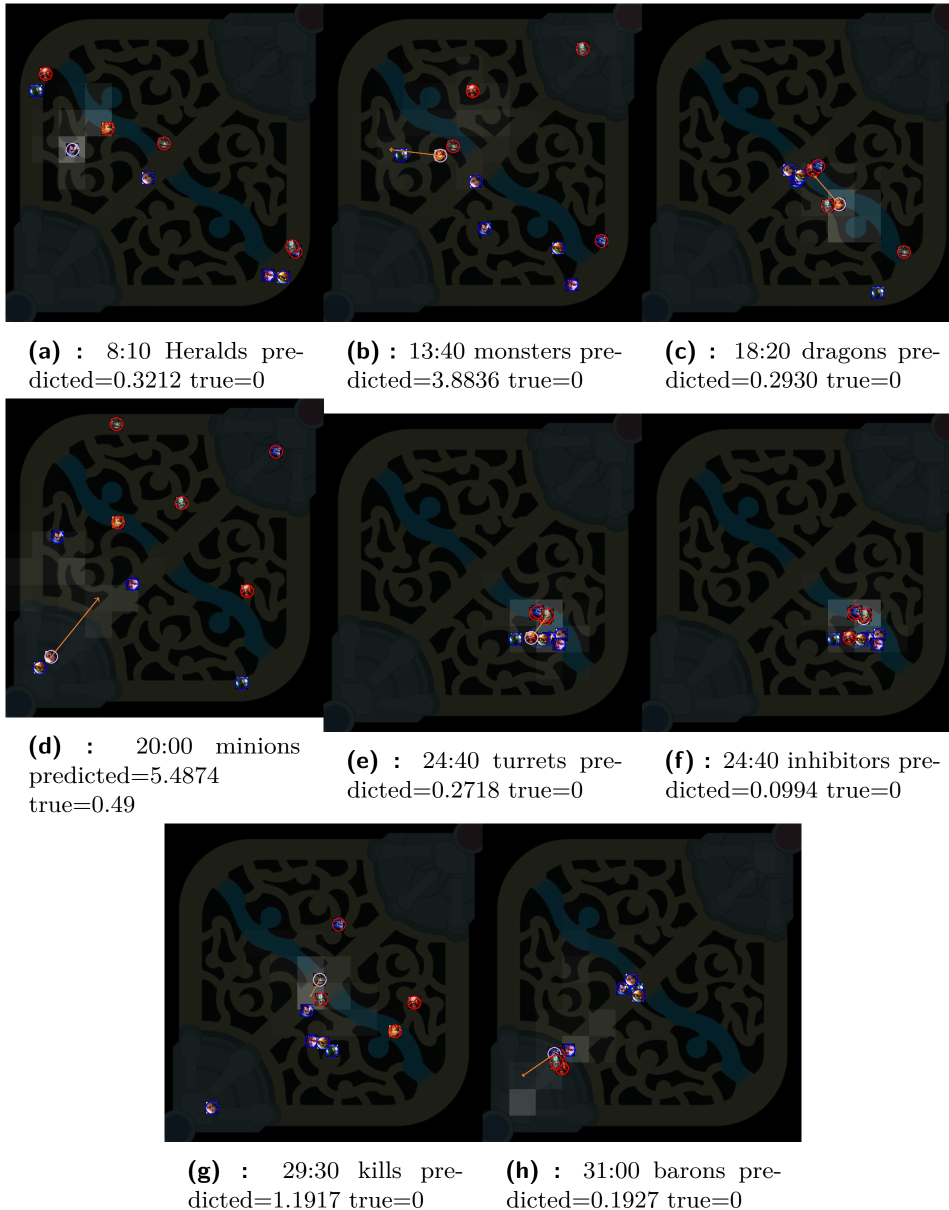
Figure 5.8: Champion movement (orange arrow) and a 2D visualization of predicted positions in different game states. Lighter means higher probability. (**a**), The player goes into a strategic position at the start of the game. (**b**), The player is helping the jungler with the red camp but they need to go to the bottom lane to start last hitting minions. (**c**), The player chooses to go kill the dragon. Staying on the lane and recalling back to base were also options considered by the model. (**d**), After killing the blue team, the red team goes to destroy the enemy nexus and win the game.



(a) : 0:20 **(b) :** 0:30

(c) : 15:50 **(d) :** 17:00

Figure 5.9: Actual movement (orange arrows) and predicted movements (lavender arrows) at different game states. (**a**), All players go into strategic positions at the start of the game. (**b**), All players stay near their positions to guard the entrances to their teams' jungles. (**c**), Two members of the red team go to kill the Rift Herald and the others go in the direction of the mid lane. It was predicted that one of the players going mid would join the others in killing the Rift Herald instead. (**d**), Members of the blue team tactically retreat to their base during a team fight.



(a) : 8:10 Heralds predicted=0.3212 true=0
(b) : 13:40 monsters predicted=3.8836 true=0
(c) : 18:20 dragons predicted=0.2930 true=0
(d) : 20:00 minions predicted=5.4874 true=0.49
(e) : 24:40 turrets predicted=0.2718 true=0
(f) : 24:40 inhibitors predicted=0.0994 true=0
(g) : 29:30 kills predicted=1.1917 true=0
(h) : 31:00 barons predicted=0.1927 true=0

Figure 5.10: Champion movement (orange arrow) and a 2D visualization of predicted positions at detected "bad decision" states. Lighter means higher probability. (**a**), It was expected that the player would go kill the Rift Herald. Instead, they continued slaying monsters in their jungle. (**b**), Instead of slaying monsters in their jungle, the player decided to invade the enemy jungle, not being able to slay any monsters. (**c**), Instead of killing the dragon, the player goes to join a skirmish near the mid lane, killing the dragon about 1 minute after this state. (**d**), We are missing minion data so it is possible there were no minions for this player to farm. (**e**), (**f**), The model expected that there was a chance that the red team would win the team fight and go destroy enemy turrets and inhibitors. (**g**), It took more than a minute before the teams engaged in a team fight and the player was able to get kills. (**h**), After killing the enemy team, the red team can slay Baron Nashor. Instead, they went to destroy the blue team's nexus and win the game, which was definitely a better decision.

Chapter 6 Conclusion

In this Thesis, we developed statistical models for League of Legends that predict information about the game and demonstrated how these models can be used to detect disadvantageous individual decisions. The results indicate that our models were able to predict the outcome of games and model some of the strategic decisions performed by players. While our code is limited to LoL, our proposed methods could easily be generalized to other MOBA games.

This research clearly illustrates how to detect disadvantageous individual decisions, but it also raises the question of how best to give feedback to players so that they get the most out of it and also stay motivated to keep playing.

We introduced basic information about League of Legends and the various entities the game state consists of. Related work in the fields of game AI, MOBA game analysis, and traditional sports analysis was also discussed. While there have recently been many breakthroughs in playing games using AI, analysis of LoL is usually still done using rule-based methods or simple logistic regression models. A thorough search of the relevant literature yielded no other publication in which neural networks were utilized to process full game states in order to analyze individual players' performance and evaluate the decisions they made during games.

We formulated supervised learning tasks and created required datasets for win prediction and macro decision prediction from the raw data. The created datasets have been used to train and evaluate our models.

We developed a logistic regression model that predicts the probability of winning the game. We have shown how this model can be used to visualize the impact of individual game events on the outcome of games.

We proposed a general framework for detecting disadvantageous decisions in MOBA games inspired by recent advancements in game AI. We also showed how our proposed methods, paired with a user interface or visualization software, could aid players to deepen their game knowledge by providing advanced insight into transpired game events.

As further work, we are going to test our trained models on games played by lower-ranked players once we gain access to that data.

Our results could likely be improved upon by either training the models on

a larger dataset or supplying the parts of the game state which were missing in our data. Testing a different neural network architecture, for example using transformers [46] instead of the set processors we used could also lead to further improvements.

We hope that our work inspires further research in using neural networks to infer useful information about games from game states. A model that predicts behavior, like the one we developed, could for example also be used to find game highlights.

Appendix A

Bibliography

[1] statista, eSports market revenue worldwide from 2019 to 2024, viewed 18 December 2021, <https://www.statista.com/statistics/490522/global-esports-market-revenue/>

[2] statista, eSports audience size worldwide from 2019 to 2024, viewed 18 December 2021, <https://www.statista.com/statistics/1109956/global-esports-audience/>

[3] Hrabec, O., 2017. Categorizing play styles in competitive gaming. International Journal of Gaming and Computer-Mediated Simulations (IJGCMS), 9(4), pp.62-88.

[4] Data dragon, a set of static data files that provides images and info about champions, runes, and items. viewed 11 May 2022, <https://riot-api-libraries.readthedocs.io/en/latest/ddragon.html>

[5] Berner, C., Brockman, G., Chan, B., Cheung, V., Dębiak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C. and Józefowicz, R., 2019. Dota 2 with large scale deep reinforcement learning. arXiv preprint arXiv:1912.06680.

[6] Ye, D., Chen, G., Zhao, P., Qiu, F., Yuan, B., Zhang, W., Chen, S., Sun, M., Li, X., Li, S. and Liang, J., 2020. Supervised Learning Achieves Human-Level Performance in MOBA Games: A Case Study of Honor of Kings. IEEE Transactions on Neural Networks and Learning Systems.

[7] Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M. and Dieleman, S., 2016. Mastering the game of Go with deep neural networks and tree search. nature, 529(7587), pp.484-489.

[8] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A. and Chen, Y., 2017. Mastering the game of go without human knowledge. nature, 550(7676), pp.354-359.

[9] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T. and Lillicrap, T., 2018.

A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. Science, 362(6419), pp.1140-1144.

[10] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M., 2013. Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602.

[11] Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T. and Lillicrap, T., 2020. Mastering atari, go, chess and shogi by planning with a learned model. Nature, 588(7839), pp.604-609.

[12] Vinyals, O., Babuschkin, I., Czarnecki, W.M., Mathieu, M., Dudzik, A., Chung, J., Choi, D.H., Powell, R., Ewalds, T., Georgiev, P. and Oh, J., 2019. Grandmaster level in StarCraft II using multi-agent reinforcement learning. Nature, 575(7782), pp.350-354.

[13] Jaderberg, M., Czarnecki, W.M., Dunning, I., Marris, L., Lever, G., Castaneda, A.G., Beattie, C., Rabinowitz, N.C., Morcos, A.S., Ruderman, A. and Sonnerat, N., 2019. Human-level performance in 3D multiplayer games with population-based reinforcement learning. Science, 364(6443), pp.859-865.

[14] Ye, D., Chen, G., Zhang, W., Chen, S., Yuan, B., Liu, B., Chen, J., Liu, Z., Qiu, F., Yu, H. and Yin, Y., 2020. Towards playing full moba games with deep reinforcement learning. arXiv preprint arXiv:2011.12692.

[15] Hochreiter, S. and Schmidhuber, J., 1997. Long short-term memory. Neural computation, 9(8), pp.1735-1780.

[16] Coulom, R., 2006, May. Efficient selectivity and backup operators in Monte-Carlo tree search. In International conference on computers and games (pp. 72-83). Springer, Berlin, Heidelberg.

[17] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A. and Bengio, Y., 2020. Generative adversarial networks. Communications of the ACM, 63(11), pp.139-144.

[18] Jeong, Y., Choi, H., Kim, B. and Gwon, Y., 2020, April. Defoggan: Predicting hidden information in the starcraft fog of war with generative adversarial nets. In Proceedings of the AAAI Conference on Artificial Intelligence (Vol. 34, No. 04, pp. 4296-4303).

[19] De Palma, A., Ben-Akiva, M., Brownstone, D., Holt, C., Magnac, T., McFadden, D., Moffatt, P., Picard, N., Train, K., Wakker, P. and Walker, J., 2008. Risk, uncertainty and discrete choice models. Marketing Letters, 19(3), pp.269-285.

[20] Wang, S., Mo, B. and Zhao, J., 2021. Theory-based residual neural networks: A synergy of discrete choice models and deep neural networks. Transportation research part B: methodological, 146, pp.333-358.

[21] Wong, M. and Farooq, B., 2019. ResLogit: A residual neural network logit model. arXiv preprint arXiv:1912.10058.

[22] Araujo, V., Rios, F. and Parra, D., 2019, September. Data mining for item recommendation in MOBA games. In Proceedings of the 13th ACM Conference on Recommender Systems (pp. 393-397).

[23] Villa, A., Araujo, V., Cattan, F. and Parra, D., 2020, September. Interpretable Contextual Team-aware Item Recommendation: Application in Multiplayer Online Battle Arena Games. In Fourteenth ACM Conference on Recommender Systems (pp. 503-508).

[24] Chen, S., Zhu, M., Ye, D., Zhang, W., Fu, Q. and Yang, W., 2021. Which heroes to pick learning to draft in moba games with neural networks and tree search. IEEE Transactions on Games.

[25] Lee, S.K., Hong, S.J. and Yang, S.I., 2020, October. Predicting Game Outcome in Multiplayer Online Battle Arena Games. In 2020 International Conference on Information and Communication Technology Convergence (ICTC) (pp. 1261-1263). IEEE.

[26] Birant, K.U., Multi-view rank-based random forest: A new algorithm for prediction in eSports. Expert Systems, p.e12857.

[27] Kim, S., Kim, D., Ahn, H. and Ahn, B., 2020. Implementation of user playstyle coaching using video processing and statistical methods in league of legends. Multimedia Tools and Applications, pp.1-13.

[28] OP.GG: LoL Stats, Record Replay, Database, Guide, viewed 26 March 2022, <https://www.op.gg/>

[29] Maymin, P.Z., 2021. Smart kills and worthless deaths: eSports analytics for League of Legends. Journal of Quantitative Analysis in Sports, 17(1), pp.11-27.7

[30] Blitz App - Your personal gaming coach, viewed 26 March 2022, <https://blitz.gg/>

[31] Yurko, R., Matano, F., Richardson, L.F., Granered, N., Pospisil, T., Pelechrinis, K. and Ventura, S.L., 2020. Going deep: models for continuous-time within-play valuation of game outcomes in American football with tracking data. Journal of Quantitative Analysis in Sports, 16(2), pp.163-182.

[32] Reyers, M. and Swartz, T.B., 2021. Quarterback evaluation in the national football league using tracking data. AStA Advances in Statistical Analysis, pp.1-16.

[33] 1.10. Decision Trees, viewed 9 May 2022, <https://scikit-learn.org/stable/modules/tree.html>

[34] Weight Initialization for Deep Learning Neural Networks, viewed 18 May 2022, <https://machinelearningmastery.com/weight-initialization-for-deep-learning-neural-networks/>

[35] He, K., Zhang, X., Ren, S. and Sun, J., 2016. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).

[36] Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H. and Bengio, Y., 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. arXiv preprint arXiv:1406.1078.

[37] Mirhoseini, A., Goldie, A., Yazgan, M., Jiang, J.W., Songhori, E., Wang, S., Lee, Y.J., Johnson, E., Pathak, O., Nazi, A. and Pak, J., 2021. A graph placement methodology for fast chip design. Nature, 594(7862), pp.207-212.

[38] Hunter, J.D., 2007. Matplotlib: A 2D graphics environment. Computing in science & engineering, 9(03), pp.90-95.

[39] League of Legends Wiki, viewed 18 May 2022, <https://leagueoflegends.fandom.com/wiki/League_of_Legends_Wiki/>

[40] Huang, Z., 1998. Extensions to the k-means algorithm for clustering large data sets with categorical values. Data mining and knowledge discovery, 2(3), pp.283-304.

[41] Nelis J. de Vos, kmodes categorical clustering library, <https://github.com/nicodv/kmodes/>, 2015-20214

[42] McInnes, L., Healy, J. and Melville, J., 2018. Umap: Uniform manifold approximation and projection for dimension reduction. arXiv preprint arXiv:1802.03426.

[43] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L. and Desmaison, A., 2019. Pytorch: An imperative style, high-performance deep learning library. Advances in neural information processing systems, 32.

[44] Harris, C.R., Millman, K.J., Van Der Walt, S.J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N.J. and Kern, R., 2020. Array programming with NumPy. Nature, 585(7825), pp.357-362.

[45] Kingma, D.P. and Ba, J., 2014. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

[46] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł. and Polosukhin, I., 2017. Attention is all you need. Advances in neural information processing systems, 30.