TÉCNICO LISBOA

Creating AI Bots in DOTA2

Paulo Henrique Fernandes Ritto

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Carlos António Roque Martinho
Prof. João Miguel De Sousa de Assis Dias

Examination Committee

Chairperson: Prof. José Luís Brinquete Borbinha
Supervisor: Prof. Carlos António Roque Martinho
Members of the Committee: Prof. Francisco António Chaves Saraiva de Melo

November 2018

Acknowledgments

This thesis' buildup would have not been possible without all the support I received during the whole development process. Firstly, I would like to thank my supervisors, Professors Carlos Martinho and João Dias, for all their guidance and feedback from the moment of this work's outset to its completion, always keen to help with any problems encountered. Secondly, I have to express my gratitude to the github community of Clarity parser, which promptly answered all my questions and doubts, regarding this tool. Finally, I would also like to thank my friends and family, for all their encouragement, patience and support throughout my entire journey at Instituto Superior Técnico. A big thanks to all of you.

Abstract

There has been a considerable amount of research regarding the application of machine learning to DotA2 and other games of the same genre, which are characterized by complex environments, with vast search states, and where decision-making must be performed in real-time, often under uncertainty or unfamiliar scenarios. This work focuses on the design of a supervised-learning recognition system to classify strategic decisions performed throughout the first 15 minutes of DotA2 matches. Main steps involved are: definition of pattern classes, feature extraction and selection, collection of training and testing samples from human game-play and the learning of an Artificial Neural Network classifier, capable of identifying the correct strategies to be employed at different times throughout matches. Any possible integrations of the developed work with the game's Bot API are also explored. The main purpose is to ultimately use the model developed to improve strategic decision-making of AI bots in this game. Our solution is evaluated with specific metrics, commonly used in classification or pattern recognition tasks, such as precision, recall, f-measure, among others, which dictate an overall performance of our model. Although far from perfect, the obtained results prove that the built classifier performs considerably better than any model making random classifications. This document also alerts to other problems that remain unsolved: lack of proper integration of Artificial Inteligence models with the game's API and the nonexistence of any automatic-labeling methods for DotA2 strategies.

Keywords

Artificial Intelligence, Real-Time Strategy, Multi-player Online Battle Arena, Machine Learning, Supervised Learning, Bot Development

Resumo

Recentemente, tem ocorrido uma quantidade considerável de pesquisa sobre a aplicação de Machine Learning ao DotA2 e a outros jogos do mesmo género, sendo caracterizados por ambientes complexos, com vastos espaços de estados, onde a tomada de decisão tem de ser realizada a tempo real, frequentemente em situações de incerteza. Este trabalho foca-se no desenvolvimento de um sistema de reconhecimento, baseado em aprendizagem supervisionada, com o objectivo de classificar decisões estratégicas tomadas ao longo dos primeiros 15 minutos de cada partida de DotA2. Os principais passos envolvidos são: a definição das classes padrão, seleção e extração de atributos, coleção de amostras de jogabilidade humana de alta habilidade para treino e teste e a aprendizagem de um classificador baseado em redes neuronais, capaz de identificar as estratégias correctas a ser aplicadas a cada momento de jogo. Quaisquer possíveis integrações deste trabalho com a plataforma de agentes do jogo, são também exploradas. O objectivo principal é utilizar o modelo desenvolvido para melhorar o processo de decisão estratégico dos agentes artificiais existentes no jogo. A solução é avaliada através de métricas específicas para este tipo de tarefas de classificação, como a precision, recall e f-measure, entre outras, que ditam o desempenho geral do nosso modelo. Resultados obtidos demonstram que o classificador desenvolvido, ainda que longe de ideal, é consideravelmente melhor que qualquer outro modelo que realize classificações aleatórias. Este documento alerta também para outros problemas que continuam por resolver: a falta de uma integração de modelos de Inteligência Artificial com a plataforma de jogo e a não-existência de métodos de anotação automática para estratégias do DotA2.

Palavras Chave

Inteligência Artificial , Estratégia em tempo real, Arena de Batalha Multi-jogador Online, Aprendizagem de máquinas, Aprendizagem Supervisionada, Desenvolvimento de Agentes

Contents

List of Figures

List of Tables

Acronyms

CNN	Convolutional Neural Network
KSMO	Kohonen's Self-Organizing Map
LSTM	Long Short Term Memory
RBF	Radial Basis Function
ANN	Artificial Neural Network
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
AUC	Area Under the ROC Curve
ROC	Receiver Operating Characteristic
MCC	Matthew's Correlation Coefficient
TP	True Positive
FP	False Positive
TN	True Negative
FN	False Negative
SMOTE	Synthetic Minority Over-sampling Technique
GUI	Graphical user interface
MLP	Multi-Layer Perceptron
ReLU	Rectified Linear Units
NN	Neural Network

ARFF	Attribute-Relation File Format
WEKA	Waikato Environment for Knowledge Analysis
RTS	Real-Time Strategy
MOBA	Multi-player Online Battle Arena
ML	Machine Learning
AI	Artificial Intelligence
SL	Supervised Learning
RL	Reinforcement Learning
API	Application Program Interface

1

Introduction

Contents

In the last few years, real-time strategy games have played a huge role as Artificial Intelligence (AI) researchers test-beds in several areas, such as case-based reasoning and planning, machine learning, deep learning and adversarial search [1]. More recently, Multi-player Online Battle Arena (MOBA) games, which were born from this genre, also became a target of AI study. However, the majority of traditional AI techniques fail to create something that can play at human level, due to this type of games characteristics like making decisions under uncertainty and deciding in real-time, in an non-deterministic environment with vast search states, making search-based techniques inapplicable. As so, one of the goals of AI research is to be able to develop techniques that can be directly applied to these domains for solving the existing constraints, easing the game developers' jobs to include a good AI system in their games.

In this work, we propose a method for dealing with existing issues in strategic-planning of AI agents for the game DotA2[1], aiming to improve their decision-making and adaptability to different in-game situations. Specifically in this first section, my motivations are stated, along with the actual problem and hypothesis this thesis aims to solve. It concludes with the main objectives wished to achieve.

1.1 Motivation

One of the most noticeable flaws found in games these days, usually has to do with poor Artificial Intelligence systems (weak performance, inaccurate decision-making and lack of adaptability to unfamiliar scenarios), specifically in MOBAs where the amount of regular updates performed can be cited as one of the reasons for the low AI research, in these type of games. However, games provide limited environments that can reproduce many characteristics from real world, and where winning or losing have well-defined rules, making it easy to evaluate such criteria. This has driven many AI experts to use games as testbed for development of AI: producing a better performing AI content and hopefully benefiting the research by reducing costs of test.

We've all seen how brilliant AI can be in certain cases, taking for example the most recent Google's machine learning application AlphaZero[2] which taught itself how to play chess in only 24 hours of self-playing, being able to beat not only chess top professional players but also the best available chess engines. This machine is said to have a lower number of state evaluations than other existing chess machines however it compensates that by using its deep neural network to focus more selectively on the most promising move-variations, arguably providing a more "human-like" approach to search[3].

The good thing about DotA2 and other similar MOBAs is the amount of new strategies that are continuously being developed as the game changes, which always keeps the game interesting and

[1] Valve Corporation, Defense of the Ancients 2, 2013. Online: http://www.dota2.com/

[2] AlphaZero Chess Engine. Online: https://www.technologyreview.com/s/609736/alpha-zeros-alien-chess-shows-the-power-and-the-peculiarity-of-ai/

[3] Chess24 news about AlphaZero. Online: https://chess24.com/en/read/news/deepmind-s-alphazero-crushes-chess

challenging to its players. The team which performs better at an early-stage of the game, is usually the one with higher chances to win, making this stage extremely important. This lead us to focus our development in the early-game phase (in the first 15 minutes of a DotA match), where strategic decision-making is a decisive factor.

In DotA2, there are five different game-roles to be played. The strategies applied to be successful in a match will differ based on the specific role each player is taking, with each role having the most impact in the game, at some point. The *Support* role is arguably the most important in earlier phases of any match, due to starting out as the strongest and falling off in relevance, in later stages. For this reason, and since our work will only concern the earliest moments of the game, in order to simplify the problem, we decided to focus our approach on this specific role. In fact, small classification experiments were initially conducted with the *Carry* role, which usually starts out weak, increasing its impact in later stages. As expected, results obtained were predictable and not very interesting due to the fact that, during initial-phases of the game, this role only changes between 2 or 3 strategies, making the decision-making involved quite simplified. On the contrary, the *Support* role is arguably the richest strategy-wise, interchanging between strategies the most, during initial phases of each DotA2 match.

1.2 Problem Description

This thesis focuses on the development of AI Bots for the game DotA2. The decision process of AI agents in this game comes in two forms: *tactical*, which has to do with specific micro-action behaviour, like moving to a certain location or using an ability; and *strategic*, which has to do with a higher-level behaviour related with general objectives, like helping an ally or attacking and enemy hero, which represent more complex behaviours, encompassing a whole sequence of micro-actions. The available default AI Bots have different levels of difficulty: in the hardest mode, they show a decent tactical behaviour (mainly because their reaction time is improved) but still have a poor strategic decision-making. As so, the main problem addressed is how to develop a model that has a better control of the strategies applied by agents during early stages (first 15 minutes) of a DotA match.

This can be divided in two sub-problems: how to properly identify different strategies employed by analyzing human game-play; and how to use that information to improve AI agents' strategic-planning.

1.3 Hypothesis

From the previously described research problem, we can isolate the following testable hypothesis:

- *By developing and training of an Artificial Neural Network classifier, we will be capable of identifying, for the support role, the correct strategies employed during the first 15 minutes of a match,*

based off the labeling of replays by highly-skilled players.

1.4 Objectives

The main objectives this thesis aims to accomplish are:

1. Research and Analyze previous and relevant works to our problem, about the application of Artificial Intelligence in games
2. Design and Develop AI agents for DotA2
3. Implement Learning Algorithms that can be used for training the Agents
4. Evaluate the viability of applying AI techniques to improve AI agents' behaviour in DotA2

1.5 Document Outline

The remainder of the document is organized as follows:

The following section, **Related Work**, starts by giving a background and brief explanation about the game, mentioning important concepts that will be used later on. It continues by covering the current state of art, when it comes to applications of AI in games, beginning with a more general view, and then going into a more detailed analysis of previous works performed on the Real-Time Strategy(RTS) genre, and DotA2 itself. At the end, a brief discussion will be held with the intent of creating bridges that connect the related work and our intended solution.

Section **Solution** will present the approach methodology to be followed, as well as detailing specifics from the solution's implementation, that consists of six main steps: initial steps are responsible for gathering data and building the dataset to be used, while later steps are responsible for training and testing our to-be-developed classifier, as well as providing respective accuracy results, which also serve as evaluation for our work. To conclude this chapter, some limitations encountered, regarding the integration of the developed model with the game's API, are thoroughly explained.

The ending section, **Conclusion**, is divided in three sub-sections, starting with the main contributions of our work, sharing ideas and problems that remain unsolved and may be dealt with in future works, and finishing with a description of some personal lessons learned with the development of this thesis.

2

Related Work

Contents

2.1 Background

With the release of the Real-Time Strategy (RTS) and Multi-player Online Battle Arena (MOBA) genres, AI found a challenging testbed emerging, where not only machine behaviour was important, but also dealing with a high number of complex variables, that would directly affect the chances of winning or losing the game. Features like resource management and coordination, among others, bring a whole new level of difficulty to AI research: agents are now built in uncertainty conditions and complex state spaces, in games where frequently human intuition is a game-deciding factor. DotA was the first game of the MOBA-genre, bringing all these new constraints to the topic of producing AI systems in games. In the following subsection, a brief explanation about its origins is provided, as well as summary of how the game works.

2.1.1 Game History: Defense of the Ancients

The game was firstly developed, back in 2003, with a tool present in Warcraft III: Reign of Chaos[1] game, that allowed any player to create their own custom scenarios and maps. Many variations of the original concept were created, the most popular being Dota Allstars which was then simplified to DotA.

With the appearance of newer MOBAs at the time, like League of Legends[2] and Heroes of Newerth[3] which had newer and better features, and due to Warcraft III engine limitations in improving DotA, Valve Corporation[4] decided to release a sequel in 2013, called DotA2, with completely different but improved graphics and gameplay, without using the old engine, in order to be able to compete with these newer and similar games, that kept coming into the gaming scene. It is important to not forget that all these big popular games nowadays, like League of Legends, were all based on DotA, making it the "father" of the MOBA genre.

2.1.2 Gameplay

In DotA there are two opposing teams, called *Radiant* and *Dire*, whose goal is to destroy each other's base, in order to win the game. Each team consists of five players, that should work together to take down the *Ancient*, which is the enemy's base main structure.

The game area consists of two base sites, that are in opposite locations, and three main *lanes*, which are routes that the players should take to get to their opponents' base, as we can see in the following figure 2.1.

[1] Blizzard Entertainment, Warcraft Three: Reign of Chaos, 2002.
[2] Riot Games, League of Legends, 2009. Online: http://play.euw.leagueoflegends.com/
[3] S2 Games, Heroes of Newerth, 2010. Online: http://www.heroesofnewerth.com/
[4] Valve Corporation, 1996. Online: http://www.valvesoftware.com/

Figure 2.1: The Game Map, showing the three different Lanes: *Top*, *Middle* and *Bottom*. The Radiant base is located at the bottom-left corner and the Dire base at the top-right corner. The defensive structures placed in each Lane, called *Towers* are also shown, marked as painted squares. Everything else on the map is called as the *jungle area*.

For a better understanding of the game, three very important concepts are presented and explained:

- **Heroes**

  When the game is starting, each player has to chose one of several available characters, called *heroes*. Each hero has unique abilities that can be upgraded as the hero evolves. Each unit killed provides gold and experience for the player, two very important resources: gold allows the player to buy items for their hero, which are used to boost the hero stats or even grant new abilities, making it considerably stronger; and experience, that is used to level up (leveling up unlocks new abilities and increases overall strengths of the hero).

- **Defensive Structures & Units**

  Each team has a set of defensive structures, that are in place to protect the team's base, which need to be taken down first, as well as computer-controlled units that periodically spawn from the three possible entrances of each base. Those units, known as *lane creeps*, will run all the way from their spawn location to the enemy base along the lane path, fighting any units from the opposing team that they encounter. Since the lane creeps from each team spawn at the same exact time, both waves of creeps in each lane are designed to meet each other half-way.

  As a player, you'll be 'pushing' alongside your lane creeps, who are very useful in taking down the mentioned defensive structures.

  Neutral jungle creeps are units that have certain camp spawns in different locations of the game map, re-spawning every minute. These are useful in comparison to lane creeps because there

is no need to share the gold and experience you gain with other teammates that are already occupying lane positions, and it's much safer to kill units in the jungle, since the three main lanes are usually much more targeted by enemies.

- **Player & Hero Roles**

  In each team, the players should assume different roles, picking their heroes based on those roles. Some characters are designed to be better at certain roles, and should be chosen in that context. The five main positions that compose a team are:

  1. *Carry Role* - These type of heroes are usually very weak initially, but become the most important later. They are also very dependent on items and levels, so they should be taking most of the two main resources of the game (gold and experience). They usually occupy the easy-lane of each team, which is shown in the previous map picture.
  2. *Midlaner Role* - designed for the player whose hero can have a big impact during all stages of the game. This role is the second when it comes to resources priority, and occupies the mid lane position.
  3. *Offlaner Role* - designed for heroes that have any type of endurance or escaping mechanism that allows them to stay in lane without dying. This role is more a question of survival than actually winning the lane, since the lane is disadvantageous to begin with.
  4. *Roamer/Jungler Role* - designed for heroes that can jungle and gather his own resources without having to share them with his teammates, or for heroes have good abilities which allows them to roam around the map helping a little on every lane.
  5. *Support Role* - designed for heroes that do not depend much on resources and can have impact with abilities alone. These are usually the strongest early game but very weak later since they don't scale well.

2.2 State of Art

This section will present literature reviews of some works done in this area. Michael Mateas, in one of his works, gave a very good definition of the game AI concept [2]:

"The phrase 'game AI' covers a diverse collection of programming and design practices(...)What links these practices together under the single term 'game AI' is a concern with 'intelligent' behavior, that is, behavior that the player can read as being produced by an intelligence with its own desires, behavior that seems to respond to the player's actions at a level connected to the meaning of the player's actions"

To further understand this definition it is required to establish a very important distinction between *game AI* and *game physics*. In contrast to an AI, game physics are responsible for the "dead" part of a game, referring to multiple aspects of the game that do not have behaviour derived from intentions, rather behaving the same way in every iteration. A good early example of game physics application are many of Atari's games [3], such as *Space Invaders*[5], *Asteroids*[6], *Pong*[7] and *Breakout*[8], in which the enemies/environment move purely mechanically (no goals, emotions, etc), which doesn't fit in AI category. This concept, however, was of big importance in the development of these brilliant games, which were of great success at their time, inspiring many others and having a crucial role in technology advancements.

On the other hand, great examples of AI application started with *"Pac-man"* [4], which was one of the pioneer games in introducing proper game AI. In contrast to previously stated games, in Pac-man the details of ghosts' behaviour (enemies) is one of the primary determinants of player experience and it's critical to understand the game, referred by its creator as the *"heart of the game"*. Other notable games worth considering, that brought innovations to the game AI scene [5] are *Half-Life* [6], in which squad AI is used effectively for the first time; *Thief* [7], which brought an accurate sensory model allowing AI actors to respond realistically to light and sounds; *Sims* [8], which introduced basic desires driving choices and actions, also modeling emotional interaction between its characters; *Total War* [9], where thousands of AI-controlled soldiers are featured for the first time, introducing accurate battle simulations; and *Facade* [10], a first-person interactive drama that makes use of natural language parsers allowing the player to coach the AI's and guide the story.

It is important to mention that AI techniques applied in these games are often simplistic in comparison to those used in academic research and other industrial applications [11]. This does not necessarily mean that game AI is poorly done, as can be seen from the previous examples. Some reasons for this have to do with AI not being the main focus for some game developers or simply not having good systems that could allow an AI integration without affecting games' performance. Thankfully, with technology advancements over the years, the improvement of hardware managed to solve some old issues that would constraint AI development like processing time, now allowing to further explore and use higher demanding AI techniques in games, without breaking performance or ruining player experience.

AI has grown a lot in the last few years, providing better player experience, either from improving aesthetic feel to having agents properly competing or aiding the player in some way. Some great examples that showcase the power of a good AI in different gaming areas range from the recently built AlphaGo [12] which managed to "outsmart" and defeat the best Go board-game players, to the impressive self-learning Google's DeepMind AI which taught itself how to walk, run and jump, having absolutely

[5]Online: https://en.wikipedia.org/wiki/Space_Invaders
[6]Online: https://en.wikipedia.org/wiki/Asteroids_(video_game)
[7]Online: https://en.wikipedia.org/wiki/Pong
[8]Online: https://en.wikipedia.org/wiki/Breakout_(video_game)

no prior knowledge of the environment. [13]

The primary focus of this thesis is, however, on the development of AI agents for DotA2, which is a game where real-time actions must be taken, having a huge time-constraint for decision making and less time available for planning. In the following sub-sections, are described works performed in the game and in a similar game-genre, stating their respective contributions to the evolution of AI research in this area.

2.2.1 Artificial Intelligence in RTS games

Since this thesis focuses on the research of AI techniques to apply in a MOBA game, which primarily originated from RTS games, it only seems fair to talk a little about the state of AI research in these.

Developing intelligent agents for RTS games has proven to be a very hard task due to constraints of the genre. Early researches identified six main challenges [14]:

1. Resource management
2. Decision-making under uncertainty
3. Spatial and temporal reasoning
4. Collaboration (between AIs)
5. Opponent modeling and learning
6. Adversarial real-time planning

There are two different types of decision making that need to be taken into consideration in RTS's: *micro-management*, responsible for single unit behaviour; and *macro-management*, responsible for sequencing the overall actions like building units and structures in a certain order, as well as the global strategy and resources management, which involves allocation for different tasks.

A work by Tavares et al. [15] explored the important *Task Allocation* problem in the popular game *Starcraft*. Starcraft's application programming interface called BWAPI, intended for algorithm development, was used for implementing the bot in which was applied an algorithm based on *swarm intelligence*, which is often used for these type of assignment problems like task distribution, and scheduling problems, at least in the gaming context. For the execution of the algorithm, stimulus for all the agent's possible tasks and capabilities were modeled for every agent-task combination, ending up with an agent being controlled by 27 swarm parameters. However, since finding a good combination of parameters may prove impractical when done manually, genetic algorithms were applied to perform this automatically, finding a proper combination that maximizes a global metric of the agent's performance.

Martinho et al. [16] also tackled the *Task Scheduling* problem, essential to the RTS-genre, by implementing an algorithm inspired in *wasp-behaviour* to schedule unit production in a modified scenario from the game *Warcraft III: Frozen Throne*. This newly built scenario allowed the player to still decide what factory type to build and where it is built, but the underlying system would then devise a near-optimal production schedule, according to the quantity and different types of units specified. By evaluating their routing-wasp algorithm, results shown that it was most appropriate for situations with continuously high request rates and for large scale scenarios, requiring less computation power than other algorithms to achieve similar levels of performance.

A common problem when creating agents for Strategy games is their lack of adaptability and improvement, which is very common in scripted implementations using classical AI hard-coded approaches, the most common being *Finite State Machines*(FSM), in which the AI behaviour is decomposed into easier and manageable states like "attacking", "gathering resources", etc... Although these approaches have achieved some success and have been used in some academic RTS AI research systems, this usually turns out to be a problem: every game match is different and requires different methodologies used by the agent to be successful, which is precisely where these approaches struggle: in building an adaptive behaviour. Cunha et al. [17] tackled this problem, also applying swarm intelligence, using a *decentralized approach*, to improve an AI in a game called 'Almansur'. As the name states, in a decentralized solution, each unit has a more complex thought process, allowing them to perform local planning and exchanging requests at unit level. When correctly implemented, this approach allows for quicker reactions and better adaptability from the AI to any unpredictable events that may occur.

Planning Techniques have also been explored in the literature. Ontanon et al. [18] used real-time case-based planning (CBP) for another RTS game, using human demonstration to learn plans, which are then composed to form strategies to properly play the game. This architecture is capable of dealing with both vast decision spaces and the real-time component of the genre. However, planning had some time constraints, which made the same authors improving their previous CBP approach, by experimenting with *situation assessment* [19] for improving the quality and speed of planning retrieval.

Weber et al. [20] also made their contribution by proposing a data mining approach to strategy prediction. This was done by collecting several Starcraft matches replays and labeling the game logs using rules based on analysis of expert play (rules designed to capture a wide variety of game strategies), opting for a *Supervised Learning* approach.

One last thing to add about AI researches in RTS games is that most of them assume perfect information all the time, which is not true because these games are typically only partially observable, due to the "fog-of-war" idea which only allows a player to see areas near to him. For this reason, many AI systems in these games cheat, and Bots have access to the whole game map at all times. Unfortunately, there is still not many published researches taking in consideration this fog-of-war feature in their

AI development.

All the works presented above had to do with applying AI to improve agent's overall *Strategy* playing the game, which corresponds to high-level decision process, and the highest level of abstraction for game comprehension. They did not focus so much on the *Tactical* part of games, since they are a lot more implementation-specific, and related to micro-actions. Also, in these type of games, if it's already known which specific tactics and actions to take to perform a certain strategy, what matters most to build a better AI system is when to employ each strategy, depending on the game situation.

2.2.2 Artificial Intelligence in DotA2

In this section are reviewed some recent applications of Machine Learning in DotA2. One important branch of research has to do with detection and classification of heroes' roles and positions in the game.

A work by Gao et al. [21] targets the classification of both heroes that players are using and the role they are taking, based on data obtained from available matches replays. Ten data-points (referent to performance and play style of each player throughout the game) are collected per match, via the Valve match history web application programming interface (WebAPI), which allows access to data regarding individual performance. From each data-point collected, 275 features were extracted and used for hero classification. The classification method itself used supervised learning techniques, specifically *logistic regression* and *random forest* classifiers. Each classifier was trained using 90% of the dataset and then used to predict the remaining 10% data. Two different datasets were used: one containing public matches data and the other containing professional matches data. This process was repeated 20 times, after which the average percentage accurately classified was used to determine the quality of the classifier. The labels classified were hero roles and hero IDs, and the results obtained were better than expected, specially in the hero IDs label in which, due to a big variety of possible heroes, the initially expected accuracy was much lower. The different datasets used had a certain impact on the results obtained: the accuracy of both classifiers in both labels was considerably better when using professional data, due to the fact that professional players are more consistent when playing their roles, and the discrepancy in performances and play styles is lesser than in public matches, where there can be players that range from low-skill to near-professional skill levels. Anyhow, the results from this study were very good overall, showing a massive improvement that can be attained by using logistic regression and random forest supervised learning techniques over a random guess, which would be expected to yield less than 1% accuracy.

Eggert et al. [22] continued to explore this subject, providing approaches regarding the construction of complex attributes from low-level data extracted from replay files, as well as investigating the applicability and performance of *Supervised Machine Learning* to classify player behaviour in terms of specific heroes' roles within a team. By comparing and discussing the effectiveness of a bigger vari-

ety of supervised classification algorithms, including *support vector machines with sequential minimal optimization* (SMO), *naive Bayes* (NB), and *Bayesian networks*, in addition to the previously mentioned *Logistic Regression* and *random forest decision trees*, even better results were obtained. Another important difference from the previous work has to do with the attributes construction from the gathered data. The already mentioned WebAPI only allows to directly extract some basic summary data about each match, but capturing positional information and fighting behaviour requires more complex processing of the replay data. As so, an attribute construction processing was built on top of an already existing Java-based replay parser, which allowed for the processing of these more complex attributes, such as 'early movement', 'teamfight participation', among others.

Based on the performed attributes selection, different classifiers were trained and tested in the same manner as in the previous work (using 10-fold cross validation on the dataset), and evaluated according to three established performance metrics: accuracy, mean absolute error and area under ROC (AUC). The whole classifying process was then repeated but with a reduced set of classes (assuming only three player roles: carry, support, and solo lane), which had a considerably better outcome. This work concluded that, although not the best performing classifier, *Logistic Regression* is very stable and most suited for this domain. The classification for the reduced set of classes was very successful, with an astounding average of 96% accuracy, which shows this approach should be applicable to other similar games, with slight modifications, reinforcing the benefits of applying supervised learning techniques to the MOBA genre.

Johansson et al. [23] also experimented with supervised learning classifiers to predict the winning team based on partial collected data from replays. About 15000 replays were gathered (only from high-skill level bracket) as data set and used for training with multiple algorithms like *Random Forest*, *Naive Bayes*, *NNge* (*k-Nearest Neighbour* with generalization) and *Support Vector Machine* (SVM). After initial results proving that SVM and NNge were unsuitable due to excessive training times(over 12 hours), the authors decided to focus solely on testing different parameterizations of *Random Forest*. RF had the highest prediction accuracy of 88.83%, concluding that partial game-state data can be used to accurately predict the results of an ongoing game, by applying machine learning techniques.

In MOBA games, correct interaction between players and better planned team fights are ultimately what dictates the winning team. As so, another obvious branch of research in DotA2 is <u>encounter detection and combat results prediction</u>.

Yang et al. [24] explored this concept by analyzing combat logs from collected matches replays and modeling combat as a sequence of graphs in order to extract patterns that could predict the outcome, not only of team fights but of entire games. Their approach to discover patterns in combat tactics was to attribute features to the graph, by computing graph metrics. After potential features are extracted, a method of feature selection is applied to identify a set of predictive features, which are used to build a

decision tree that predicts the outcome of a game. A set of combat rules is then extracted from the built tree and used to identify patterns in combat that are predictive of success. The whole process can be better captured in the following picture:

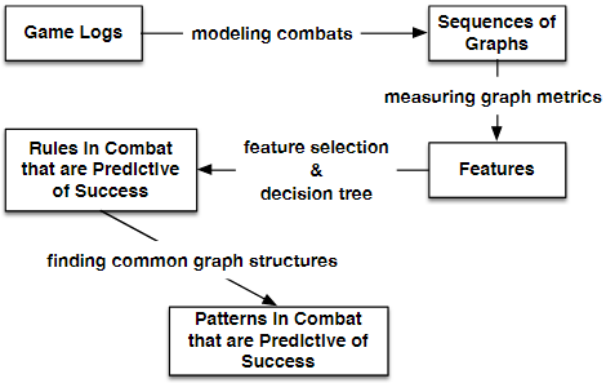


Figure 2.2: An overview of the explained graph theory methodology, by Yang et al. [24]

Results obtained revealed an accuracy in predicting combat results of 80%, showing that graph theory can certainly be helpful in combat predictions of similar games.

Another interesting work about this subject, by Schubert et al. [25], tackles the *Encounter Detection* problem by providing a definition of 'encounter' based on spatial positioning, and using an algorithm to extract these, from available match replays. An experimental evaluation based on their collected dataset examines the importance of performance evaluation metrics, and their predictive power, which are present in the following graph.

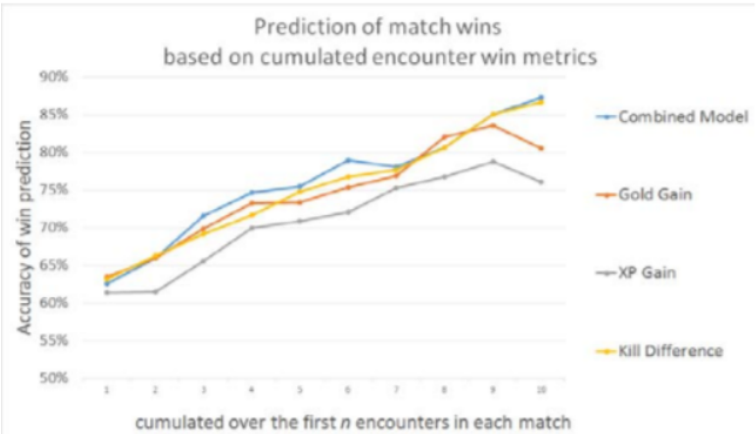


Figure 2.3: Win prediction based on cumulated encounter outcomes, by Schubert et al. [25]. The encounter results are described by XP gain, Gold gain, and the number of killed opponents. Additionally, the blue line shows the performance of a combined model using all three metrics for the encounter result.

By evaluating the performance metrics after just 10 encounters, there is almost 90% of success in correctly predicting the winning team, which proves that the use of algorithms for Encounter Detection

is beneficial for this branch of research.

In contrast to RTS-games,the gameplay in MOBAs is much more focused on tactical combat, which lead Drachen et al. [26] to investigate three data-driven measures of spatio-temporal behaviour in DotA2: *zone changes*, *distribution of team members* and *time-series clustering via a fuzzy approach*. A method for obtaining accurate positional data from replay files is also presented. An available replay parser called 'Dotalys2' by Tobias Mahlmann [27] was used and further improved to be able to extract precise spatio-temporal information from the files, specifically x-y coordinates of the position of all heroes and the time stamp. For the analysis of *zone changes*, a grid map with zonal information was used. The trajectories of players were assigned to a zone for each time stamp, and the number of times each player changed zone was calculated. For the analysis of *team members' distribution*, the distance between all team players was calculated, using the average euclidean distance between pairs of players. An interesting analysis from a scientific point of view is the *time-series cluster*, in which unsupervised learning was utilized via time-series clustering of average distance between players, per second, since this technique had been used in the past to find patterns in non-spatial player behaviour [28]. The objective of this was to find matches with similar movement patterns and explore factors that may lead to different behaviours. *Permutation Distribution*(PD) was used to determine similarity between time series. The PD of a time series can loosely be defined as a measure of complexity, where similarity is determined by the divergence between the distributions of two time series. The authors opted for this clustering method since it was the most computationally efficient, running in linear time, and had no scaling problems that Manhattan or Euclidean Distances presented. For each time-series, PD was computed and fuzzy clustering algorithms were applied on the resulting matrix. The best solution encountered was determined with 3 fuzzy clusters.

The main observations drawn were that, at a higher skill bracket, players tend to make more zone changes, and play more united, informations that could be useful for the community of players in general to improve their game.

This work highlights the use of spatio-temporal patterns to analyze gameplay, contributing with a method for obtaining and visualizing precise positional data from match replays.

Batsford [29] also made some research on the topic of analysis of movement patterns by applying a feed-forward sigmoidal neural network in connection with genetic algorithms, in order to find optimal routes that maximize jungling efficiency in DotA2. The network used had 16 neurons, each one representing important features like the player current hitpoints, current experience, and status (alive or dead) for all creep camp spawns used. Simulations of playing the role of jungler were ran, and its movements were based on the neural network output values. Ultimately, even the best encountered simulation didn't show great results, and the author pointed out that there appeared to be convergence in optimal routes but that more work was needed, highlighting the complexity involved in representing game states.

The most popular application of AI in DotA2 is probably the OpenAI Bot[9], created for the purpose of playing 1vs1 matches. Reinforcement Learning (RL) was applied to build the bot: it started without any prior knowledge of the world, and learned everything from scratch through *self-play*, not using any kind of *imitation learning* or *tree search*. Self-Playing means that the Bot plays with a copy of itself, always having an evenly-matched opponent, which allows it to slightly improve after hours and hours of play. This approach was justified by the complexity of the game: "DotA rules are so complicated that if anyone tried to write down those rules the bot wouldn't even be able of reaching the performance of an average player" [10]. This experiment proved to be of great success: after weeks of constant training (equivalent to thousands of hours playing against itself), the bot managed to beat the top professional human players. By that time, the bot had learned to predict where other players would move, and to improvise in unfamiliar situations. The agent operated off three interfaces:

1. *Observations* - Bot API features, the same set of features that humans can see, like game units and terrain.

2. *Actions* - Accessible by the API, actions were chosen in a frequency comparable to humans. Some of those actions include moving to location, attacking an unit or using an item.

3. *Feedback* - Basic game metrics like health, mana and last-hits were given to the Bot as well as incentives for winning.

Constant improvements were made by OpenAI team, from adding new features to algorithmic enhancements. This development process was eased by the data extracted from the matches the bot played against testers (including pro-players), which allowed to prepare the AI for some possible opponent tricks and tactics that were unknown to it. Very recently (posterior to this thesis' inception), the OpenAI team has also adapted their work to 5vs5 game-scenarios, and shared some interesting information about the agent's internals, regarding their new *"OpenAI Five"* project[11]:

1. Learns entirely from self-play, playing 180 years worth of games against itself every day. It starts out with random parameters and does not use search or bootstrap from human replays.

2. Trains using a scaled-up version of Proximal Policy Optimization (a class of RL algorithms), running on 256 GPUs and 128,000 CPU cores. The agent is trained to maximize the exponentially decayed sum of future rewards, weighted by an exponential decay factor γ.

3. Uses a separate Long Short Term Memory (LSTM) network (a special kind of Recurrent Neural Network (RNN), capable of learning long-term dependencies) for each hero and no human data,

[9] More on DotA2. Online: https://blog.openai.com/more-on-dota-2/
[10] OpenAI DotA2 Bot. Online: https://blog.openai.com/dota-2/
[11] OpenAI Five DotA2 Bots. Online: https://blog.openai.com/openai-five/

learning recognizable strategies. Regarding its structure more specifically, each network contains a single-layer, 1024-unit LSTM that sees the current game state and emits actions through several possible action heads.

4. In order to force exploration of such a combinatorially-vast space environment efficiently, during training, randomization techniques were applied. Exploration is also helped by a good reward, consisting mostly of metrics humans track to decide how they're doing in the game: net worth, kills, deaths, assists and last hits, along others.

5. Other small aspects related to in-game stuff, like hero items and skill builds were all hardcoded, choosing which build to use at random.

6. Regarding coordination, the AI does not contain an explicit communication channel between the heroes' neural networks. Instead, teamwork is controlled by a hyperparameter, ranging from 0 to 1, putting a weight on how much each of hero should care about its individual reward function versus the average of the team's reward functions.

Unlike their previous 1vs1 bot, *OpenAI Five* has yet to beat the best players in the world. However, it already managed to win matches against semi-pro teams (99th percentile), indicating that reinforcement learning can in fact yield long-term planning with large but achievable scale — without fundamental advances, contrary to OpenAI team's expectations upon starting the project. The results obtained show that *self-play* can be a learning technique applied to improve the performance of machine learning systems from below human-level to superhuman. Unlike Supervised Learning algorithms, which will only allow a machine to be as good as their given datasets, self-play systems seem to have no limit for improvement, since the available data keeps getting better at the same pace of its agent.

2.3 DotA2 Bot Scripting API

As an incentive for programmers or even regular players to experiment with machine learning, Valve decided to release a Bot Scripting API, which allows creating new agents or modifying the default Bots that already exist. An advantage of using this, is that all the scripting occurs at server level, so there is no need to simulate mouse clicks or examine screen pixels: scripts can query the game state and issue orders directly to units, making seemingly complex actions like attacking a specific target easily doable by simply calling one of the many already existing functions in the API. The scripts also have access to all game values like entities, locations, cooldowns and many more, allowing this tool to be good enough to achieve different scripting ideas.

When it comes to agent's architecture, bots are organized as follows:

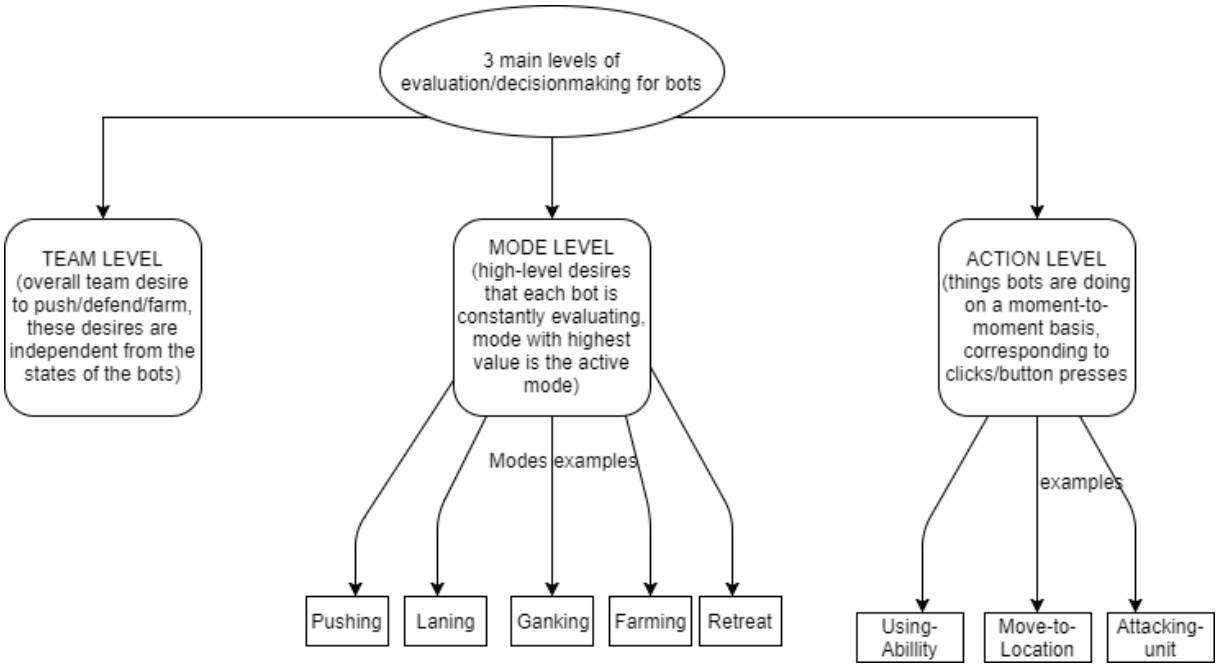


Figure 2.4: The three levels of evaluation and decisionmaking

The best way to understand how it works is that team-level provides guidance on the current strategy of the team, while each bot individually is evaluating the desire scores for each of its modes, taking into account both team-level and bot-level desires. The highest scoring mode becomes the active mode, which is solely responsible for issuing specific actions (depending on the mode it is in) for the bot to perform.

The API is structured in such a way that there are many elements that can be independently implemented by scripts. The actual logic that is overridden depends on which functions the user implements and the specific files that are being coded on.

Each of the following scripting elements has its own scope:

- *Complete Takeover*

  Allows to completely taking control over a Bot behaviour. This is done by implementing the *Think()* function in a file called *bot_heroname.lua* (for a specific bot) or in a file called *bot_generic.lua*(for controlling all bots). The mentioned function is responsible for issuing actions to the bot and is called every frame in lieu of the bot normal thinking code. As so, in this case, there will be no team-level or mode-level thinking.

- *Mode Override*

  Allows to work with the existing mode architecture, but overrides the logic for mode desire and

behaviour. Each existing mode has a file associated, for example *mode_laning_generic.lua* refers to the *Laning* mode for all bots (once again it is possible to replace 'generic' by the hero name, to only change the mode behaviour for one specific hero). In each of the modes files, there are 4 main functions that can be implemented:

1. *GetDesire()*: Simply returns a float between 0 and 1, indicating how much the specific mode wants to be the active mode
2. *OnStart()*: Called when the specific mode takes control as the active mode
3. *OnEnd()*: Called when the specific mode ceases to be the active mode
4. *Think()*: Called every frame while the specific mode is active, issuing actions for the Bot to perform

- *Ability and Item Usage*

  Used to just override decisionmaking around ability and item usage, by implementing in the file called *ability_item_usage_generic.lua* the following functions:

  1. *ItemUsageThink()*: Responsible for issuing item usage related actions
  2. *AbilityUsageThink()*: Responsible for issuing ability usage related actions
  3. *CourierUsageThink()*: Responsible for issuing commands to the courier
  4. *BuybackUsageThink()*: Responsible for issuing a command to buyback
  5. *AbilityLevelUpThink()*: Responsible for managing ability leveling

- *Minion Control*

  Used to just override minions, illusions, basically everything that's under control of the bot's hero. To accomplish this, the *MinionThink(hMinionUnit)* function can be implemented in the specific hero file *bot_heroname.lua*. This function will be called every frame and for every minion in control of the bot.

- *Item Purchasing*

  Allows to override decisionmaking about item purchasing. This is done by impleementing the function *ItemPurchaseThink()* in the *item_purchase_generic.lua* file or in *item_purchase_heroname.lua* (for a specific Bot).

- *Hero Selection*

  Used to handle hero picking and lane assignment. This is done by implementing in *hero_selection.lua* the following functions:

1. *Think()*: Responsible for selecting heroes for bots
2. *UpdateLaneAssignments()*: Returns 10 playerID-Lane pairs, responsible for assigning lanes for each bot

It's important to add that any created scripts can be uploaded by its authors to the DotA2 Workshop, where other players can download and use the scripts in their own games. This is good because it promotes not only a bigger player community interaction but it also contributes to AI programmers sharing ideas (since they can inspect each others codes after downloading). It can also be used to arouse interest of casual players to the Bot Scripting topic and Artificial Intelligence in general.

2.4 Artificial Neural Networks

As will be contemplated in the upcoming chapter regarding our solution, we tackled the main problem addressed in this document with an approach based on Artificial Neural Network (ANN) systems. Consequently, this sub-section serves as a brief introduction to these type of networks, their role in dealing with classification tasks and main advantages in comparison to other methods.

The concept of ANN [30] derives from an engineering approach of the biology term *neural network*: a web of numerous interconnected neurons, responsible for processing and transmitting information through some electrical charge, playing a key role in the human body. Likewise, an ANN consists of a large number of simple processing units that are interconnected through layers. Each of these elements receives inputs from other artificial neurons, which are weighted and summed, the result is then transformed through a mathematical formula, called the activation function, into an output signal [31], as displayed in the following figure 2.5.

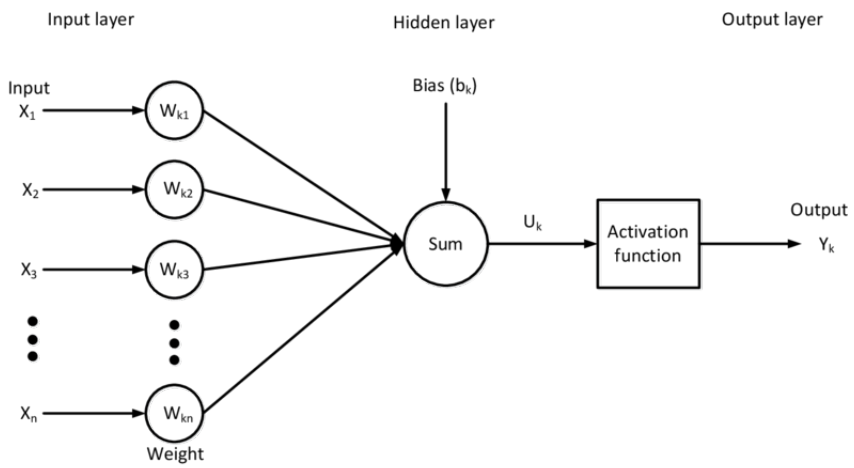


Figure 2.5: Typical ANN structure of a nonlinear model

A work developed by Basu et al. [32], addresses the impact of using ANN systems in pattern recognition problems, identifying research topics and applications which are at the forefront of this challenging field. According to the aforementioned paper, neural networks are mainly characterized by their ability to learn complex nonlinear input-output relationships, using sequential training procedures, and adapting themselves to the data. The most commonly used family of ANNs for pattern classification tasks is the feed-forward type, which includes the Multi-Layer Perceptron (MLP) and Radial Basis Function (RBF) networks.

The MLP network consists of an input layer, an output layer, and one or more hidden layers of neurons in between these two, aiming to increase the ability of modeling complex functions. There are typically three steps involved in regard to data processing [33]: *training stage*, *weight determination stage* and *classification stage*. Usually, the training process starts by assigning arbitrary connection weights which are constantly adjusted to minimize the error obtained, until an acceptable model accuracy on the training data is reached. Specifically, in an MLP architecture, the input that a single hidden neuron j receives from the neurons i of its preceding input layer, may be expressed as:

$$net_j = \sum_{i=1}^{t} w_{ij} p_i \tag{2.1}$$

Where w_{ij} represents the connection weight between input neuron i and hidden neuron j, p_i is the data at the input neuron i and t is the number of input neurons. After having determined the net sum at a hidden node, an output response is provided at the node using an activation function.

Throughout years, many activation functions have been proposed however, most have drawbacks and are case-specific[12]. One of the most used is the *sigmoid* function, which is represented as follows:

$$S(x) = \frac{1}{1 + e^{-x}} \tag{2.2}$$

It can be deducted from the previous formula, that the output of this function will always be in the range (0,1) which is perfect representation of probabilities and classifications, where large negative real numbers will become 0 and large positive numbers will become 1. It has been widely used for this property, which makes a clear distinction between predictions. Despite their main advantage, *sigmoid functions* have lost popularity in recent years, due to a major drawback: the *vanishing gradient* problem, which affects training of networks with gradient-based methods. These methods learn a parameter's value by understanding how a small change in the parameter's value will affect the network's output. In logical terms, if a change in the parameter's value causes a very small change in the network's output, then it won't be able to learn the parameter effectively. And this is what happens when using functions like *sigmoid*, which maps any real numbers to a small range of (0,1). This means that there are regions

[12]Online: https://blog.paperspace.com/vanishing-gradients-activation-function/

of input space that are mapped to a small range, and any considerable changes that might occur there, will have almost no effect in the output, hence the gradient is small, resulting in a network that is not able to learn further, or has at least a drastically low learning rate. This concept becomes worse when there are multiple hidden layers involved: the first layer will map a large input region to a smaller output region, which will be mapped to an even smaller region by the second layer, and so on.

Recently, other alternatives have been used to deal with this issue, like the Rectified Linear Units (ReLU) activation function, computed with the following formula:

$$f(x) = max(0, x) \tag{2.3}$$

ReLU is an extremely simple function, which returns zero, when $x < 0$ and is linear, when $x > 0$. Naturally, its main advantage is that it does not constrain the input space into a small region, avoiding the aforementioned *vanishing gradient* problem.

Going back to detailing the usual training process of an MLP, once the output from the activation function is obtained, the model's error is calculated using a cost function (e.g., the mean squared error) and its value is back-propagated and used to figure out any necessary updates to the weights in the network, in order to have succeeding outputs to be closer to the target output, consequently minimizing the error for each output neuron and the model as a whole. This method is called *Backpropagation* [34], one of the most widely applied algorithms in ANN models.

Once the training process is over, the trained network (with the already updated weights) is then tested on new data in order to evaluate its generalization capability and accuracy. The performance of a model can finally be measured by relying on specific evaluation metrics, commonly used in classification tasks.

On the other hand, RBF networks have a structure consisting of a single hidden layer where each neuron implements a radial-activated function. As so, hidden unit outputs are not calculated using the weighted-sum mechanism/sigmoid activation, typical of MLPs; rather each output Z_j is obtained by closeness of the input X to an n-dimensional parameter vector u_j, associated with the jth hidden unit [35].

The increasing popularity of neural network models to solve pattern recognition problems has been primarily due to their low dependence on domain-specific knowledge and the availability of efficient learning algorithms, inciting the usage of ANN systems in several different applications, some of which mentioned in [32]. Along with these benefits, a study performed by Sharma et al. [36] describes six fundamental characteristics comprised in this unique technology:

1. *Network Structure* - An ANN can either have a recurrent or non recurrent structure. While in a non recurrent network, known as feed-forward, the signal only travels in one way, in non recurrent

structures, also known as feed-back, data flows in both directions, by introducing loops in the network.

2. *Parallel Processing Ability* - Each neuron in an ANN serves as a processing element. Computations required to simulate these networks are mainly matrix ones, and the parallel structure of the interconnection between neurons facilitates such calculations.

3. *Distributed Memory* - neural networks do not store information in a central memory. Instead, information is stored as patterns throughout the network structure: the state of neurons represents short-term memory (since it may change with the next input instance), while the weight matrix forms a long-term memory, which is only changeable on a longer time basis, with gradual input experience.

4. *Fault Tolerance Ability* - Due to their parallel processing ability and distributed memory, ANNs are relatively fault tolerant, since the failure of one or more parts may degrade accuracy but will not break the system, which only happens if all parts fail at the same time. This provides a measure of damage control.

5. *Collective Solution* - Unlike conventional computer processes, where instructions are programmed sequentially, neural networks rely on the collective outputs of all connected neurons, hence stopping the solution process before it is completed would result in a nonsensical outcome.

6. *Learning Ability* - ANNs are capable of adapting to the changing environment while discovering useful knowledge, implicit in received responses. There are three main learning methods: *supervised*, in which the desired output for a set of training examples is provided to the network, thus it learns by example; *unsupervised*, where there is no evaluation of performance provided to the network; and *reinforcement*, which is an hybrid method, where the network is given a scalar evaluation signal (reward), instead of being told the exact desired output.

Although two of the most commonly used ANNs in dealing with classification tasks were already described, there are several other types of ANN worth mentioning due to their ability to solve problems from different areas of application: Convolutional Neural Network (CNN) [37], frequently used in signal processing and image classification tasks; RNN [38], ideal for tasks that require dealing with sequential data, like speech synthesis and music generation; and Kohonen's Self-Organizing Map (KSMO) [39], which is frequently used in medical analysis, to cluster data into different categories.

2.5 Discussion

After carefully exploring previous works about possible AI applications for DotA2, it can be stated that there has been a good amount of research that covers different aspects of the game like utilizing graph theory for predicting the outcome of teamfights, classifying in-game roles by applying different Supervised Learning algorithms, and creating a Reinforcement Learning agent that learned through self-play. All these works were studied and contributed to an increase of my knowledge about this subject, however, none of these covered how limited the available API for the development of AI bots is, and how poorly they behave at a strategic-level, making them not so interesting to play with, from a gaming and scientific point of view. The lack of previous work addressing the strategic problems with DotA2 bots is where the next section of this document fits, in trying to propose a solution, that solves these issues and hopefully incites the future creation of more competent AI agents.

3

Solution

Contents

This chapter is divided in two parts: the solution's general approach and, in a more concrete way, its implementation. The first part summarizes the overall structure of our solution, naming significant 3rd parties used, and stating the main differences from the initially-thought approach. On the other hand, the second part details the implementation, specifying important algorithms and methodologies used, presented in four main steps.

3.1 Approach

Our approach to improve AI agents' decision process at a strategic-level in the game DotA2 was built using a *Supervised Learning* approach, to tackle a strategy-classification problem. The main idea was to make agents able to classify each game state with a certain strategy (from a list of different strategies) that corresponds to the best one to be employed in that state. The correct strategy to take at a given time would vary, based on each agent's in-game role and on other relevant features. To accomplish this, high-skilled human game-play data was captured and annotated by experienced human players, and a classifier was trained and tested, according to gathered info.

Although we had also planned to explore possible integrations of the trained and tested classifier with the DotA2 Bot Scripting API, in order to properly dictate for each Bot which strategy it should be employing at real-time, hopefully resulting in a considerable improvement of their strategy-planning throughout a match, unfortunately this could not be performed due to specific constraints from the DotA2 API, that weren't initially expected. As so, the whole development process to be explained in this section, will only concern the previous steps, while these constraints will be better explained in a later chapter. Therefore, our solution was designed to be run in the following briefly-described steps:

1. *Collecting Data* - replay files, (game matches files, which allow re-running and simulating complete matches that have already happened) to be annotated and used as data, are gathered

2. *Labeling Replays* - previously collected replays are labeled by human players, with the current strategies being used by the *support-role* player of the winning team in that replay file, during the first 15 minutes of each match

3. *Defining relevant features* - in-game features that have an impact in which strategy to apply, are extracted from each replay file, in order to build the data-points needed for our Machine Learning (ML) classifier.

4. *Training the classifier* - our Supervised Learning classifier is trained, using the previously built data-points which are separated in training and validation datasets

5. *Finding accuracy results* - the trained classifier is then tested on new data (testing dataset), to obtain accuracy results

3.2 Implementation

In this section, implementation details are described, ranging from the method used for collecting and labeling replays, to specifics of the parser utilized to extract and define all features used in our ML approach, which is also thoroughly explained, finishing by presenting the results obtained.

Before continuing, it is essential to mention that, although the initial idea was to improve agent's decision-making for each in-game role, we decided it would be best to shift our work to focus only on the *support* role, since this is the hardest yet richest role, due to having the biggest amount of interchanging between strategies in early phases of each game match.

3.2.1 Collecting and Labeling Replays

The first step needed was to gather DotA2 replay files (.dem). It is important to understand what these are: each DotA2 match is unique, represented by a specific ID serial number and, everytime each DotA match ends, a replay file with that ID number is generated in DotA2 Match History Database. These replay files referent to each match, can be queried and downloaded by users and can be run in DotA2 engine to entirely simulate and re-play these matches that already happened, as displayed in figure 3.1.

Figure 3.1: DotA2 engine simulating an already played match, by running its replay file

Initially, an amount of **100** replay files were collected via the Opendota platform[1]. For each of these, the idea was to analyze the replay simulation in DotA2, annotating which strategies the *support* player of the winning team was applying, according to the following list of strategies:

[1] Open source Dota 2 match data and player statistics. Online: https://www.opendota.com/

- *laning*
- *attack*
- *roam*
- *retreat*
- *secret_shop*
- *side_shop*
- *rune*
- *push_tower_top*
- *push_tower_mid*
- *push_tower_bot*
- *defend_tower_top*
- *defend_tower_mid*
- *defend_tower_bottom*
- *assemble*
- *team_roam*
- *farm*
- *defend_ally*
- *evasive_maneuvers*
- *roshan*
- *ward*

Although the replays labeled and analyzed were of high-skilled human gameplay, we decided to have these 20 strategies as our labels, since they represent the existing Bot modes, and would ultimately result in an easier integration of our classifier with the DotA2 Bot Scripting Application Program Interface (API). The whole idea was to create a cause-effect relation between what was happening in-game and which strategy the experienced human *support* player decided to apply in that situation, so it makes sense to have these strategies being the exact same as those that any default game Bot can already employ.

When starting the hand-labeling process and by quickly realizing how time-consuming it was, only **10** replays ended up being labeled. Although this seems like an extremely low amount of data, each labeled replay file contains the strategies being employed by the player, during the first 15 minutes of the match, and the **labeling of these strategies was performed for each second**, which sums up to 900 lines labeled for each replay, effectively resulting in a total of **9000** labeled strategies, if accounting for all 10 replays labeled.

A different method of hand-labeling worth mentioning consists in only labeling strategies for each fixed 15-seconds intervals (instead of our per-second labeling), and assuming that the strategy would remain the same for the whole duration of each interval (*for example, by labeling that, at minute 04:30, the strategy employed is A, we were assuming that for those 15 seconds until 04:45, the strategy employed would always be A*). Although this would result in a substantially faster labeling process and, consequently, more replays and data gathered, it quickly proves to be a faulty method, since the strategies employed during a DotA2 match are constantly changing and this method would directly affect and wrongly skew our accuracy results later on, ended up not being used.

3.2.2 Parsing and Defining Features

To be able to understand when a certain strategy should be employed, a “bridge” between what is currently happening in-game and optimal strategies to be taken needs to be defined. In order to do this, relevant attributes that represent the nuances of different game-states and directly affect strategic decision-making, need to be considered.

After having gathered and labeled our specific amount of DotA2 match replay files, we needed a way to extract the data contained in each of those files, regarding their matches. This step was of extreme importance to build proper features and data necessary to create our *data-points*, which are necessary and would later be used in our Supervised Learning (SL) approach.

Opendota's API provided basic information (received in JSON, URL format) about each match whose replay was requested: *match duration, winning team, team scores, heroes, items, abilities, gold per minute, experience per minute* and *individual score (kills/deaths/assists)* of each player.

The problem with replays was that, although they allowed for complete simulations of matches that had already happened (which means it contains all data regarding those matches), there was not an easy nor evident way of how to extract more complex data from those files, and the basic information already collected from Opendota was insufficient.

After searching for a solution to this problem, existing replay parsers like *manta*[2] and *clarity*[3] were considered. We ended up choosing *clarity* due to having clearer code examples[4], showcasing the parser's capabilities. Not only that, the community around this project was very active and the parser itself was still maintained, regularly updated and improved based on the feedback from its users. This parser allowed to break the data contained in replay files into different categories, from which we can highlight the following:

1. *Combat Log* - a detailed log of events that happened in the game.
2. *Entities* - in-game things like heroes, players, and creeps.
3. *Modifiers* - auras and effects on in-game entities.
4. *User Messages* - spectator clicks, global chat messages, particle systems, overhead events (like last-hit gold, and much more)
5. *Overview* - end-of-game summary, including players, game winner, match id, duration, and often picks/bans.

Having studied the code of some available examples on how to extract specific values of certain entities, we progressed by beginning to write our own code to build a program that would receive any replay file as an input, and would ultimately be able to output all the yet to-be-defined data features, per each second of the match. This time-interval for the output was of extreme importance, since the replays had also been labeled per second. This was possible by using the *@OnTickStart* annotation, which allowed for any info to be processed per game tick. We also learned that 1 second is equivalent to 30 game ticks, so we just had to process info every 30 ticks.

[2]Online: https://github.com/dotabuff/manta
[3]Online: https://github.com/skadistats/clarity
[4]Online: https://github.com/skadistats/clarity-examples

As we finished setting up the main loop of our program, all that was left was to grab the actual data from whatever entity we needed in order to build our planned data features. To further aid on examining exactly which game information each entity contained, we also tried a JavaFX-application[5], to interactively visualize all entity data of a replay at real-time, as shown in figure 3.2.

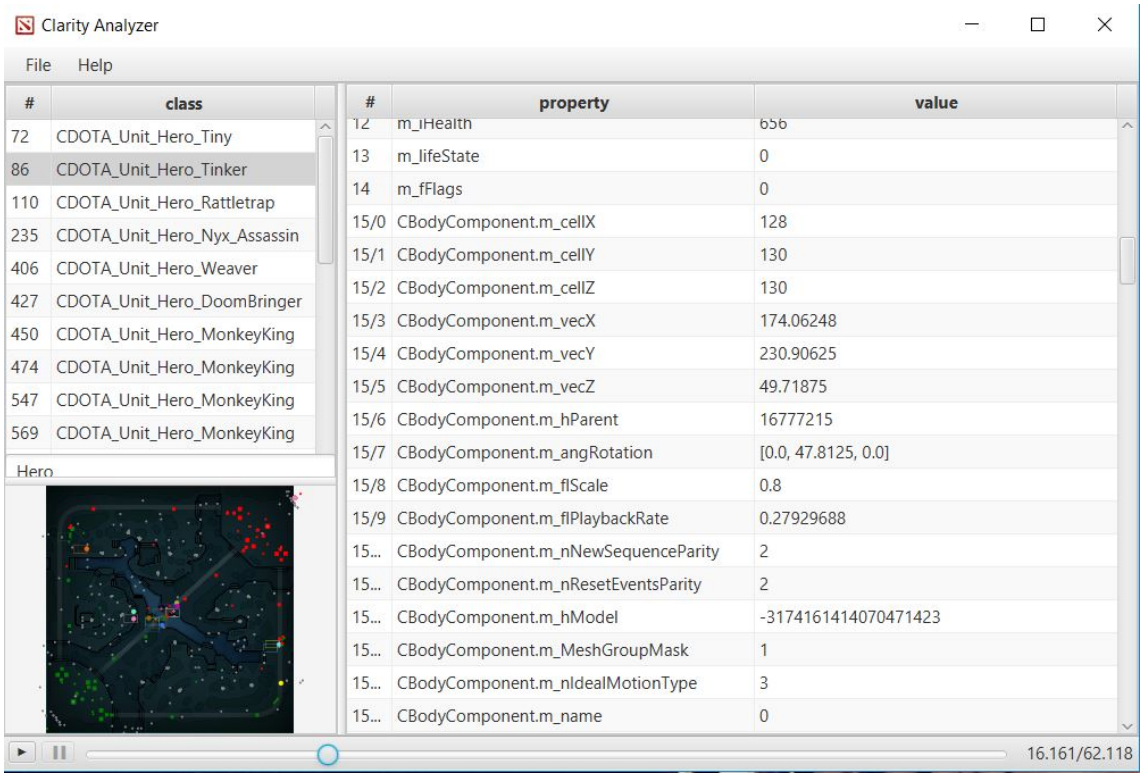


Figure 3.2: *Clarity Analyzer* application, with the selected Entity's name on top-left, and all its properties on the right with their respective values

Once it was clear what classes to call, in order to get the values about any game entity, we then needed to chose among the many available, which game attributes made sense to extract, in order to build proper features for our solution. Initially, we focused on 98 features, although some of them were later discarded due to lack of relevance, as will be explained later on, ending up with a total **70** features.

Our features included basic stats from each hero, like *hitpoints* and *mana*, their *spatial positions*, but also somewhat more complex attributes like *distances between heroes* and *distances to laning and jungling areas*, all of which have a great impact on strategic decision-making.

Initially our accuracy results were very poor, due to not considering the variation of some of our features' values. In sum, each feature-vector contained in our datasets only had values of attributes regarding its specific second, without taking into account how these have changed in comparison to previous time-frames. This is extremely important and was crucial to implement, in order to attain better

[5]Online: https://github.com/spheenik/clarity-analyzer

results, since it allows us to understand what was happening in the previous moment that lead up to the current second, which has influence in the strategies employed. As so, half of our features are the variations of the values obtained in the current second, in comparison to the values of the previous second, which are values that we always keep (the values processed in the previous second are temporarily stored). The next table gives an overview of all features that were considered:

Features	Number of Features
Hero Stats	40
Hero Positions	20
Closest Neutral Camp Hitpoints +	2
Closest Neutral Camp Distance +	2
Closest Lane Creeps Distance +	2
Closest Enemy Hero Distance +	2
Fountain Distance +	2
Hero Networth *	2
Buildings Stats *	24
Roshan Stats *	2

Figure 3.3: Attributes for our classification approach. Features marked with + are not directly available from replays. Features marked with * were initially chosen but ended up being discarded thus did not make part of the final set of features. Any additional DotA2 info/terminology can be found in: https://liquipedia.net/dota2/Main_Page

It was also important to make sure, whatever features had been chosen, that the DotA2 Bot Scripting API (where we would integrate our solution), would be able to obtain all these same values during a match, so that they could be accessed and fed to our ML model, at real-time, to ultimately output the decision-making of the *support-role hero* of the winning team. This was something verified before-hand, by experimenting and editing some Bot default files from the Scripting API, in which we called methods that would return the values of all these same attributes, regarding one hero, just to make sure that they were indeed accessible. The following code, displayed in figure 3.4, was written in a file called *mode_laning_bane.lua*, representing the *laning* strategy-mode of the AI Bot, which is playing an hero called *bane*:

The previous figure showcases one of several available methods that would be used to extract our defined features (in this case, finding the hero current level with *GetLevel()* method), once we had our classifier ready and integrated with the API, in order to feed this info to our model, while a match would be running, and ultimately outputting the correct strategy to apply, at real-time.

The documentation about the terminology of all these methods can be found on the official DotA2 Bot Scripting web page, cited in figure 3.5.

Going back to our java code, we were able to successfully extract and build all desired features from

```
function GetDesire()
    local npcBot = GetBot();
    if(npcBot:GetLevel()<7)
    then
        return 0.35
    else

        return 0.27
    end
end
```

Figure 3.4: Function that represents the bot's desire of remaining in laning mode, based on its hero's current level, whose value is accessed by calling the *GetLevel()* method.

int **GetHealth**()

Gets the health of the unit.

int **GetMaxHealth**()

Gets the maximum health of the specified unit.

int **GetHealthRegen**()

Gets the current health regen per second of the unit.

int **GetMana**()

Gets the current mana of the unit.

int **GetMaxMana**()

Gets the maximum mana of the unit.

int **GetManaRegen**()

Gets the current mana regen of the unit.

Figure 3.5: Example of some more methods, available at https://developer.valvesoftware.com/wiki/Dota_Bot_Scripting

replays and, based on the labeling previously done, we were able to build data-points regarding each second, for the first 15 minutes of each replay. A shell script was also created in order to randomly shuffle all our replays, and then picking 70% to be called by our program and used as training data, and the remaining 30% as testing data. This would guarantee that data-points regarding one replay, would all be either on training or testing datasets. The idea behind this is to figure out if the model is able to generalize its strategies to completely different match scenarios (in some matches, the ideal strategy to be applied in a specific situation might not the be the same in others).

By concluding this step, we ended up with a total of 9000 data-points, each with 70 relevant numerical attributes. Although the data wasn't great in size, it was significant enough to move on to the next step: building, training and testing our classifier.

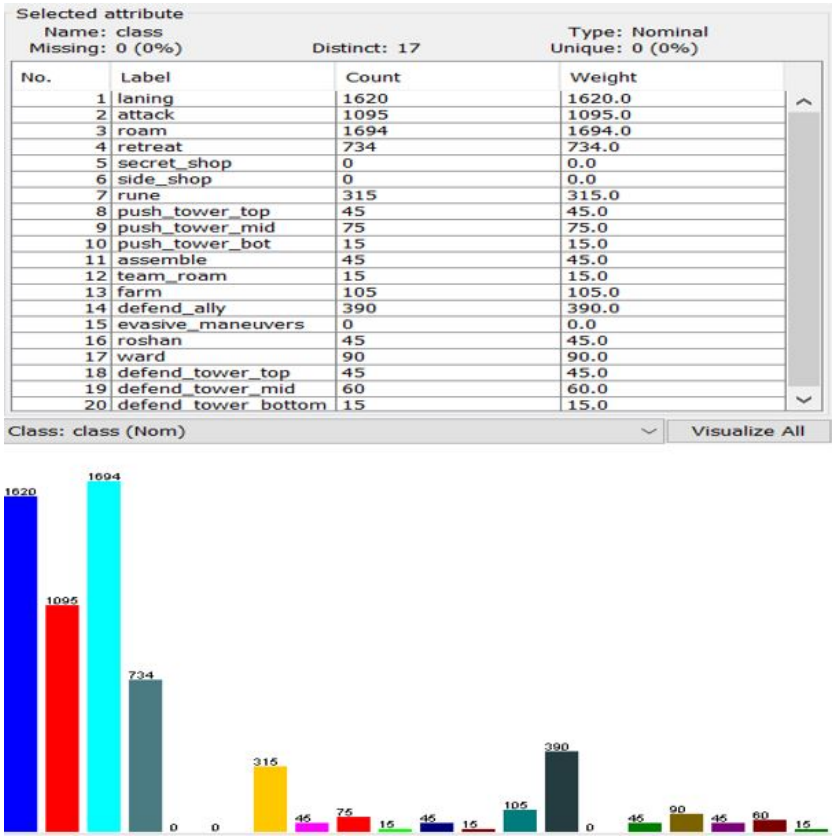
3.2.3 Training and Testing the Classifier

While searching for any ML softwares to be used for building, training and testing our classifier, we found Waikato Environment for Knowledge Analysis (WEKA), which contains a collection of algorithms

and data pre-processing tools, designed so that users can quickly try out any existing ML methods on new datasets, in very flexible ways. It provides extensive support for the whole process of experimental data mining, including preparing the input data, evaluating learning schemes statistically, and visualizing both the input data and the result of learning [40]. This was our software of choice, after reading about some of its benefits, and understanding how straight-forward the interface was.

First step was changing the previously obtained datasets to the file format that WEKA is familiar with: Attribute-Relation File Format (ARFF), an ASCII text file that describes a list of instances sharing a set of attributes, using *@DATA* and *@ATTRIBUTE* to declare them, respectively.

After this was performed, we were able to properly analyze our data through WEKA's interface, which details information about each individual feature, as well as the amount of each class contained in the dataset. This is helpful since we are able to visualize if there are any imbalanced classes and act accordingly.



No.	Label	Count	Weight
1	laning	1620	1620.0
2	attack	1095	1095.0
3	roam	1694	1694.0
4	retreat	734	734.0
5	secret_shop	0	0.0
6	side_shop	0	0.0
7	rune	315	315.0
8	push_tower_top	45	45.0
9	push_tower_mid	75	75.0
10	push_tower_bot	15	15.0
11	assemble	45	45.0
12	team_roam	15	15.0
13	farm	105	105.0
14	defend_ally	390	390.0
15	evasive_maneuvers	0	0.0
16	roshan	45	45.0
17	ward	90	90.0
18	defend_tower_top	45	45.0
19	defend_tower_mid	60	60.0
20	defend_tower_bottom	15	15.0

Figure 3.6: WEKA's Preprocessing tab, displaying the count of each class contained in the training dataset, from the 20 initial classes to label, in conformity with the Bot Scripting API strategies

Just with a quick analysis of figure 3.6 (which showcases the number of instances contained in our dataset, regarding each of the 20 classes), it is conclusive that there are class imbalances, with some classes having a very high number of instances while others are not even present in the dataset.

Although these issues are usually indicative of a worse model performance, initially we tried building the classifier having these class imbalances.

It is meaningful to add that there are many other data-preprocessing tools available, called “filters”, that can be applied to the data (either to attributes or instances), like discretization or re-sampling, which ended up being used to fix some of the main class imbalances, as will be explained later on.

WEKA also offered numerous classification and regression algorithms, available to be picked from, and that could be applied to our preprocessed data. Since we are trying to verify an hypothesis regarding the use of a Neural Network (NN) classifier to address this problem, our specific choice was the MLP, as it is the most used type of network in pattern classification problems. [32]

As explained in **section 2.4**, MLP is a type of feed-forward NN architecture, that can contain multiple layers of nodes, allowing its user to tune some important parameters: number of hidden layers, number of nodes, learning rate, epochs, among others. To have a better grasp of how the chosen classifier worked, it is important to further understand the architecture behind it, which can be defined, based on three following characteristics:

1. Network structure, defining the number of layers and neurons in each layer, as well as tuning any additional parameters

2. The learning mechanism applied for updating the weights of the connections

3. The activation functions used in hidden and output layers, dependent on the type of problem aimed to solve

The package we had initially chosen was one of WEKA’s defaults. Therefore, it barely contained any info regarding its specifications (omitting learning mechanisms and activation functions), only allowing the user to try out different network structures.

As so, different setups were tested, by only altering the number of hidden layers and amount of neurons per layer, leaving other perks as default. There is not a correct/defined number of neurons and layers that is optimal, since it is dependent on each case scenario, although there are some empirically-derived rules-of-thumb, presented by Panchal et. al [41]. Even after applying these, we still weren’t able to attain good results. There were 2 other main concerns with the NN package being used: the training time was very long (probably due to the algorithms used not being very efficient), and the network did not contain any new advancements used for ML, like *dropout*[6].

In search of better results, we decided to download an extra package from WEKA’s package manager, which contained a NN implementation less secretive about its specifics, when compared to the previous package used.

[6]Dropout brief definition. Online: https://machinelearningmastery.com/dropout-regularization-deep-learning-models-keras/

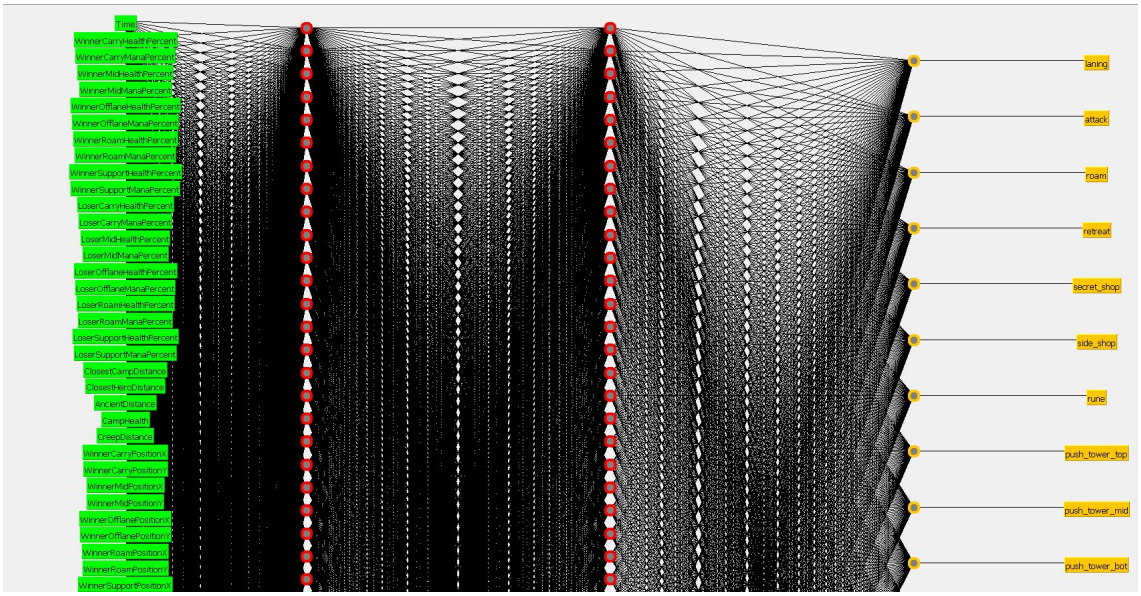


Figure 3.7: GUI of Multi-Layer Perceptron Architecture, with two hidden layers marked as red, attributes on the left, as green, and the output layer with output classes on the right, as yellow

Regarding its training method, this implementation uses *mini-batch gradient descent* (with a batch size of **100** training examples and **1000** training iterations), which is a variant from the *gradient descent* algorithm, treated nowadays as the dominant method used to train deep learning models. The goal of this algorithm is to find model parameters (weights, in NN architectures), updating those to minimize the error of the model in predictions made with the training data. This is performed by making changes to the model that move it along a gradient or slope of errors down towards a minimum error value, hence the name given to the algorithm. The *mini-batch gradient descent* is a variation of this algorithm, that splits the training dataset into smaller batches, used to calculate model error and update model weights. This specific method has some upsides when compared to the Stochastic Gradient Descent (SGD) and Batch Gradient Descent, which are other two variations of *gradient descent*[7]:

- It is computationally more efficient than the SGD method, which calculates the error and updates the model for every single example in the dataset, causing it to be more computationally expensive than other configurations

- Its update frequency is higher than the other variation (*batch gradient descent*), allowing for a more robust outcome, avoiding premature convergence of the model to a less optimal set of parameters

When it comes to activation functions used, this implementation uses ReLU as an activation function for the hidden layers and *Softmax* for the output layer. In fact, the *softmax* function used in our classifier is basically an extension of the *sigmoid* function (detailed in **section 2.4**), but used for multi-class domains,

[7] https://machinelearningmastery.com/gentle-introduction-mini-batch-gradient-descent-configure-batch-size/

where we have multiple outputs, creating a probability distribution that adds to 1. The aforementioned *vanishing gradient* problem is also one of the reasons we opted for this NN package, which had the ReLU implemented for hidden units. Overall, this specific activation function brings a good set of benefits when compared to the commonly used *sigmoid* function:

- It is less computationally expensive, because it involves simpler mathematical operations
- It was found to accelerate the convergence of gradient descent, resulting in faster training [42]
- It does not constrain the input space into a small region, avoiding the *vanishing gradient* problem

Regarding our NN structure, the final model contained 1 hidden layer with 50 nodes. This was decided by using the rule-of-thumb approach, presented by Panchal et al. [41], which is a method for determining the ideal number of neurons per hidden layer, for each case, stating that the number of hidden neurons should be in a range between the size of the input layer (70, in our case), and the size of the output layer (20 initially, reduced to 8, as will be explained next), and it should be about 2/3 the size of the input layer, plus the size of the output, hence ending up with 50 neurons. *Dropout regularization* was also applied to our network, with the goal of creating a model that has a better generalization and less likelihood of overfitting the training data. Dropout is a technique where randomly selected neurons are ignored during training. This is a useful concept: by switching some neurons off, it prevents them from becoming too specialized in the data they are training with, having other neurons to step in and handle the representation required to make predictions for the missing neurons, increasing adaptability. Our dropout rate was 50% in hidden layer and 20% in input layer (percentage of units subjected to dropout, in each layer), based on some useful pointers about proper dropout usage, available at [43].

It is important to not forget that, all of these implementation sub-sections were performed multiple times, altering their specifications iteratively, in order to attain a better model performance.

As aforementioned, 70% of our data was used to build a training set and the left out 30%, to serve as testing set. 10-fold cross-validation was performed on our training set, in order to flag problems like overfitting, and to give an insight on how the model would generalize to an independent dataset. Once this was done, the model was tested on our previously prepared independent dataset (containing data of different DotA2 matches from the ones used to extract the training data, hence representing completely different game situations). Our initial results were poor, but after analyzing some metrics displayed in WEKA's classifier output window, it was clear that class imbalances were affecting the development. Some classes were present in very few data-points of our sets, that the model simply wasn't able to learn them. In more extreme cases, some class categories only appeared on either the training or the testing datasets, so they were either not learned or never tested, which obviously skews the results. To tackle the issue, we decided to remove some classes and respective data-points, based on the following

criteria:

1. classes represented by less than 2% of the total data
2. classes that represent rare in-game situations, hence only present in one of all DotA2 match replays collected as data

After applying this method, 20 output classes were reduced to 8, with these representing the most frequent and common in-game strategies: *laning, attack, roam, retreat, rune, farm, ward* and *defend_ally*. This means that, if AI Agents were ever able to have, in regard to these strategies, a near similar decision-making to the highly-skilled gameplay gathered as data, that would turn out to be a considerable improvement in comparison to the performance of available default Bots.

Despite having hand-picked the "most promising" classes, there were still some imbalances, with half of the 8 classes representing a huge majority of the total data. To solve this issue, there are several different methods that can be applied to imbalanced datasets [44], such as:

1. *Undersampling* - aims to balance class distribution through the random elimination of the majority class examples/instances. Drawbacks from this method are that it can potentially discard useful data that could be important for the induction process, and also not ideal in our specific case, where would be useful to increase the data size
2. *Oversampling* - aims to balance class distribution through the random replication of the minority class examples/instances. A major drawback from this method is that it can increase the likelihood of occurring overfitting, since it makes exact copies of the minority class examples. In short terms, overfitting means that the predictive power of a classifier will be very good on a specific type of data (the training data), but the model will fail to fit additional data or predict future observations reliably
3. *Synthetic Minority Over-sampling* - aims to balance class distribution through the generation of synthetic minority examples to over-sample the minority class. This generation is performed by interpolating between several minority class examples that lie together. A great benefit of this method is that, since new instances generated are in fact different and not exact copies, this can help avoid the overfitting problem that occurs in the regular random oversampling.

After researching pros and cons of each, Synthetic Minority Over-sampling Technique (SMOTE) [45] seemed the right choice, since it was also available as one of many pre-processing tools from WEKA. In more detailed terms, when using SMOTE, the minority class is over-sampled by taking each minority class sample and introducing synthetic examples along the line segments joining any/all of the k minority class nearest neighbors. The amount of neighbors used for the synthetic generation of new samples will depend on the amount of over-sampling needed. In our case, from the 8 classes that ended up

being chosen, half were over-sampled by 300%, which means that, from the 5 nearest neighbors used in WEKA's SMOTE package implementation, only 3 were picked and new samples are generated in the direction of each. The method applied to generate more samples is quite simple: take the difference between the feature vector (sample) under consideration and its nearest neighbor. Multiply this difference by a random number between 0 and 1, and add it to the feature vector under consideration.

With this, we were able to tackle some of the main imbalances between all 8 classes, consequently improving the consistency of the results obtained, which are presented in the following section.

3.2.4 Accuracy Results and Evaluation

Our shell script, used to randomly split the data in training and testing datasets, was run 10 times in order to train/test the model in 10 different iterations, increasing the size and consistency of obtained results. WEKA comes in handy, since it displays a classifier output window with useful information about the performance summary of the ML model, presented in three phases:

1. *Classification accuracy* - the ratio of the number of correct predictions out of all predictions made, displayed as a percentage
2. *Accuracy by class* - contains performance metrics regarding each individual class, such as Precision, Recall, F-Measure, among others
3. *Confusion matrix* - table showing the number of predictions made for each class, compared to the number of instances that actually belong to each class. This is very useful to get an overview of the model's mistakes

Due to constraints explained in the following section, that restrained us from testing the classifier in a real-case scenario (using it on a real DotA2 match), consequently limiting our evaluation process, we relied on the following described metrics (which were obtained/calculated in each of 10 iterations), to theoretically evaluate how well would the model actually perform:

- *Classification accuracy*

  Accuracy is often the starting point for analyzing the quality of a predictive model, however it can sometimes be an unreliable metric, since it does not take into account possible dataset imbalances.

  As can be perceived from figure 3.8, this classifier presents an accuracy of roughly about **57%**, which is the percentage of instances that were associated with the correct class. The next term displayed is *Kappa statistic*, a metric that compares our observed accuracy with an expected accuracy (accuracy if classification was random). It can be calculated using the following formula:

$$Kappa = \frac{observedaccuracy - expectedaccuracy}{1 - expectedaccuracy} \tag{3.1}$$

```
=== Summary ===

Correctly Classified Instances        1553               57.1586 %
Incorrectly Classified Instances      1164               42.8414 %
Kappa statistic                          0.3911
Mean absolute error                      0.0568
Root mean squared error                  0.1678
Total Number of Instances             2717
```

Figure 3.8: Results obtained from evaluating the classifier in our 10th training/testing iteration

According to the scale used to interpret Kappa statistic's strength, created by Landis and Koch [46], our result, **0.3911**, is considered "fair", with any positive value below 0.2 considered "slight", and a negative result considered "poor". By making use of the formula, one can quickly realize that having a negative Kappa statistic value means that the observed accuracy is smaller than the excepted accuracy, resulting in a model performing worse than if it was randomly classifying instances.

The other values presented are the *Mean absolute error* and *Root mean squared error*, however these will not be considered for evaluation, since they are often used in regression problems. Instead, we will focus on other more reliable known metrics, commonly used for classification problems, that are going to be discussed shortly. Before addressing any other evaluation measures, it is important to understand that, although it seems a very strong metric, the global accuracy of one model can sometimes be misleading, due to a concept called *Accuracy Paradox* [47]. This concept states that predictive models with a given level of accuracy may have greater predictive power than models with higher accuracy. This is specially frequent when there are unbalanced datasets. For illustration, with a real-case scenario, a small study [48] was performed to explain the meaning behind this concept, which is available for consulting in the section **Appendix A**.

In short, this scenario teaches that accuracy as the only metric is insufficient, therefore the final decision in model selection should consider a combination of different performance measures instead of relying on one, an important lesson that was followed in our work.

Anyhow, here are the results obtained throughout all 10 iterations of training/testing the classifier:

Iteration nº	Accuracy(%)	Kappa Statistic
1	59.0725	0.4266
2	54.6485	0.3808
3	56.2084	0.3977
4	54.3145	0.3367
5	49.9628	0.3295
6	51.8903	0.3717
7	58.8934	0.4143
8	51.5584	0.3086
9	51.4830	0.3433
10	57.1586	0.3911
Avg.	**54.5190**	**0.37**

Table 3.1: Summary of classifier accuracy and kappa statistic results

- *True and False Positive Rates*

The following metrics are often used in binary classification problems, however they work just as fine in multi-class problems like ours, since they can ultimately be simplified to two-class problems using the one against all approach [49]. By convention, in binary classification, the class label of the minority class is called positive while the class label of the majority class is called negative. Using the mentioned approach, with multiple classes, we compare each one individually against the rest, with the class being compared called positive and all other classes called negative. Rates are calculated using 4 essential concepts: *True Positive (TP)*, which is the number of positive cases correctly classified as such; *False Negative (FN)*, the number of positive cases incorrectly classified as negatives; *False Positive (FP)*, the number of negative cases that are incorrectly identified as positive cases and *True Negative (TN)*, the number of negative cases correctly classified as such.

Having the values for each of these 4 concepts, TP and FP Rates can then be calculated using the following formulas:

$$FPrate = \frac{FP}{FP + TN} \tag{3.2}$$

$$TPrate = \frac{TP}{TP + FN} \tag{3.3}$$

From these, it is quickly deductible the meaning of each rate: *FP rate* is the ratio between the number of negative events wrongly categorized as positive (false positives) and the total number of actual negative events, while *TP rate* measures the proportion of actual positive examples that are correctly identified as such. The latter is also commonly mentioned as *Sensitivity* or *Recall*.

We obtained the following results for each iteration of training and testing our model:

Iteration nº	TP Rate	FP Rate
1	0.591	0.164
2	0.546	0.165
3	0.562	0.161
4	0.543	0.209
5	0.500	0.143
6	0.519	0.132
7	0.589	0.169
8	0.516	0.212
9	0.515	0.174
10	0.572	0.188
Avg.	**0,545**	**0.172**

Table 3.2: Summary of true and false positive rates weighted averages results

- *Precision*

  Commonly used in these type of classification problems, it is a measure of a model's exactness. A high precision value is usually an indication of a good classifier. It can be calculated using the following formula:

$$Precision = \frac{TP}{TP + FP} \tag{3.4}$$

  Precision and *Recall* are often used together to fully evaluate the effectiveness of a model: high precision means that most examples classified as positive were indeed positive, while high recall means that most positive examples were correctly classified as so. Results regarding this metric are displayed in table 3.3.

Iteration nº	Precision
1	0.592
2	0.577
3	0.552
4	0.521
5	0.606
6	0.582
7	0.616
8	0.477
9	0.503
10	0.556
Avg.	**0.558**

Table 3.3: Summary of weighted average precision values, obtained throughout all iterations

- *F-Measure*

Often called *F-Score*, it measures a test's accuracy, considering both previous metrics *Precision* and *Recall* to compute the score, using the following mathematical expression:

$$Fmeasure = \frac{(1+\beta^2)Recall \times Precision}{\beta^2 Recall + Precision} \tag{3.5}$$

Where β corresponds to the relative importance of recall versus precision, and is usually set to 1 in problems where both equally matter. Adopting the generic approach to our problem (β = 1), the formula can be simplified, as follows:

$$Fmeasure = \frac{2 \times Recall \times Precision}{Recall + Precision} \tag{3.6}$$

A common misconception is that the standard *classification accuracy* and *f-measure* are basically the same, however, knowing now the meaning behind some concepts, we can express the previously mentioned accuracy in the following mathematical terms:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.7}$$

Accuracy can be predominantly contributed by a large number of TN examples which, in most circumstances, are not nearly as important as FN and FP. For example, in the already mentioned fraudulent credit card classifier, we wouldn't care for TN at all. However, if presented high numbers of FN and FP (fraudulent credit cards classified as genuine or genuine credit cards classified as fraudulent), it would probably result in a costly problem, if assuming a real-world scenario.

For this reason, *F-measure* is usually a better metric to use, if we are seeking a balance between Precision and Recall and there is an uneven class distribution. Results are displayed in table 3.4.

Iteration nº	**F-Measure**
1	0.581
2	0.543
3	0.544
4	0.519
5	0.505
6	0.533
7	0.588
8	0.487
9	0.500
10	0.547
Avg.	**0.535**

Table 3.4: Summary of weighted average F-measure values, obtained throughout all iterations

- *Matthew's Correlation Coefficient*

  Yet another measure commonly applied to binary classifications, Matthew's Correlation Coefficient (MCC) can also be applied to multi-class domains, using the previously mentioned one-vs-all approach, and then calculating the average of each class MCC value. It can be summarized with the following formula:

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{3.8}$$

  The result is in essence a correlation coefficient between the observed and predicted classifications, returning a value between -1 and +1, with +1 representing a perfect prediction, 0 as no better than random prediction and -1 the worst possible prediction.

  We also decided to utilize MCC to evaluate our model, since its value is least influenced by unbalanced data, serving as a balanced measure which can be used even if the classes are of very different sizes [50].

Iteration nº	**MCC**
1	0.441
2	0.403
3	0.412
4	0.344
5	0.368
6	0.391
7	0.443
8	0.304
9	0.349
10	0.390
Avg.	**0.385**

Table 3.5: Summary of weighted average MCC values, obtained throughout all iterations

- *Area Under the ROC Curve*

  One of the most common metrics used is the Receiver Operating Characteristic (ROC) curve analysis, and the associated use of the Area Under the ROC Curve (AUC), to assess overall classification performance. The ROC curve is represented by a graph showing the performance of a model at all classification thresholds. This curve plots two already mentioned parameters: FP rate on the x-axis and TP rate on the y-axis. Therefore, with the meaning of each of these concepts in mind (explained in equations 3.2 and 3.3), the point (0,1) is ideal for learners, representing a perfect classification, while a random guess would give a point along a diagonal line from the left bottom to the top right corners. In sum, it is desirable to have a plot as close to the vertical axis

and as high as possible.

Again, in multi-class problems like ours, we use the *one vs. all* approach to draw ROC curves for each class, which are available for visualization in WEKA:

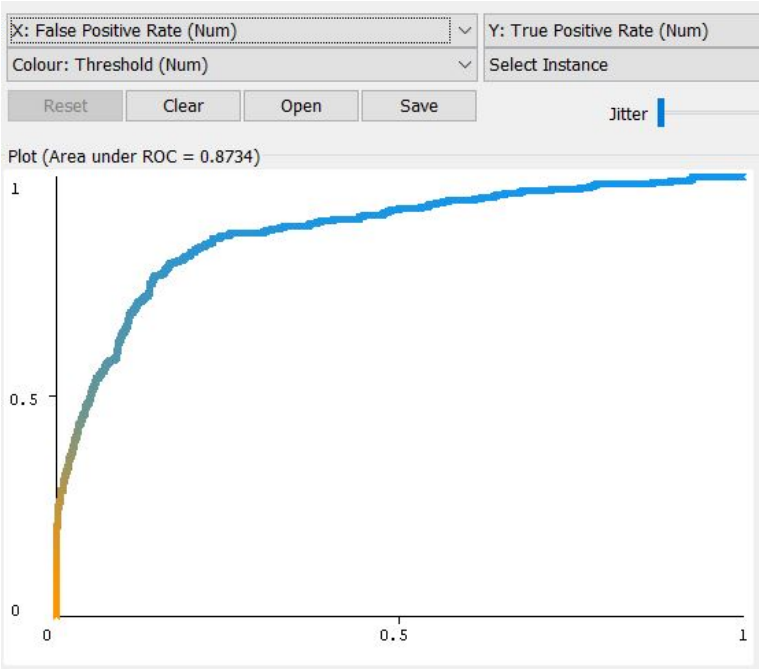


Figure 3.9: Example of a ROC curve of class *Retreat*, using the one class vs. all approach

As can be seen, WEKA also displays the AUC value, which basically measures the entire two-dimensional area underneath ROC curve. In more specific terms, AUC is an aggregate measure of performance across all possible classification thresholds, interpreted as the probability that the model ranks a random positive example higher than a random negative example[8]. Its value varies between 0 and 1, with an area of 1 representing a perfect model and an area of 0.5 representing a worthless model [48].

WEKA provides a weighted average of all AUC values of each class, in which we obtained the following results, presented in table 3.6:

After having individually analyzed all of the previously detailed evaluation metrics, it can be stated that, although it presents far from perfect results, our classifier is still considerably better than any random classifications, and we can only speculate that it would improve DotA2 default agents' decision-making. All results obtained can be summarized in table 3.7, which hopefully serves as a viable and realistic overall measurement of our classifier's performance, on average:

[8]Online: https://developers.google.com/machine-learning/crash-course/classification/roc-and-auc

Iteration nº	Area Under ROC Curve
1	0.812
2	0.787
3	0.822
4	0.741
5	0.807
6	0.769
7	0.807
8	0.736
9	0.763
10	0.807
Avg.	**0.785**

Table 3.6: Summary of weighted average AUC values, for each iteration

Iteration	Accuracy(%)	Kappa Statistic	TP Rate	FP Rate	Precision	F-Measure	MCC	AUC
1	59.0725	0.4266	0.591	0.164	0.592	0.581	0.441	0.812
2	54.6485	0.3808	0.546	0.165	0.577	0.543	0.403	0.787
3	56.2084	0.3977	0.562	0.161	0.552	0.544	0.412	0.822
4	54.3145	0.3367	0.543	0.209	0.521	0.519	0.344	0.741
5	49.9628	0.3295	0.500	0.143	0.606	0.505	0.368	0.807
6	51.8903	0.3717	0.519	0.132	0.582	0.533	0.391	0.769
7	58.8934	0.4143	0.589	0.169	0.616	0.588	0.443	0.807
8	51.5584	0.3086	0.516	0.212	0.477	0.487	0.304	0.736
9	51.4830	0.3433	0.515	0.174	0.503	0.500	0.349	0.763
10	57.1586	0.3911	0.572	0.188	0.556	0.547	0.390	0.807
Avg.	**54.5190**	**0.37**	**0.545**	**0.172**	**0.558**	**0.535**	**0.385**	**0.785**

Table 3.7: Summary of all classifier evaluation metrics

3.3 Limitations

The two main limitations to our solution were related with: output issues in WEKA (software used to experiment ML algorithms, to train and test our classifier), and the lack/inability of proper integration of ML models with DotA2 Bot Scripting API.

After having used WEKA to build our classifier and having obtained proper accuracy results, we found out that the file format in which our model was being saved was native to the software, hence only working inside the environment. The only way WEKA had of exporting models, was an option that would generate the java code in conformity to the model used, but even this was only available for models built with certain ML algorithms. As so, the option to generate code from our model wasn't available, and we required another way of building and integrating our model with the DotA2 API.

It is important to understand that all the training and testing performed in WEKA was not wasted time, because despite not being able to use our model directly, we already had a fine-tunned NN model, datasets, and respective accuracy results. All that was needed was re-creating the model, with the same

algorithms and perks used, in an environment that would actually allow us to save models and export them.

The DotA2 Bot Scripting API had its default Bot files coded in Lua Programming Language, so we searched for any Lua libraries/frameworks with support for ML, and found Torch[9]:*"At the heart of Torch are the popular neural network and optimization libraries which are simple to use, while having maximum flexibility in implementing complex neural network topologies."*. We decided to test it out by writing some Lua code, using Torch's NN libraries, to build a simple model, training and testing it, using the popular *Iris flower dataset*[10], which has been used for years, as data for classification problems [51].

The results obtained from training and testing this new model were great, and saving/loading the model to any other regular .lua file was also tested and worked as well. All that was left was loading the model in any Bot file from the DotA2 API, and trying to run a Bot match in-game, just to check if everything was working as expected. Unfortunately, when running everything in-game, our model wasn't recognized by the system, showing the following error in the game console, presented in figure 3.10.

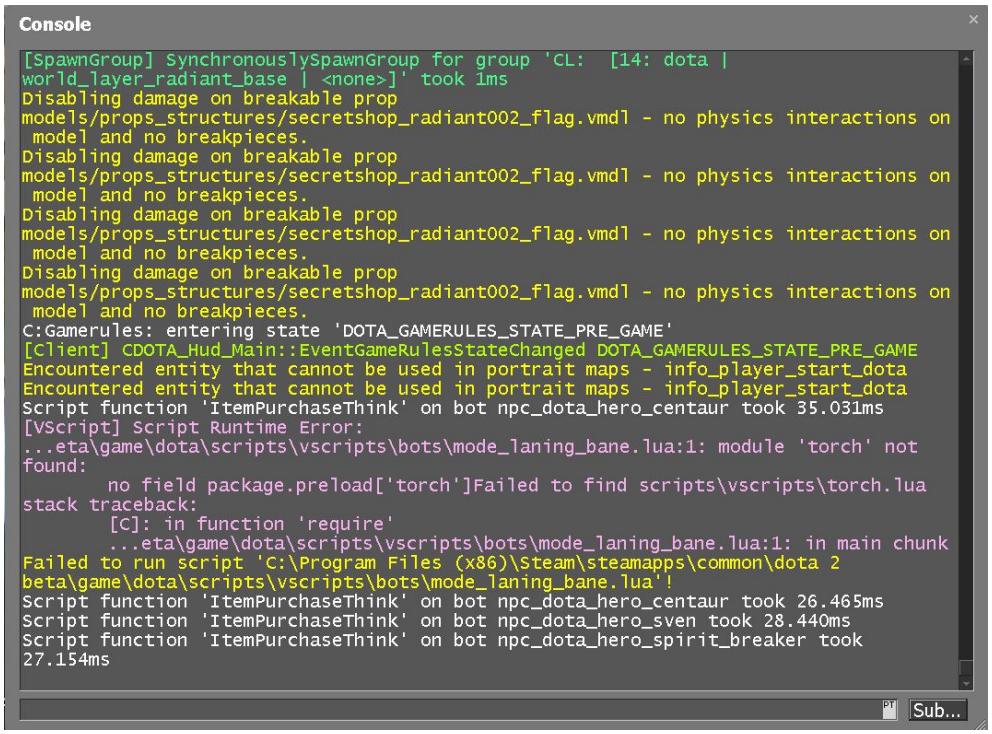


Figure 3.10: DotA2 game console, showcasing the model importing error, highlighted in purple color

After searching for a fix to this problem, we still didn't find a proper answer, maybe due to the lack of existing works related with the application of ML in DotA2, making ours somewhat a pioneer project, and ended up not being able to solve the issue, which deprived us from properly testing our solution in a real-

[9]Framework with wide support for machine learning algorithms. Online: http://torch.ch/
[10]Online: https://en.wikipedia.org/wiki/Iris_flower_data_set

case scenario, and obtaining respective results, which would be interesting to have for the evaluation section.

4

Conclusion

Contents

4.1 Main Contributions

The goal of this work was to develop a supervised learning model that would have better control over which are the ideal strategies to be employed by AI Bots, during a DotA2 match. In this document we covered the implementation process of our solution, and the accuracy results obtained from testing our SL classifier. By analyzing these results, available on **section 3.2.4**, we can verify that our hypothesis, presented in **section 1.3**, which declares that *by building an ANN classifier, we are able to identify the correct strategies employed during early stages of a match*, was successfully proven.

Overall, three important contributions of this thesis can be highlighted:

1. A java code that receives any DotA2 replay as input, which extracts relevant features, useful for strategic decision-making, with these being processed for each second of the match. Hopefully this program will be useful for any future works regarding the topic of ML in DotA2.

2. A trained and tested NN model, which had decent evaluation results, and would, in theory, significantly improve the strategic decision-making of the available default DotA2 Bots, had it been successfully integrated with the Scripting API.

3. This document also pointed out existing issues that have a negative impact in the development of ML systems for DotA2 and that are hopefully tackled and dealt with in future works, like the limitations in integrating models with the DotA2 Bot Scripting API, and not having any available methods (to our knowledge) of automatic-labeling for DotA2 replays.

4.2 Future Work

We decided to shift our work to focus only on the *support role* decision-making, to learn its common behaviors and the correct decisions to apply at different in-game situations. Although we had started with the idea of building 5 classifiers, one for each in-game role, this specific role ended up being chosen since it was arguably the richest and most complex, due to having the biggest amount of interchanging between strategies in early phases of each game match.

As so, and due to time constraints, many experiments and tests regarding the other 4 roles have been left for the future:

1. Properly building other 4 classifiers, which would require the extraction of different features, regarding each role specifications.

2. In contrast to the *support role*, some others that were left out, present richer decision-making patterns in later stages of the game, which indicates that, more interesting results would be obtained if replays were to be labeled for the entirety of each match duration. This would allow each role

to an equal chance of presenting complex behaviors, also improving our solution, which would ultimately be able to dictate decisions for Bots to employ, during the entirety of any match.

3. Regarding our dataset, its augmentation by labeling more matches would probably result in more accurate and consistent results, since it would increase the diversity of in-game situations. This can be easily achievable in the future to improve the work, since the task itself is not hard, just time-consuming. Ideally, a future work that would greatly help, would be coming up with a method to allow automatic-labeling of DotA2 replay matches, although this seems very hard to accomplish, due to the complexity and amount of variables present in MOBA environments.

It would also be of major importance solving the integration issues in importing ML models to the DotA2 Bot Scripting API, in future works. This specific limitation didn't allow us to fully test our developed work in a real-time scenario, which would contribute with significant results. However, it is not unthinkable to state that, would this issue ever be fixed in the future, our work here developed would be ready to be fully applied and we would be able to verify, in practical terms, which outcome would be obtained from an arranged DotA2 Bot match, using our ML model.

4.3 Lessons Learned

During early phases, like defining our work plan and steps, our time schedule, and deciding exactly what this thesis would consist of, my supervisors warned me about the importance of verifying if the whole "development cycle" that we planned to perform, would actually be possible to achieve. The idea behind this, was to test all the steps that we set out to do, but in a simplified way, just to make sure that they were feasible, and that everything would work out in later stages, where we would apply our actual project settings. Although I tried to verify this, as told, my mistake was only experimenting that the DotA2 Bot Scripting API was working fine on its own and that it allowed for agent's to interchange between strategies, by hand-editing in their respective strategy-files, the values of each strategy desire. This proved to be a wrong decision, since I never ended up testing if the API would be able to read out imported ML models and use them to affect its bots' decision-making. Although this oversight could have possibly been avoided, it was still important for my experience and growth, as a future professional in this area, and taught me how crucial is to make sure that the development cycle of any project is doable, before starting any kind of implementation.

Other aspect learned with the development of this thesis, was the importance of having a well-planned project before-hand, and gathering as much information as possible about any softwares/tools that are going to be used, since learning their constraints or fully understanding their strengths in advance, will allow for big time-saves, during the implementation phase. As an example, although some prior research was made about WEKA (the software that was used for building our classifier), we would

probably end up not using it and relying on a different software instead, if we had known before-hand of all the existing issues and inabilities with exporting models to other environments.

To conclude, from a theoretical and technical point of view, this work surely increased my knowledge about general topics of ML usage, and its application in games, more specifically about the capabilities of neural networks, their advantages and disadvantages in comparison to other methods, as well as learning which types of ANN are ideal to use, in conformity to the type of problem being tackled.

Bibliography

[1] N. A. Barriga, M. Stanescu, and M. Buro, "Combining Strategic Learning and Tactical Search in Real-Time Strategy Games," in *Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 2017. [Online]. Available: http://arxiv.org/abs/1709.03480

[2] M. Mateas and C. Street, "Expressive AI : Games and Artificial Intelligence," in *Proceedings of Level Up - Digital Games Research Conference*, 2003, pp. 1–9.

[3] K. Stuart, "Those classic Atari games were harder than you think," p. 1, 2015. [Online]. Available: https://www.theguardian.com/technology/2015/feb/26/those-classic-atari-games-were-harder-than-you-think

[4] B. Wu, "The Computational Intelligence of the Game Pac-Man," *Springer, Berlin, Heidelberg*, vol. 312, pp. 646–647, 2012.

[5] A. J. Champandard, "Top 10 Most Influential AI Games," p. 1, 2007. [Online]. Available: http://aigamedev.com/open/highlights/top-ai-games/

[6] ——, "The AI from Half-Life's SDK in Retrospective," p. 1, 2008. [Online]. Available: http://aigamedev.com/open/article/halflife-sdk/

[7] T. Leonard, "Building an AI Sensory System: Examining the Design of Thief: The Dark Project," p. 2, 2003. [Online]. Available: https://www.gamasutra.com/view/feature/131297/building{_}an{_}ai{_}sensory{_}system{_}.php

[8] Y. Bourse, "Artificial Intelligence in The Sims series," p. 6, 2012. [Online]. Available: http://www.yoannbourse.com/ressources/docs/ens/sims-rapport.pdf

[9] A. Arsenault, "Have Fun Storming the Castle! Siege Battle AI in Total War: Warhammer," in *Game Developers Conference*, 2016, p. 76.

[10] M. Mateas and A. Stern, "Integrating Plot, Character and Natural Language Processing in the Interactive Drama Façade," *Proceedings of the Technologies for Interactive Digital Storytelling and Entertainment (TIDSE) Conference*, vol. 2, pp. 139–151, 2003.

[11] C. Fairclough, M. Fagan, B. Mac Namee, and P. Cunningham, "Research Directions for AI in Computer Games," in *Irish Conference on Artificial Intelligence and Cognitive Science*, 2001, pp. 333–344. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.17.2579{&}rep=rep1{&}type=pdf

[12] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016. [Online]. Available: http://dx.doi.org/10.1038/nature16961

[13] N. Heess, D. TB, S. Sriram, J. Lemmon, J. Merel, G. Wayne, Y. Tassa, T. Erez, Z. Wang, S. M. A. Eslami, M. Riedmiller, and D. Silver, "Emergence of Locomotion Behaviours in Rich Environments," 2017. [Online]. Available: http://arxiv.org/abs/1707.02286

[14] S. Ontañón, G. Synnaeve, A. Uriarte, F. Richoux, M. Preuss, and D. Churchill, "A Survey of Real-Time Strategy Game AI Research and Competition in StartCraft," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 5, no. 4, pp. 293–311, 2013. [Online]. Available: http://webdocs.cs.ualberta.ca/{%}7B{~}{%}7Dcdavid/pdf/starcraft{_}survey.pdf

[15] A. R. Tavares, H. Azpúrua, and L. Chaimowicz, "Evolving Swarm Intelligence for Task Allocation in a Real Time Strategy Game," in *Brazilian Symposium on Games and Digital Entertainment, SBGAMES*, vol. 2014-Decem, no. December, 2014, pp. 99–108.

[16] M. Santos and C. Martinho, "Wasp-like agents for scheduling production in real-time strategy games," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7026 LNAI, pp. 71–82, 2011.

[17] A. A. Cunha, "Swarm Intelligence in Strategy Games," Master's, Instituto Superior Técnico, 2015.

[18] S. Ontañón, K. Mishra, N. Sugandh, and A. Ram, "Case-based planning and execution for real-time strategy games," in *Proceedings of ICCBR*, no. Iccbr, 2007, pp. 164–178. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-540-74141-1{_}12

[19] K. Mishra, S. Ontañón, and A. Ram, "Situation assessment for plan retrieval in real-time strategy games," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5239 LNAI, no. Eccbr, pp. 355–369, 2008.

[20] B. G. Weber and M. Mateas, "A data mining approach to strategy prediction," *CIG2009 - 2009 IEEE Symposium on Computational Intelligence and Games*, pp. 140–147, 2009.

[21] L. Gao, J. Judd, D. Wong, and J. Lowder, “Classifying dota 2 hero characters based on play style and performance,” *Univ. of Utah Course on ML*, 2013.

[22] C. Eggert, M. Herrlich, J. Smeddinck, and R. Malaka, “Classification of Player Roles in the Team-Based Multi-player Game Dota 2,” in *Proceedings of Entertainment Computing - ICEC 2015*, vol. 9353. Springer International Publishing, 2015, pp. 112–125. [Online]. Available: http://link.springer.com/10.1007/978-3-662-45212-7

[23] F. Johansson and J. Wikström, “Result Prediction by Mining Replays in {DotA} 2,” Ph.D. dissertation, Blekinge Institute of Technology, 2015.

[24] P. Yang, B. Harrison, and D. L. Roberts, “Identifying Patterns in Combat that are Predictive of Success in MOBA Games,” in *Proceedings of Foundations of Digital Games 2014*, 2014, pp. 1–8.

[25] M. Schubert, A. Drachen, and T. Mahlmann, “Esports Analytics Through Encounter Detection,” in *MIT SLOAN Sports Analytics Conference*, 2016, pp. 1 – 18. [Online]. Available: http://www.sloansportsconference.com/?p=17536

[26] A. Drachen, M. Yancey, J. Maguire, D. Chu, I. Y. Wang, T. Mahlmann, M. Schubert, and D. Klabjan, “Skill-Based Differences in Spatio-Temporal Team Behavior in Defence of The Ancients 2 (DotA 2),” in *Proceedings of Games Media Entertainment*, no. DotA 2, 2014, pp. 1–8.

[27] T. Mahlmann, “Dotalys 2,” p. 1, 2014. [Online]. Available: http://www.lighti.de/projects/dotalys-2/

[28] A. Drachen and M. Schubert, “Spatial game analytics and visualization,” in *IEEE Conference on Computatonal Intelligence and Games, CIG*, 2013.

[29] T. E. Batsford, “Calculating Optimal Jungling Routes in DOTA2 Using Neural Networks and Genetic Algorithms,” 2014.

[30] E. Y. Li, “Artificial neural networks and their business applications,” *Information & Management*, vol. 27, no. 5, pp. 303–313, 1994.

[31] P. Sibi, S. Jones, and P. Siddarth, “Analysis of different activation functions using back propagation neural networks,” *Journal of theoretical and applied Information Technology*, vol. 47, no. 3, pp. 1264–1268, 2013.

[32] J. K. Basu, D. Bhattacharyya, and T.-h. Kim, “Use of Artificial Neural Network in Pattern Recognition,” *International Journal of Software Engineering and its Applications*, vol. 4, no. 2, pp. 23–34, 2010.

[33] M. Zare, H. R. Pourghasemi, M. Vafakhah, and B. Pradhan, "Landslide susceptibility mapping at Vaz Watershed (Iran) using an artificial neural network model: A comparison between multilayer perceptron (MLP) and radial basic function (RBF) algorithms," *Arabian Journal of Geosciences*, vol. 6, no. 8, pp. 2873–2888, 2013.

[34] J. Li, J.-h. Cheng, J.-y. Shi, and F. Huang, "Brief Introduction of Back Propagation (BP) Neural Description of BP Algorithm in Mathematics," *Advances in Computer Science and Information Engineering*, vol. 2, pp. 553–558, 2012.

[35] P. Venkatesan and S. Anitha, "Application of a radial basis function neural network for diagnosis of diabetes mellitus." *Current Science (00113891)*, vol. 91, no. 9, pp. 1195 – 1199, 2006.

[36] V. Sharma, S. Rai, and A. Dev, "A Comprehensive Study of Artificial Neural Networks," *International Journal of Advanced Research in Computer Science and Software Engineering*, vol. 2, no. 10, pp. 278–284, 2012. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.468.9353{&}rep=rep1{&}type=pdf

[37] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *Advances In Neural Information Processing Systems*, pp. 1–9, 2012.

[38] Z. C. Lipton, J. Berkowitz, and C. Elkan, "A Critical Review of Recurrent Neural Networks for Sequence Learning," pp. 1–38, 2015. [Online]. Available: http://arxiv.org/abs/1506.00019

[39] J. Tuckova, M. Bartu, P. Zetocha, and P. Grill, "Self-Organizing Maps As Data Classifiers in Medical Applications," *Ncta 2011: Proceedings of the International Conference on Neural Computation Theory and Applications*, no. December, pp. 422–429, 2011.

[40] E. Frank, M. Hall, G. Holmes, R. Kirkby, and I. H. Witten, "Chapter 1 WEKA A Machine Learning Workbench for Data Mining," *Proceedings of the Second Australia and New Zealand Conference on Intelligent Information Systems*, pp. 1–10, 1994.

[41] F. S. Panchal and M. Panchal, "Review on Methods of Selecting Number of Hidden Nodes in Artificial Neural Network," *International Journal of Computer Science and Mobile Computing*, vol. 311, no. 11, pp. 455–464, 2014. [Online]. Available: https://www.ijcsmc.com/docs/papers/November2014/V3I11201499a19.pdf

[42] A. Krizhevsky, I. Sutskever, and H. Geoffrey E., "ImageNet Classification with Deep Convolutional Neural Networks," *Advances in Neural Information Processing Systems 25 (NIPS2012)*, pp. 1–9, 2012. [Online]. Available: https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[43] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting,” *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.

[44] S. Kotsiantis, D. Kanellopoulos, and P. Pintelas, “Handling imbalanced datasets : A review,” *Science*, vol. 30, no. 1, pp. 25–36, 2006. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.96.9248{&}rep=rep1{&}type=pdf

[45] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, “SMOTE: Synthetic minority over-sampling technique,” *Journal of Artificial Intelligence Research*, vol. 16, pp. 321–357, 2002.

[46] J. R. Landis and G. G. Koch, “The Measurement of Observer Agreement for Categorical Data,” *Biometrics*, vol. 33, no. 1, p. 159, 1977. [Online]. Available: http://www.jstor.org/stable/2529310?origin=crossref

[47] F. J. Valverde-Albacete and C. Peláez-Moreno, “100% classification accuracy considered harmful: The normalized information transfer factor explains the accuracy paradox,” *PLoS ONE*, vol. 9, no. 1, 2014.

[48] J. S. Akosa, “Predictive Accuracy : A Misleading Performance Measure for Highly Imbalanced Data,” *SAS Global Forum*, vol. 942, pp. 1–12, 2017.

[49] J. Milgram, M. Cheriet, and R. Sabourin, ““One Against One” or “One Against All”: Which One is Better for Handwriting Recognition with SVMs?” *Tenth International Workshop on Frontiers in Handwriting Recognition*, pp. 1–6, 2006. [Online]. Available: http://hal.inria.fr/inria-00103955

[50] S. Boughorbel, F. Jarray, and M. El-Anbari, “Optimal classifier for imbalanced data using Matthews Correlation Coefficient metric,” *PLoS ONE*, vol. 12, no. 6, pp. 1–17, 2017.

[51] M. Swain, S. K. Dash, S. Dash, and A. Mohapatra, “An approach for Iris plant classification Using neural Network,” *International Journal on Soft Computing (IJSC)*, vol. 3, no. 1, pp. 79–89, 2012.

A

Accuracy Paradox Case-Scenario

Assume two classification models with the goal of detecting fraudulent credit cards were built. From the testing data gathered, there were 990 examples of genuine and only 10 fraudulent credit cards. Each model performed as follows:

	Classified positive	Classified negative
Actual positive	0	10
Actual negative	0	990

Table 1. Confusion matrix for classifier 1 in illustrative example

	Classified positive	Classified negative
Actual positive	6	10
Actual negative	4	980

Table 2. Confusion matrix for classifier 2 in illustrative example

A brief analysis will show an accuracy of 99% in the first model and slightly less, 98.6% in the second. If accuracy was the only metric accounted for, we would wrongly conclude that the first model

performed better. Despite having a good accuracy value, the classes are extremely unbalanced, and the first classifier opted to ignore the minority examples (the important ones, fraudulent credit cards) and classified every example as the majority class. This can have disastrous consequences, since the initial purpose was to flag these fraudulent situations, and in the first situation those are being neglected. Although the second classifier has lower accuracy value, it clearly performs better since it was able to flag 6 out of 10 minority examples.