University of Liège

Faculty of Engineering
Montefiore Institute

Learning Artificial Intelligence in Large-Scale Video Games

— A First Case Study with Hearthstone: Heroes of WarCraft —

Master Thesis Submitted for the Degree of
MSc in Computer Science & Engineering

Author:
David Taralla

Supervisor:
Prof. Damien Ernst

Academic year 2014 – 2015

Acknowledgements

My first thanks go to Prof. Damien Ernst who let me work on a subject I am passionate about.

I wish also to thank Aaron Qiu, PhD student and researcher at the University of Liège and as today on the way to get the title of *Legend* in *Hearthstone*. He is the expert we mention several times in this thesis, and without his knowledge on the game, the autonomous agent we designed would not have the quality reached so far. As a side note and to insist on his expert role, the agent was first named *Aaron*. However, as we kept seeing it as a "she", it (or she) finally was named *Nora*, which is nothing but... *Aaron* in reverse.

Of course, I acknowledge the interest of my friends and family, even when most of them did not understand what my thesis was about. Therefore, a big thanks to all of you, because you were my day-to-day supporters.

Last but definitely not least, I would like to thank Dr. Raphaël Fonteneau who always supported me these past few months, giving me scientific advice and reviews of great quality. Always present to answer my numerous interrogations, constantly looking to give me tracks to follow and ensuring they interested me, it was a real pleasure to work with him and I hope our paths will cross again, in an academic context or another.

Contents

Chapter 1

Introduction

1.1 Video Games

1.1.1 History

From the early maze games (like *Pac-Man*, Namco, 1980) and platform games (like *Donkey Kong*, Nintendo, 1981), today's video games have come a long way and feature a rich history.

At their beginning, there were mostly simple arcade games where the challenge relied on the players' capability to react quickly in order to win.

Then, with the emergence of better storage supports like the CD-ROM, followed by the DVD and now the Blu-Ray Disc and dematerialization, video games became more and more complex, featuring rich 3D environments and deep interactions between the player and the game. Video games became more and more realistic from a graphical point of view, and the challenges they brought involve now the simultaneous consideration of many variables by the players.

1.1.2 Artificial Intelligence in Video Games

Many if not all games feature a mode where the player can play against an artificial intelligence (AI). Back in the eighties with *Pac-Man* for instance, the avatar was already chased by little ghosts. Building an AI for them was not very complicated since their restrained environment and goal. However, Safadi et al. [2015] underlined the fact that as games continue to feature richer and more complex environments, designing robust AI becomes increasingly difficult.

Still today, many game agents are built in a scripted way, meaning that game scenarios, how to recognize and react to them are hard-coded. This old-fashioned approach is no longer relevant for modern games, as it is often too complicated to be applied to them regarding the time and resources developers are given for developing a game. Unfortunately, many video game companies continue to stick to it, and players are left empty handed when it comes to measure themselves against a challenging autonomous agent.

Indeed, the common way to solve the problem of the scripted approach – too overwhelming to be applied – is to reduce the number of strategies an autonomous agent can exhibit. Moreover, as every game state can not be considered in advance, numerous variables specific to the game have to be discarded, for the quantity of predefined situations a scripted agent can recognize to be tractable. Altogether, these shortcuts make the designed agent redundant in its decisions, and the average player is quickly able to recognize the agent strategies when it plays, removing all the fun and challenge the player could get being surprised by the moves of the AI.

Fortunately, with the advent of the machine learning era, we might be at the dawn of finding techniques where the AI in games is no longer coded by humans, but rather has *learned* to play the game as any human player would. This modern approach is nevertheless not without challenges, as for example large-scale video games feature complex relations between the entities they manage, leading to difficulties for simulating side-trajectories in the course of a game.

1.2 Goals

The goal of this work is twofold. It first aims to develop a prototype of theory for the generic design of autonomous agents for large-scale video games, the main contribution being a formalization of video games and the problem to solve when creating an AI for one of them.

Classification of actions based on supervised learning is attempted in order to come up with a prototype of autonomous agent playing the game *Hearthstone: Heroes of WarCraft*, without being able to train it using simulations of side-trajectories in the course of a game, nor knowing in advance what consequences any possible action has on the game. Indeed, *Hearthstone: Heroes of WarCraft* presents so many dependencies amongst all its elements that it would be intractable to implement some *undo* mechanic making an algorithm able to backtrack once a trajectory was simulated from a given state. Therefore, simulation-based approaches like Monte-Carlo Tree Search (MCTS) [Chaslot et al., 2006; Coquelin and Munos, 2007; Coulom, 2007; Gelly and Wang, 2006] are out of scope here, as they need to be able to simulate trajectories. Nevertheless, Section 6.2.3 presents possibly promising research tracks for applying MCTS to large-scale video games.

The second objective is to develop a modular and extensible clone of the game *Hearthstone: Heroes of WarCraft*, such that practitioners from all over the world can have a new benchmarking tool to test their algorithms against.

1.3 Related Work

1.3.1 Research on Video Games

For several years now, the video game field has been the center of many research works. This research is motivated by the development of new technologies to increase entertainment, but also by the fact that video games present themselves as alternate, low-cost yet rich environments to assess the performance of machine learning algorithms.

As mentioned by Gemine et al. [2012], one or both of two main objectives are usually pursued in video game AI research.

The first one is to create agents that exhibit properties making them funnier to play with. Players are usually frustrated when the AI rivals them thanks to unfair advantages, like extremely short response times, perfect aim or even cheating. Usually, this kind of work occurs for games that already feature agents (too) challenging for skilled human players.

The other goal is about complex games for which challenging agents do not exist yet; MOBAs[1] like *League of Legends* (Riot Games, 2009) are a good example of such games: the agents exhibit poor threat assessment and ignore threats when returning to their base. For those complicated games, the goal is of course to increase the performance of the AI.

In both cases, the general idea can be summarized as obtaining an AI whose performance is similar to that of humans. Mimicking human intelligence has by the way been suggested to be pursued directly in those new virtual environments [Laird and VanLent, 2001].

[1]Multiplayer Online Battle Arena

Human-like behavior in video games was already approached by several studies, often under the name *Imitative Learning*. Such studies include for example improvements in more natural movement behavior and handling weapon switching, aiming and firing in FPS[2] games [Bauckhage et al., 2003; Gorman and Humphrys, 2007], discussions about learning from humans at different cognitive abstraction levels including strategy, tactics and reactions [Thurau et al., 2004], and production order passing in RTS[3] games [Gemine et al., 2012].

1.3.2 Machine Learning & Video Games

In the light of the previous section, the idea of bringing the benefits of machine learning techniques to video games is not new. Researchers of the University of Liège for instance developed a theory about designing intelligent agents based on imitative learning to solve the problem of macro-management in RTS games [Gemine et al., 2012].

In recent years, MCTS has been seen as the method of choice for creating AI agents for games [van den Herik, 2010]. In particular, it was successfully applied to Go [Lee et al., 2009; Rimmel et al., 2010] and a large variety of other game environments [Browne et al., 2012], for example General Game Playing (GGP) [Méhat and Cazenave, 2010; Bjornsson and Finnsson, 2009], Hex [Arneson et al., 2010] and Havannah [Teytaud and Teytaud, 2010].

Even though we said in the previous section that large-scale video games are in general difficult to simulate, it has to be noted that attempts to apply simulation-based techniques to those games do exist, for example for the RTS genre [Soemers, 2014; Buro, 2003; Sailer et al., 2007; Balla and Fern, 2009] or *Magic: The Gathering* [Ward and Cowling, 2009; Cowling et al., 2012], a card game similar to *Hearthstone: Heroes of WarCraft*. These approaches however usually solve subproblems of the game they are applied to, and not the game itself, by making abstraction of many variables.

Some researchers were concerned about the performance issues encountered in games where the action space is too large to be thoroughly exploited by generic scripting, that is large-scale video games. By using neural networks-based classifiers, Baysian networks and action trees assisted by quality threshold clustering, Frandsen et al. [2010] were able to successfully predict enemy strategies in StarCraft, one of the most complex RTS game today.

Safadi et al. [2015] state that a big flaw in today's video game AI is its inability to evolve. They argue that video game companies are usually reluctant to make use of machine learning techniques to palliate non-evolving behaviors:

> *There are a few reasons the video game industry has been reluctant in making use of machine learning techniques. The complexity of modern video game worlds can make it challenging to design a learning agent that can be deployed in an environment where multiple concepts are in play. Even when it is possible to create robust learning agents, the process of training them can be too costly. Furthermore, evolving agents are harder to test and can be problematic to quality assurance (QA) processes.* [Safadi et al., 2015]

Even though, they also give some examples of video games – *Creatures*, Creature Labs, 1996; *Black & White*, Lionhead Studios, 2001 – for which machine learning techniques are applied to game agents in order to make them feel more "real" to players.

These techniques are however not restrained to be applied to game agents. With the emergence of massively multiplayer games, game editors began to integrate machine learning techniques into their matchmaking and player toxicity detection algorithms.

In *League of Legends* for example, players can report others for what is called "verbal harassment". This report system feeds an algorithm that learns what *players* define as verbal harassment. In this way, players flagged as verbally harassing others are players the majority of *League of Legends* community feels as toxic. Banning those players thus result

[2]First Person Shooter
[3]Real-Time Strategy

in a healthier platform, where the remaining players share the same values. This system is therefore more robust and is able to discriminate between players simply swearing and players having uncivic behaviors (racism, homophobia,...).

Still in this game, machine learning is applied to detect other toxic behaviors that are well-defined (i.e. not left at the players' judgment), like players that were "intentionally feeding" in a game. Players can indeed die on purpose to give advantages to the enemy team, in the sole goal of making their own team angry. Recognizing an act of intentional feeding is easy for a human being but less for a computer. By making use of machine learning techniques to detect based on a game log whether a player was guilty of such toxic behavior, the system proved to be efficient enough to run permanently on the game servers [Lyte, 2014].

Given these various considerations, it seems machine learning and video games are going to have a treacherous yet bright future together.

1.3.3 Existing *Hearthstone: Heroes of WarCraft* Simulators

Even though several simulators exist [Demilich, 2015; Yule, 2015; Oyachai, 2015], they are still much works in progress and usually not as extensible, modular and suited for AI research than what is actually needed. Most of them hard-code each and every card of the game, making it difficult for practitioners to test precise behaviors by designing their own cards. Most are also still prone to bugs.

However, we want to mention an exception, the *Fireplace* open-source project [Leclanche, 2015], which is a comprehensive, full-fledged *Hearthstone* simulator written in Python. When beginning this thesis, *Fireplace* was unfortunately not as mature and actively developed as it is today. It was also missing some core features needed for pursuing this research. Consequently, it was found more adapted to build our own simulator for the purpose of this thesis rather than depending on external projects. More information about the simulator will be given in the next chapter.

1.4 Structure of This Thesis

In Chapter 2, we first go through a short presentation of the video game *Hearthstone: Heroes of WarCraft*, that will give the reader some insight about its goal, mechanics and technical challenges. This description is required in the first place as further chapters will rely on some concepts of the game. As this is a commercial, closed-source video game, this same chapter contains a description of the simulator conceived as part of this work to mimic Hearthstone behavior and mechanics.

Chapter 3 presents the theory we developed for the generic design of autonomous agents for modern video games, especially for games where it is difficult or impossible to simulate side-trajectories and where the available actions have unknown consequences. Considering these constraints, its aim is to formalize the problem of playing large-scale video games. Thanks to this formalization, we develop an approach based on supervised learning classification that attempts to solve the stated problem.

This theory is then applied in Chapter 4 to the video game *Hearthstone: Heroes of WarCraft*. This chapter is a discussion about the design of *Nora*, the autonomous agent designed from the application of this theory to this very game. This part also points out the difficulties we encountered while trying to make *Hearthstone: Heroes of WarCraft* fit in the formerly stated problem, and how we circumvented them to finally come up with Nora.

The performances of Nora are then presented in Chapter 5. It describes the methodology followed for the testing, and features qualitative and quantitative experiments against a random player and a medium-level scripted player. The results and datasets were obtained through the use of the simulator showcased in Chapter 2.

Finally, we give an overall conclusion about this work by highlighting the strengths and weaknesses of the developed theory, its application to *Hearthstone: Heroes of WarCraft* and summarizing what has been done and what could not. This part also introduces the sidetracks we followed, and presents what could be done as future work to improve Nora's capabilities.

A public GitHub repository containing the Python scripts used for model training and predictions and the source code of the simulator is available at `https://github.com/dtaralla/hearthstone`.

Chapter 2

The *Hearthstone* Simulator

Summary

The aim of this chapter is to present an overview of the game Hearthstone: Heroes of WarCraft *along with a description of the simulator that was implemented to mimic this closed-source game mechanics. We first give the general rules of the game* Hearthstone: Heroes of WarCraft*. Second, some considerations that steered the way the simulator was built are presented. An overview of the simulator engineering is then given, by studying the game loop itself. Finally, the most important parts of the software architecture are described.*

2.1 *Hearthstone: Heroes of WarCraft*

Hearthstone: Heroes of WarCraft (or simply *Hearthstone*) is a turn by turn, 2-player collectible card game created and developed by Blizzard Entertainment, released in March 2014 for PCs and April 2015 for iOS and Android devices.

This game is free-to-play and played online, usually against other human players but also in solo mode against a rather basic AI. By September 2014, there were already more than 20,000,000 registered accounts [Haywald, 2014], and four months later this number had increased by five other millions [Matulef, 2015], making it a real success when considering the fact that the developer team behind *Hearthstone* was composed of only five programmers instead of the sixty to hundred people usually affected to a Blizzard game [Blizzard, 2013].

2.1.1 Game Description

The Figure 2.1 illustrates what a typical game of *Hearthsone* looks like.

The game features a board with two sides, one for each player. Each player is represented by a *hero*. A hero has a special power, no attack points (ATK) and begins the game with 30 health points (HP). Throughout this work, we will identically refer to either *heroes* or *players*, which can be considered as exchangeable terms.

The goal of the game is to reduce the health of the enemy hero to zero. To do so, players successively play cards from their hand, cards that can either be spells or creatures (with the latters actually called *minions*). A player's hand is hidden to his opponent, who only sees the card backs (he thus knows the number of cards even though he does not know their identities).

Each player has a *deck* – a stack of cards of his choice. A deck never contains the same card more than twice, and is composed of exactly 30 cards. The decks are shuffled at the beginning of the game, which means that even though players know their deck content, they can not know what card they are going to draw next.

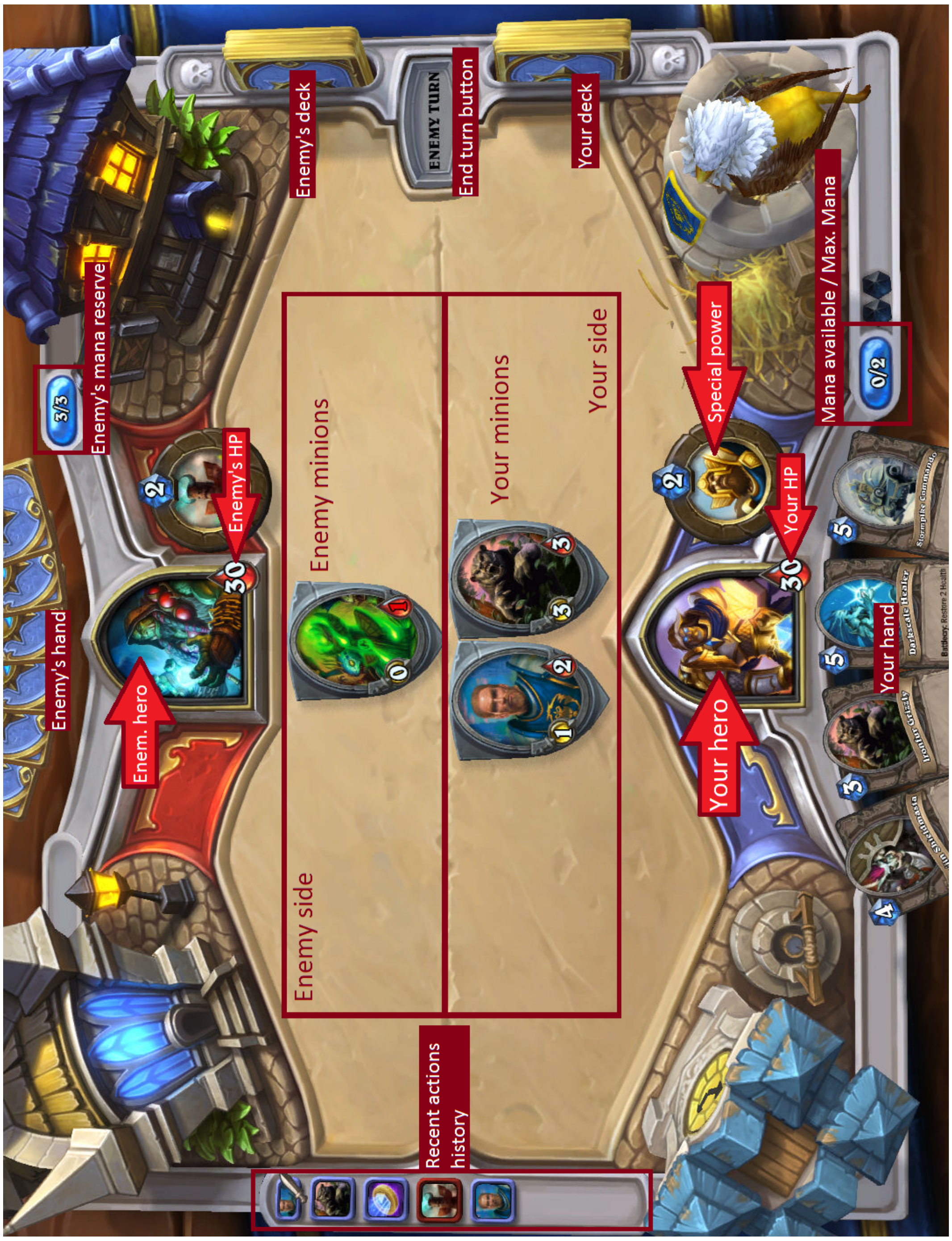


Figure 2.1: A typical *Hearthstone* board

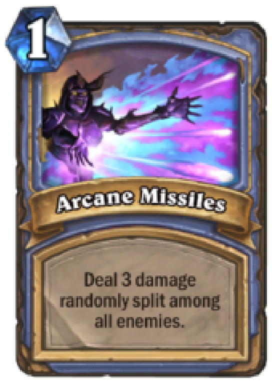


Figure 2.2: A spell card: this spell costs 1 MP. The text describes the spell's effects.

Hearthstone is turn-based: players never play simultaneously. At the start of a player's turn, he draws a card from his deck if possible. During a player's turn, he can play any number of cards from his hand. If he has played minions in his previous turns, and if they are still alive, he can make them attack enemy characters (hero or minions). The player can also use the special power of his hero, once per turn. Finally, the player can also choose to end his turn at any time.

In the case where a player is in the incapacity of drawing a card because his deck is empty, he draws a *Fatigue card*. This uncollectible card deals a number of damages to his hero equal to the quantity of *Fatigue cards* he has already drawn, this one included. This mechanism prevents never-ending games from happening.

To play cards or his hero's special power, a player needs *mana points* (MP). The mana is a resource inherent to a player which is restored at the beginning of each of the player's turns. Initially, each player begins the game with a reserve of 1 MP. At each subsequent turn, the mana reserve is incremented by one, until a maximum of ten is reached.

Special powers and cards (regardless of them being spells or minions) have a mana cost. For example, to be able to play a card of mana cost N, a player has to have at least N MP available. If he does, he can (but is not forced to) play the card, what will consume N MP from his mana reserve.

Playing a card reveals it to the opponent. If it is a spell, its text is executed then the card is removed from the game. If it is a minion, the card is dropped on its owner's side of the board at the spot of his choice (for instance between two other previously played minions), and the minion battlecry, if any, is executed (see below for more information about battlecries). A player can not play a minion if it already controls seven or more. Minions can not attack the turn they were put into play, and can also only attack once per turn.

Usually, heroes can not attack. However, a spell or their special power can make them able to do so.

Note A few elements of the game like **Secrets** and **Weapons** were voluntarily left aside here as they were not implemented in the simulator.

2.1.2 Card Text

Cards usually have text on them. This text expresses the different powers of the card: what it does when it is played, if it reacts to some event, what effect is applied on the game while the card is on the board, etc.

The text can be pretty much anything. Indeed, an implicit rule in *Hearthstone* is that any rule can be broken by cards. Most of the game behavior depends on the cards present on the board and what their text says. For instance, a general game rule would be that

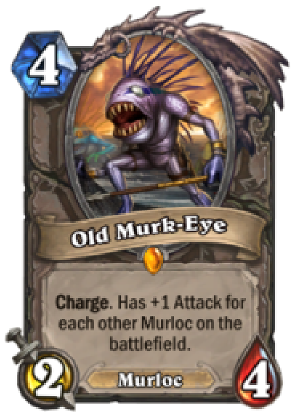

4
Old Murk-Eye
Charge. Has +1 Attack for each other Murloc on the battlefield.
2
Murloc
4

Figure 2.3: A minion card: this minion costs 4 MP, has 2 ATK and 4 HP. Its text says that it has the **Charge** ability, and also describes its permanent power.

turns never last more than one minute and thirty seconds. However, nothing prevents the existence of a card that could have a text saying something like

"*Turns can not last more than fifteen seconds.*"

This modification on the game rules applies as long as the card is present on the board. When multiple card texts would have their effects overlapping, it is the text from the last card put into play that prevails, unless specified otherwise by other cards. As you might see, this can be quite a challenge to model and implement.

Examples of card texts can be seen on Figures 2.2, 2.3 and 2.4.

2.1.3 Minion Abilities

Abilities and conditions are modifying a bit the game rules for the minion benefiting (or suffering) from them. Abilities are part of the minion's text – Figure 2.3 for example shows a minion having the **Charge** ability.

Here are the abilities and conditions implemented in the simulator:

Charge Ability. This minion can attack the turn it is put into play.

Taunt Ability. If a player wants to make his characters attack, he is forced to target one of the enemy minions having this ability.

Frozen Condition. This character loses its next attack. Practically, if the character has already attacked on this turn, it will not be able to attack on its owner's next turn. Else, it will not be able to attack on this turn, but will be on its owner's next turn.

Silenced Condition. The card text of a minion under this condition does not apply anymore. A minion under this condition thus does not benefit (nor suffer) anymore from any of its abilities, enchantments, deathrattle and powers. This condition can be removed only by some action that would send the minion back into its owner's hand. Silencing a minion will only remove its currently active effects; any abilities, enchantments, deathrattle and powers that would have in some way been granted to the minion *after* it has been silenced would still fully apply their effects.

2.1.4 Battlecry

Minions might have a **battlecry**, a text that is executed when the minion is *played*.[1] If the minion is only *summoned* thanks to some card text, like a spell, the battlecry will not be

[1]When its card is played from the player's hand, after its mana cost has been debited from its owner's mana reserve.

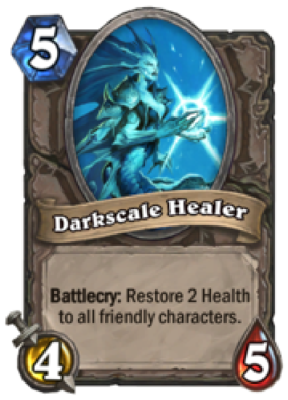


Figure 2.4: A minion card: this minion costs 5 MP, has 4 ATK and 5 HP. Its text describes its battlecry *(action executed when the minion is played)*.

executed as this minion was not *played*. The battlecry (if any) is displayed in the minion card text.

Figure 2.4 shows a minion having a battlecry.

2.1.5 Combat

Characters (heroes and minions) may attack if they have more than 0 ATK. A character can attack only one enemy target (hero or minion) per turn.

If the player's enemy has minions benefiting from the **Taunt** ability, he is forced to target one of these. The restriction disappears only when all these **Taunt** minions are dead or their **Taunt** ability was **Silenced**.

Upon attacking, the two characters will lose a quantity of HP equal to the ATK of its opponent.

2.1.6 Death

When a character's HP become less or equal to zero, its death is resolved and it is removed from the board. Some minions can also have what is called a **deathrattle**, a piece of text that will be executed upon the minion's death. The deathrattle (if any) is displayed in the minion card text.

Figure 2.5 shows a minion featuring a deathrattle.

2.2 Considerations

Here are some considerations that steered the way the simulator was built. It was a design-driven programming project.

Easy creation/editing of cards. Some researchers might want to test their agent in precise settings. These settings can be modeled by the card list available in a game. But, rather to only allow researchers to pick cards from a predefined database, the goal was for them to be able to *design their own cards*. Indeed, because they are mostly interested in their agent performance in a given setting, they should not have to dig in the simulator code and recompile it each time they want to add or modify a card. Consequently, the first goal of the simulator was to have an easily-expendable card database.

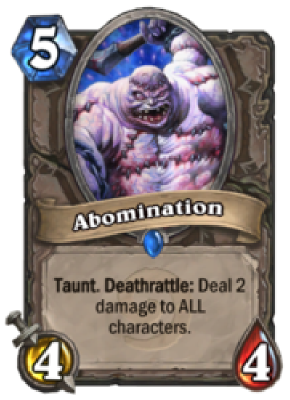


Figure 2.5: A minion card: this minion costs 5 MP, has 4 ATK and 4 HP. Its text says that it has the **Taunt** ability and describes its deathrattle *(action executed upon the minion death).*

Easy extension. The initial simulator was meant to embed basic actions that could be applied on the game: dealing damage to a target or randomly, silencing targets, freezing targets, enchanting them, make a player draw a card, gain some mana, etc. However, the library users could want to implement new types of actions that fulfill their precise needs. These actions can be anything, as they are in the original *Hearthstone*. It should also be able to easily implement the missing mechanics, like responding to new events, new minions abilities,...

Multiprogramming. The simulator should manage game threads, such that multiple games can be played simultaneously based on the same card collection, without having to reload the entire collection for each game.

2.3 Game Loop

Before going into details about the software architecture, we wanted to present a big picture, an overview of what is carried on in the game from a high-level perspective. This can be crystallized by an abstract view of the game loop.

Everything in the game is about updating the state through a call to an `updateState()` function. What it does is first checking if anything triggered, and if so calls itself recursively. Second, it checks if anybody died, and if so calls itself again.

The game loop is a repetition of the three following steps:

1. Get the current player's options and hand them to him.
2. Run the action associated with option taken.
3. Call `updateState()`.

This algorithm makes sure everything is correctly handled at the right time. The recursive calls to `updateState()` are necessary in that of it enables the game to handle a stack of events. It also allows the game to process the consequence of a character death: a minion featuring a deathrattle sees its deathrattle executed by the last recursive call to the function.

We mentioned a *stack* of events. As other card games, *Hearthstone* implements some kind of *stack* rule: an action is considered resolved only once *all* its consequences are resolved, what is translated into some event resolution nesting. In the case where an event would trigger several independent actions, they are executed in the order in which the source cards were put into play.

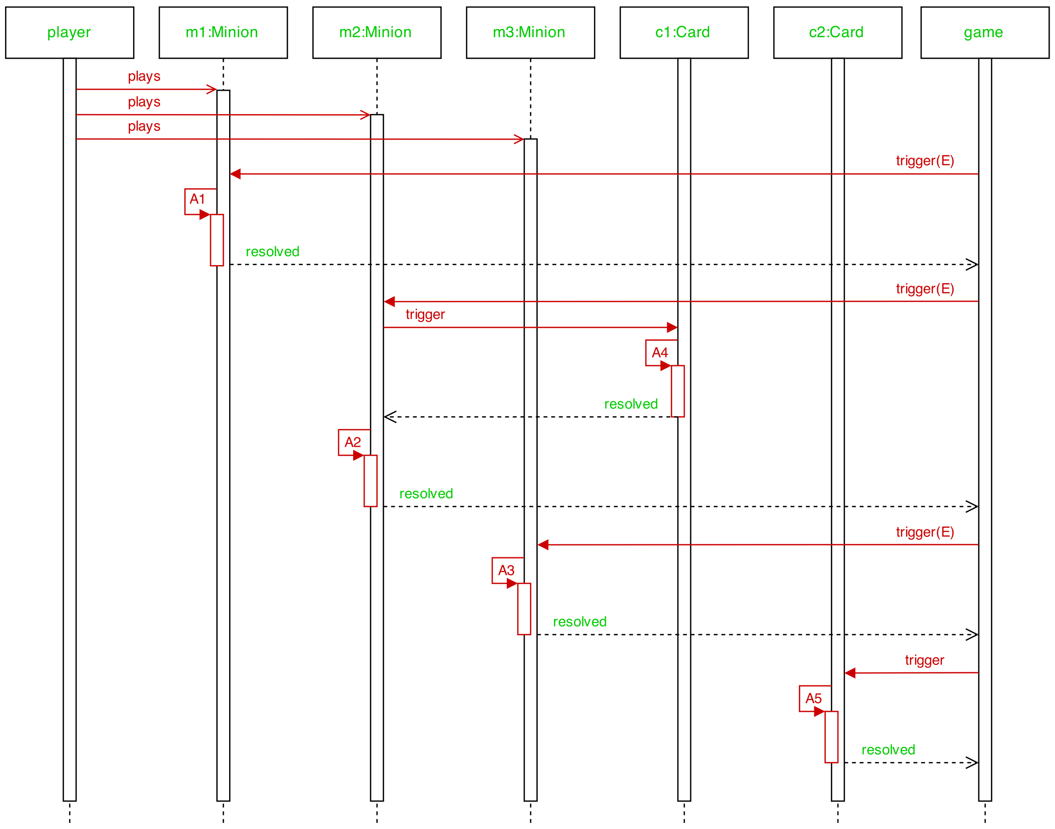


Figure 2.6: Sequence diagram of an example event nested execution

Event nesting is avoided for cards that listen to "after [something] occurred" events. In this case, it means that the action triggered is executed *after the triggering action is resolved* instead of *before*.

Let us take the example where there are three minions M_1, M_2 and M_3 on the board, summoned in this order, all listening for an event E, that execute respectively actions A_1, A_2 and A_3 when it occurs. Let us further say that action A_2 will fire an event to which another card is listening to to trigger action A_4, and that action A_3 will fire an event *after which* a last card will execute action A_5. When E is fired by the game, we have the following nested execution:

1. A_1 does not trigger anything before resolution.
2. A_1 is executed and marked as resolved.
3. A_2 triggers something before resolution.
   (a) A_4 does not trigger anything.
   (b) A_4 is executed and marked as resolved.
4. A_2 is executed and marked as resolved.
5. A_3 does not trigger anything before resolution.
6. A_3 is executed and marked as resolved.
7. A_5 does not trigger anything before resolution.
8. A_5 is executed and marked as resolved.

This execution is represented as a sequence diagram on Figure 2.6.

2.4 Software Architecture

The *Hearthstone Simulator* library is based on the Qt 5 Framework. Figure 2.7 puts the software architecture in a nutshell as a condensed UML diagram.

The design, implementation and testing of this library was a huge part of this thesis, so we wanted to at least succinctly present the software engineering processes used to build it. Along with the library were developed two programs dynamically linked to it: a database generation program and a demonstration one. The latter features a graphical user interface to make one able to test and play the game visually. This graphical user interface was also used to play against our autonomous agent.

The following sections describe more thoroughly some critical parts and entities of the simulator. As far as the other structures are concerned, they will not be described further in this thesis as its goal is to present the design of an autonomous agent.

2.4.1 Cards

Cards can be looked at in two different ways: the card *itself* – its name, text, mana cost,... – and an *instance* of the card – the card in a player's hand or on the board. These roles are represented respectively as the *card identity* and *card* entities. The card is what you play with, and you can have multiple cards sharing the same card identity. One can see the way we represented cards in the library as the *Flyweight* pattern, whose goal is precisely to handle efficiently a large combination of similar objects. Take a deck containing twice the *Shattered Sun Cleric* card. It would be pointless to store twice its maximum HP, ATK, name and text. Furthermore, imagine this in the context of multiprogramming: if N games were using this deck simultaneously, it would mean duplicating this information N times unnecessarily, what would lead in a large waste of memory.

Card Database

The library features a `CardDB` singleton, able to load a file describing the list of cards in a human-editable way. A card database file is written in JARS[2], a JSON-like language we designed along the simulator. The root object has to be a JSON array, and each element of this array has to be a card description.

JARS is only "JSON-*like*" because even if it shares with JSON its way of representing objects and arrays, it allows the card designer to use human-readable constants wherever he want, as long as they are constants recognized by JARS. These constants are always prefixed with the $ sign.

Using an external file with well-defined syntax and semantics allows any third-party developer to create new cards for the game using the available types and actions and without touching a single line of code.

As in regular JSON, the order in which you put the fields and their values is not taken into account. What is important however, is the presence of mandatory fields. JARS ignores fields that are not documented for the objects it recognizes.

An example of JARS code featuring a minion is given in Figure 2.8.

Detailed information about the JARS language can be found as part of the code documentation present in the GitHub repository coming along this thesis.

[2]Just Another Representation Syntax

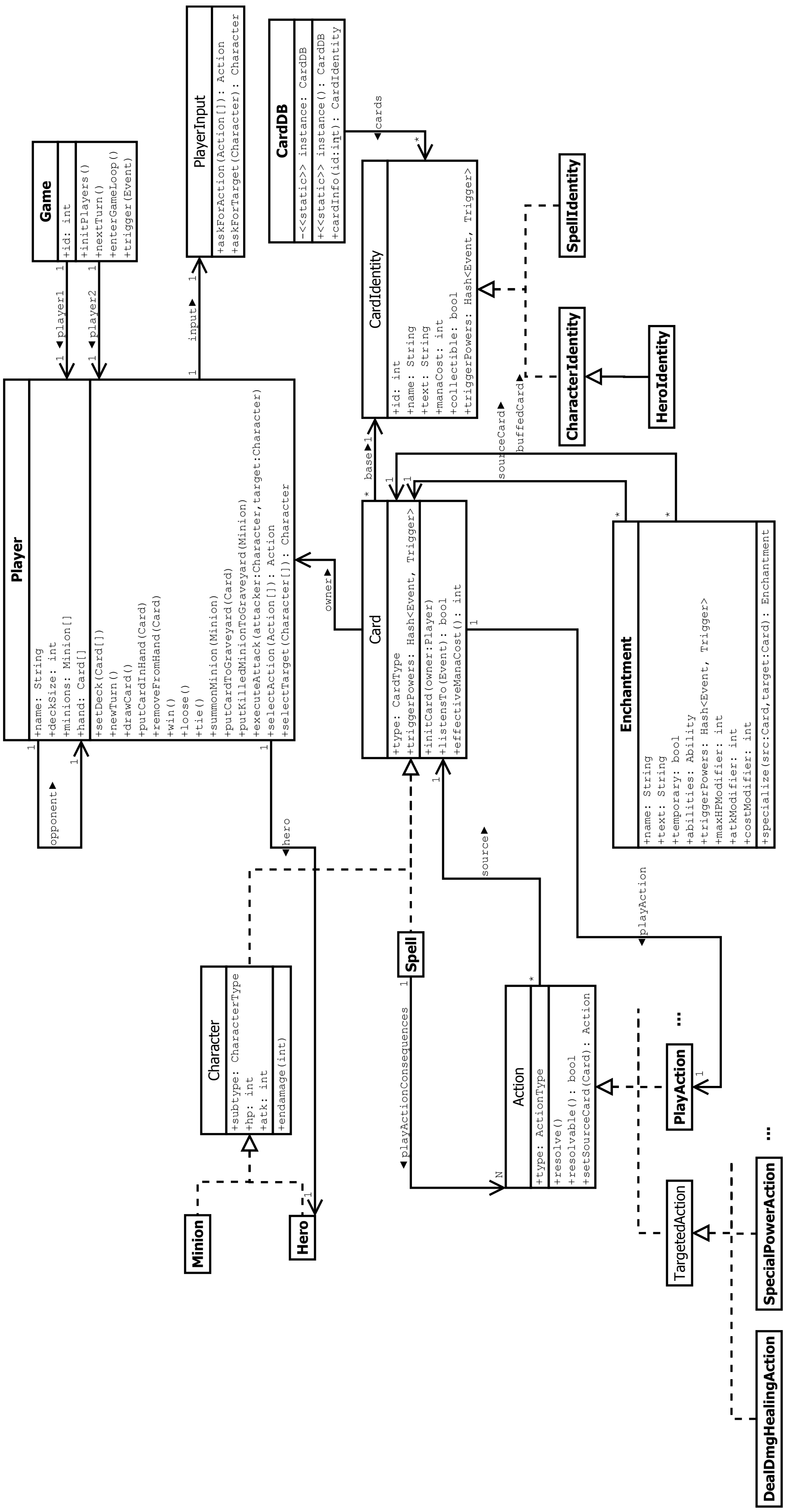
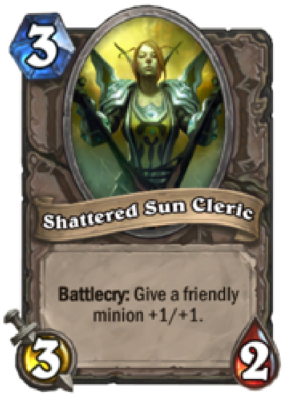


Figure 2.7: A summary of the library software architecture



(a) Represented minion

```
{
    "id": "SHATTERED_SUN_CLERIC",
    "type": $CARD_MINION,
    "name": "Shattered Sun Cleric",
    "text": "<b>Battlecry:</b> Give a friendly minion +1/+1",
    "cost": 3,
    "attack": 3,
    "health": 2,
    "battlecry": {
        "id": $ENCHANT_CHARACTER,
        "target": {
            "id": $SELECT_N_IN_GROUP,
            "quantity": 1,
            "group": {
                "owner": $OWNER_ALLY,
                "subtype": $TYPE_MINION
            }
        },
        "enchantment": {
            "name": "Sunwell energy drink",
            "text": "+1/+1",
            "atkModifier": 1,
            "maxHPModifier": 1
        }
    }
}
```

(b) Minion in JARS

Figure 2.8: The *Shattered Sun Cleric* and its JARS representation

2.4.2 Actions

Actions are usually always associated to a card, and cards have actions associated to them. All cards have for instance a link to a `PlayAction`, executed when its owner plays it. Spells have an ordered list of actions to be executed when they are played. A minion can also have a battlecry, a deathrattle and/or some trigger powers, that are just actions triggered by some event.

`Action` objects describe precisely something that can be executed to alter the game state. Some examples:

- Deal 1 damage to an enemy minion
- End the turn
- Summon a 1/1 Murloc Scout
- ...

Along with this description comes an implementation of the `Action`, materialized by the `resolve()` factory method. This method will alter the game state according to the nature of the action (its class) and its parameters (its constructor arguments, most of the time).

In software engineering, one would call the way actions were coded as *Prototype*. Because a card was chosen to be represented as an intrinsic state (its `CardIdentity`) coupled to an extrinsic state (the current HP of the card, the enchantments it benefits from,...), an `Action` can be one of two things: either a description, or a description *and* its implementation.

An `Action` linked to a `CardIdentity` is a prototype of this action. When a `Card` is created based on a `CardIdentity`, the actions linked to the `CardIdentity` are cloned and specialized for this very `Card`, giving them the context they need to execute themselves; this card is the *source card* of these `Actions`. An `Action` which has no source card usually can not compute its resolution, as it is missing context for its execution.

To make things clearer, take the following example. Let a spell be

"*Fireball: Deal 6 damages to an enemy minion.*"

In its play action consequences, the `SpellIdentity` that the spell is based on has a `DealDmgHealingAction`, with a parameter set to 6 for the amount of damages to deal, and another parameter describing the fact that the target has to be an enemy minion. In the `SpellIdentity`, the `Action` is nothing but a *prototype*, a description of the action linked to the spell. Indeed, the `Action` could not be executed as it is not linked to some `Card`, and thus has no way for now to depict a player as an enemy or a friend, so can not evaluate its possible targets.

When creating a spell card based on the *Fireball* `SpellIdentity`, and putting the resulting `Card` in a player's deck, the prototype `Action` "*Deal 6 damages to an enemy minion*" is cloned and specialized to work with a real `Card`, which has an owner. From this context information, the specialized `Action` will be able to evaluate what its possible targets are as it now has a reference to what it considers an ally.

The cloning and specialization of an `Action` prototype is done using its `setSourceCard()` method; it will return a clone specialized for the very card given as argument.

2.4.3 Players

The `Player` class, as its name suggests, represents a player. However, player input/output is implemented in a subclass of the abstract `PlayerInput`. To a `Player` is associated one and only one `PlayerInput`. The latter lives in the UI thread and the first in the game thread, in this way the input/output is not blocked while the game updates its state.

This scheme ensures that AI is decoupled from the library. The library represents a player with its `Player` class, but makes no assumption on the kind of input or output it is given (there is a clear separation between the UI and the game). In this way, anyone desiring to develop an intelligent agent for the simulator solely has to create a program where they subclass `PlayerInput`, and override its `askForAction()` and `askForTarget()` methods such that these reflect the agent behavior. The demonstration program for example contains four subclasses of `PlayerInput`:

Human player GUI. This class implements `PlayerInput` in such a way that the program displays a graphical interface and asks the user for input when an action or target as to be selected. The graphical interface is updated with the game state, and everything that happens is logged in a separate window, making debugging of the library easy. It can also be used to make oneself play against another human player or one of the players below.

Random player. This input select actions and targets at random. The only constraint is that it only ends its turn when it can not do anything else. This class has no graphical interface nor outputs anything on the screen; it just plays randomly. In code, it translates into a subclass of `PlayerInput` only overriding the `askForAction()` and `askForTarget()` methods and ignoring state update signals.

Scripted player. This input selects actions and targets using a fixed, well-defined policy, designed by *Hearthstone* experts. It analyzes the current state of the game and choses its actions and targets accordingly, with a level of medium-difficulty player. This input is deterministic, hence the *scripted* term. As the random player input, this does not display anything on the screen nor require any input from the user.

Nora. This input is the result of this thesis; its behavior will be described further in the following chapters. This does not require any input from the user, but outputs on `stdout` what "crosses its mind" when selecting actions and targets.

Inter-thread messaging was implemented thanks to the `Signal`/`Slot` mechanism of the Qt framework.

Game state updates rely on the *Observer* pattern. When the subject – the game – is updated, it sends a non-blocking signal containing all the information about what was updated and how to the registered `PlayerInput` objects signal queue – the observers. The observers are free to handle this signal or not.

For player input handling, things are a little bit more complicated; Figure 2.9 pictures a sequence diagram illustrating the collaboration between objects to handle the input. When input is required by the game to carry on (for example, asking the current player to take an action), the game sends a blocking signal to the corresponding `PlayerInput` object. The signal contains a pointer to an `IORequest` object, and it is the responsibility of the contacted agent to set the `IORequest`'s `response` field to the address of the input value chosen.

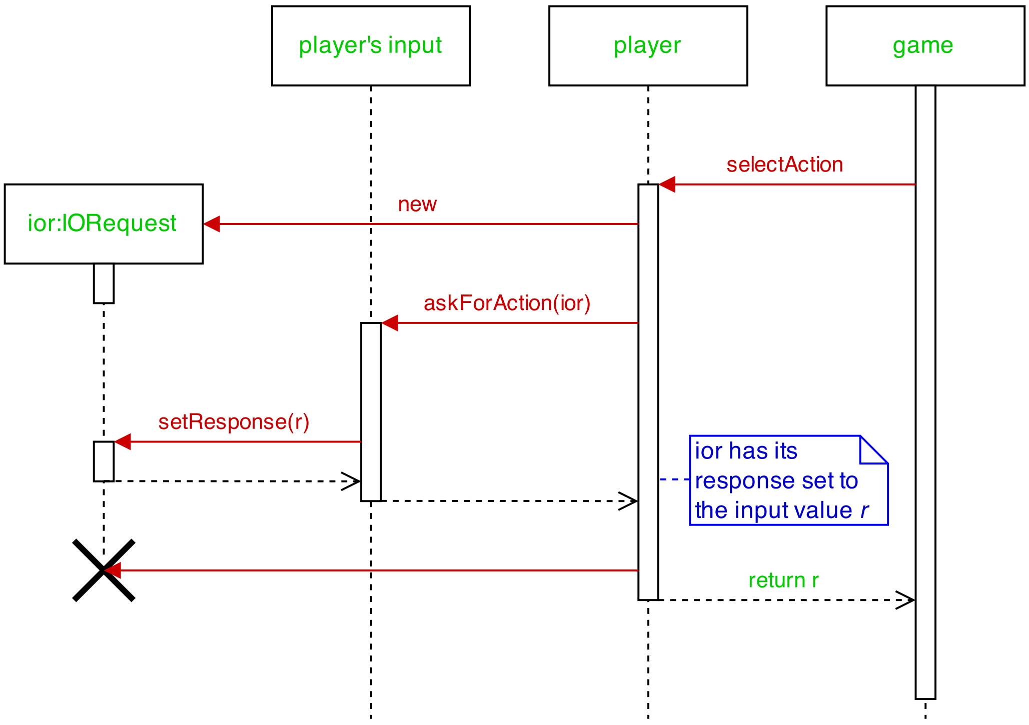


Figure 2.9: Player input handling sequence diagram

Chapter 3

General Theory

Summary

This chapter presents the general technique we developed to create intelligent agents for complex games. It first formally states the problem to solve, then present an approach based on supervised learning classification that can be applied to any game formalized as presented in Section 3.1.

3.1 Problem Statement

Originally, the approach developed in this thesis was meant to work for the game *Hearthstone: Heroes of WarCraft*, which is a turn-based, two-player card game featuring randomness and partial observability. However, our theory does not only apply to this particular, restrained class of games.

There is no restriction on the games to which this theory applies, as all games exhibit enough structure such that their states and actions can be represented as fixed vectors of numbers. We consider computable the set of all actions available to a player in a given state.

The ultimate goal is to come up with a novel approach that works for games that can not simulate trajectories easily or at all.

Finally, we take the assumption that there is a large amount of off-line computing resources, but that there are constraints on on-line resources. This assumption is done because usually, big video game firms have almost unlimited computation power while developing their games, but at the end, these games have to run on the players' personal computers which have much lower capabilities.

3.1.1 State Vector

Let the game be represented by a world vector $w \in \mathcal{W}$. For games likes *Hearthstone*, the cardinality of $\mathcal{W}$ can be pretty large, as can the length of w. We consider that w contains all the information available to an agent at any given time.

A world vector w can however contain unrelevant data for the learning and predicting processes of the targeted application; usually, only a sub-vector s of w will be needed.

Formally, if $\sigma(\cdot)$ is the projection operator such that

$$\forall w \in \mathcal{W},\, s = \sigma(w)$$

is the relevant part of w for the targeted application, we define

$$\mathcal{S} := \{\sigma(w) \mid w \in \mathcal{W}\}$$

the set of all state vectors.

The game world vector w (and thus the state vector $s = \sigma(w)$) is modified when an action is executed. This action might be taken explicitly by a player, or triggered by an event. The next section describes how we represent such actions.

3.1.2 Action Vector

The set of actions that could occur in the targeted game is considered unknown. In this way, the general theory we develop does not depend on the type of actions at hand, nor in fact on the type of game we are applying this theory to.

The actions should however exhibit some common features, such that there exists a one-to-one relation between any possible action and its representation in $\mathbb{R}^n$ for some $n \in \mathbb{N}_0$.

More formally, if $\mathcal{A}$ is the set of possible actions, there should exist a bijection Φ such that

$$\Phi : \mathcal{A} \to \mathbb{R}^n \mid a \mapsto (\phi_1, \phi_2, ..., \phi_n), \quad n \in \mathbb{N}_0.$$

In some games, not all actions can be taken in a given state $s \in \mathcal{S}$. We will thus further define

$$\mathcal{A}_s := \{\Phi(a) \mid a \in \mathcal{A} \text{ and } a \text{ can be taken in state } s\}.$$

3.1.3 State Score Function

Last but not least, there should exist a bounded function

$$\rho : \mathcal{S} \to \mathbb{R}$$

associating to a given state $s \in \mathcal{S}$ a score representing how much a player is in a good position to win the game in this state. The more the player is winning, the larger the value of ρ. Furthermore, ρ should have the following properties:

$$\begin{cases} \rho(s) < 0 & \text{if, from state } s \text{ information, the player is considered as likely to lose,} \\ \rho(s) > 0 & \text{if, from state } s \text{ information, the player is considered as likely to win,} \\ \rho(s) = 0 & \text{otherwise.} \end{cases}$$

This should usually be where expert knowledge on the game is put. For *Hearthstone* for example, some expert players helped us designing one of its ρ function. It has to be noted that the value of $\rho(s)$ could crystallize information about previous states and foreseen subsequent states.

3.1.4 Problem Formalization

We can see games as systems having discrete-time dynamics described by

$$\tau : \mathcal{S} \times \mathcal{A} \to \mathcal{S} \mid (s_t, a) \mapsto s_{t+1} \text{ for } a \in \mathcal{A}_{s_t}, \quad t = 0, 1, ...$$

Let R_ρ be an objective function whose analytical expression depends on ρ, such that

$$R_\rho : S \times \mathcal{A} \to \mathbb{R} \mid (s,a) \mapsto R_\rho(s,a) \text{ for } a \in \mathcal{A}_s.$$

We seek to optimize for $t = 0, 1, \ldots$ the objective R_ρ, thus to find an action selection policy h such that

$$h : S \to \mathcal{A} \mid s \mapsto \underset{a \in \mathcal{A}_s}{\operatorname{argmax}} R_\rho(s,a).$$

An agent using policy h to determine which action to take amongst the available ones in a given state should follow the expert knowledge and/or experience crystallized in ρ. Indeed, as long as ρ and R_ρ are "well" defined, the agent should select actions worth to be played in any state, what would eventually lead it to victory.

A first analytical expression for R_ρ is the difference between the values of ρ at two subsequent time steps induced by the choice of taking an action $a \in \mathcal{A}_s$ in state s:

$$R_\rho(s,a) := \rho(\tau(s,a)) - \rho(s).$$

Considering this analytical expression, if taking action a in state s_t is putting the game in a state s_{t+1} such that $\rho(s_{t+1}) > \rho(s_t)$, $R_\rho(s_t, a)$ will be positive, meaning that a is considered as a "good" action to take in state s_t. On the contrary, if taking action a yields to $\rho(s_{t+1}) < \rho(s_t)$, $R_\rho(s_t, a)$ will be negative, meaning that a is considered as a "bad" action to take in state s_t.

It is worth mentioning that the value of $R_\rho(s, \cdot)$ is considered *uncomputable* for the targeted games, as it works with actions having unknown effects and as action simulation is not possible. The idea is that the agent proposed will *learn by itself* the impact any action has on the game, and therefore whether taking this action is likely to lead it towards a winning or loosing state. This can be achieved by learning to predict the value of R_ρ.

3.2 Supervised Learning Classification

We make the assumption that a dataset

$$T = \{(s,a,r) \in \mathcal{S} \times \mathcal{A} \times \mathbb{R} \mid a \in \mathcal{A}_s \text{ and } r = R_\rho(s,a)\}$$

of recorded game states is available for the learning phase. Each sample $t \in T$ is a tuple (s,a,r) where $r = R_\rho(s,a)$ is the output of t and the concatenation of s and a represents the features of t. There are no further assumptions on T, not even on the quality of the actions taken. From then on, the process of generating the database can be as simple as letting random players play the game for some time, and making them explore the space of available state-action combinations.

Indeed, if a random player chooses to take action a in state s, it can save the value of $\rho(s)$, execute a and compute $\rho(\tau(s,a))$. The tuple $(s, a, \rho(\tau(s,a)) - \rho(s))$ can then be inserted in a dataset like T.

3.2.1 Extremely Randomized Trees Binary Classifier

The goal is to approximate R_ρ for unlabeled and new samples. Such an approximation for instance is to predict whether it is positive or negative. In this way, we will be able to give the agent some insight about the soundness of the possible actions it is presented. Formally, the assumption we make here is that the value of R_ρ itself is not important, as long as its sign is well-defined and sufficient for the targeted game to be winnable just by taking "positive" actions. This work can be carried out by a binary classifier that yields the predicted class probabilities, i.e. the probabilities that a sample belong to one or the other class.

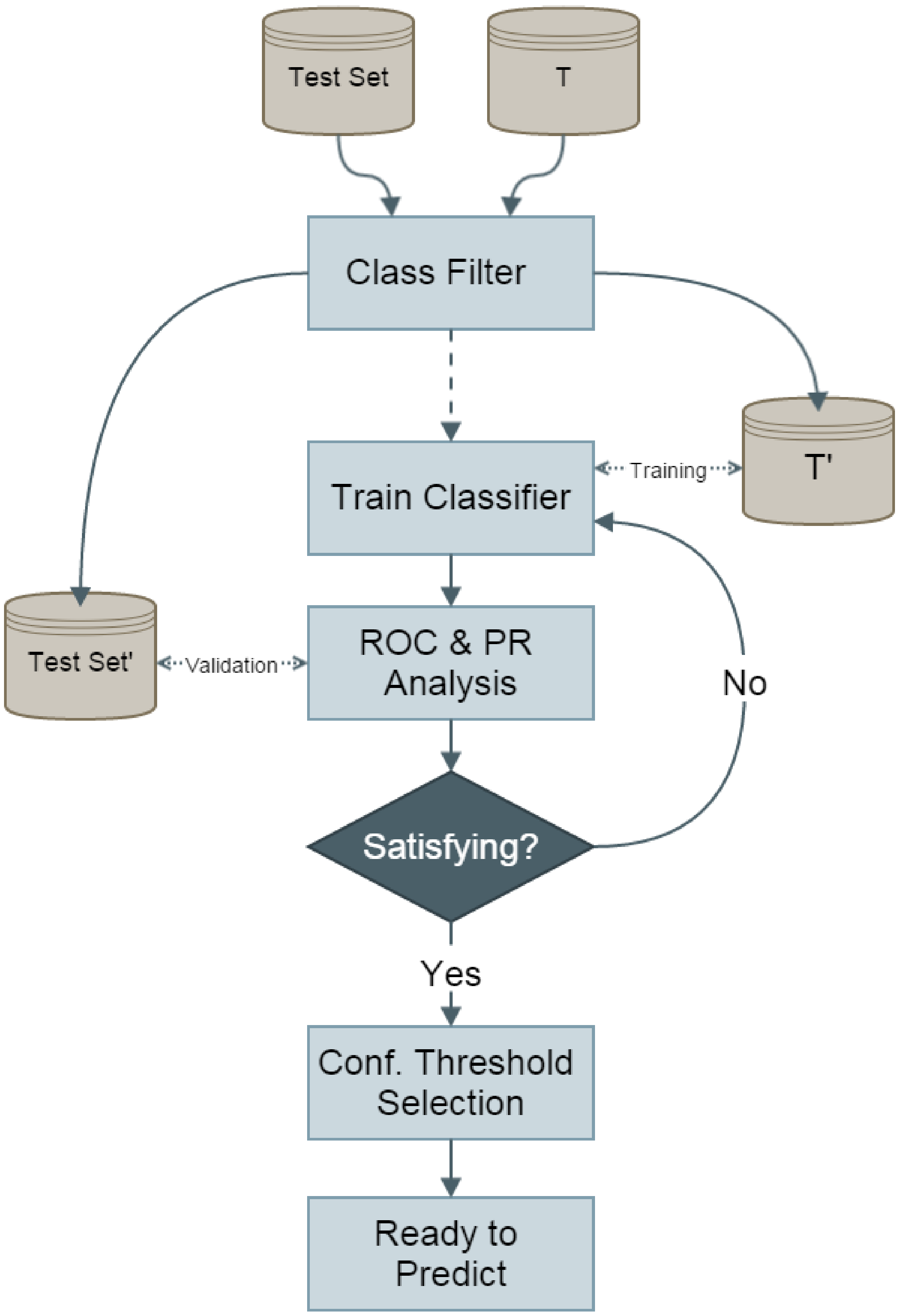


Figure 3.1: Our Supervised Learning Process

We chose to see the problem as a binary one so that the whole Receiver Operating Characteristic (ROC) and Precision-Recall (PR) framework is available, making one able to easily assess the quality of the classifier. Some research papers like Patel and Markey [2005] discussed the various ways to handle classification tasks where the number of classes is greater than two, but we chose to stick with the common case of binary problems because it is simpler to analyze and, for our particular application *Hearthstone*, a two-class classifier proved to be sufficient to get interesting results.

To this end, the decision to use random tree-based ensemble methods was taken. Decision trees are inherently suited for ensemble methods [Sutera, 2013], and random forests applying the random subspace [Ho, 1998] or random patches [Louppe and Geurts, 2012] techniques are particularly good choices for problems with many variables, as they artificially reduce the number of variables for each model by the randomization they individually bring.

Moreover, as explained in Dietterich [2000], ensemble methods can often surpass single classifiers, and this from a statistical, computational and representational point of view. Plus, decision tree ensemble methods are particularly computationally efficient. Even though usually ensemble methods are not efficient because of their need to create and maintain several models, decision tree-based ensemble methods do not suffer that much from this caveat because of the low computational cost of the standard tree growing algorithm [Geurts et al., 2006].

Besides being efficient algorithms, they also bring a lot of information about feature importances and thus feature selection [Breiman, 2001], that can be extremely valuable when designing the projection operator $\sigma : \mathcal{W} \to \mathcal{S}$.

In particular, we chose to work with extremely randomized trees, first introduced by Geurts et al. [2006]. They showed that even if extremely randomized trees generally use a bit more memory than Tree Bagging [Breiman, 1996] or Random Forests [Breiman, 2001], they outperform these techniques in terms of computational efficiency. They also showed that, on many different problems, extremely randomized trees were as accurate or more accurate than these other ensemble methods.

Considering the targeted application, which is predicting the soundness of playing a certain action according to a large amount of features, there was a requirement of using a robust learning algorithm able to deal with the possible idiosyncrasies of the R_ρ function while ensuring it was handling the numerous features correctly. The study above convinced us that an extremely randomized trees binary classifier was a fair choice for this purpose.

Nevertheless, the main caveat of using this kind of classifier is the memory it requires. Indeed, because the games complexity is not bounded in our theory, it may not be assumed that the learned models will always fit in the memory of a player's personal computer as the trees might become arbitrarily complex. To compensate, one should therefore take care of limiting the tree growing by tuning the algorithm parameters correctly and include a tree pruning pass.

Note The functioning of the extremely randomized trees algorithm is summarized in Appendix B.

3.2.2 Training the Classifier

First, a dataset T' is obtained by modifying T such that the output of each sample now represents a class rather than a real number:

$$T' = \{(s, a, \operatorname{sgn}(r)) \mid (s, a, r) \in T\} \subset \mathcal{S} \times \mathcal{A} \times \{-1, 1\}$$

How to deal with samples whose output $R_\rho(s, a) = 0$ is specific to the targeted application. They can either be included in one of the two classes, or simply not included in T' at all.

The binary classifier is then trained on T'.

3.2.3 Assessing the Classifier Performance

When it comes to classifiers, it is well-known that the accuracy (i.e., the fraction of successfully classified samples in a given test set) is not a good measure to assess a model generalization performance [Provost et al., 1998]. It is recommended when evaluating binary decision problems to use the ROC curve, describing how the fraction of correctly classified positive samples varies with the fraction of incorrectly classified negative samples. Nonetheless, ROC curve analysis might overestimate the performance of a binary classifier if there is a large skew in the class distribution [Davis and Goadrich, 2006]. Because no constraints were put on the dataset T, one can not rely on the fact that it provides a balanced number of positive and negative samples.

This is why along ROC curves analysis we validate our binary classifier with the help of PR curves. PR curves have been mentioned as an alternative to ROC curves for tasks with a large skew in the class distribution [Craven, 2005; Bunescu et al., 2005; Goadrich et al., 2004]. Indeed, when the proportion of negative samples is much greater than that of the positive ones, a large change in the fraction of false positive can lead to a minimal change in the false positive rate of the ROC analysis, because they are underrepresented in the test set. Precision analysis on the other hand relates false positives to true positives rather than true negatives, and therefore does not present the same flaw than ROC analysis in the case where negative samples are overrepresented in the test set. Therefore, analyzing both graphs to assess the quality of the model is necessary.

To classify a sample, the model yields the probability of it belonging to the positive class.[1] It means that the performance of the model will depend on the probability threshold used to predict whether a sample is positive or not. Each point of a ROC or PR curve is the performance of the model for a given threshold; thus, according to the needs of the targeted application, one can select the confidence threshold c for which he considers having the best compromise between the true positive rate, false positive rate and precision. The best trade-off is defined by the targeted game, as for instance losing an opportunity (predict a false negative) might or might not be worse than making a mistake (predict a false positive) in all games.

3.2.4 Selecting a Good Action

From now on, when an agent in state $s \in \mathcal{S}$ is presented the set of available actions $\mathcal{A}_s = \{a_1, ..., a_n\}$, it simply has to evaluate the probability p_i that the sample (s, a_i) yields a positive value for R_ρ, $\forall i$ s.t. $a_i \in \mathcal{A}_s$. It then extracts from $\mathcal{A}_s$ the actions a_i such that $p_i \geq c$, which gives it a set of valuable actions to take (randomly, or always the one it is the most confident in). In the case where the extracted action set is empty, the agent should decide to do nothing, or end its turn if applicable.

Figure 3.2 is an activity diagram describing the action selection process.

Remark For some games, predicting whether an action is good or not might not be the interesting point. Rather, it could use the classifier only to get an ordering of the available actions, from the one that seems the best to the one that seems the worst, and play them in this order. According to the targeted game, this approach might be more relevant. Even though this strategy does not need to find the right confidence threshold, it is however still prone to fail in the case where poor classifiers are used: the study of the ROC and PR curves should thus still be conducted to avoid those cases.

[1]This probability is computed as the mean predicted *positive* class probability of the trees in the forest. The predicted *positive* class probability of an input sample in a single tree is the fraction of *positive* learning sample cases in the leaf the input sample falls in.

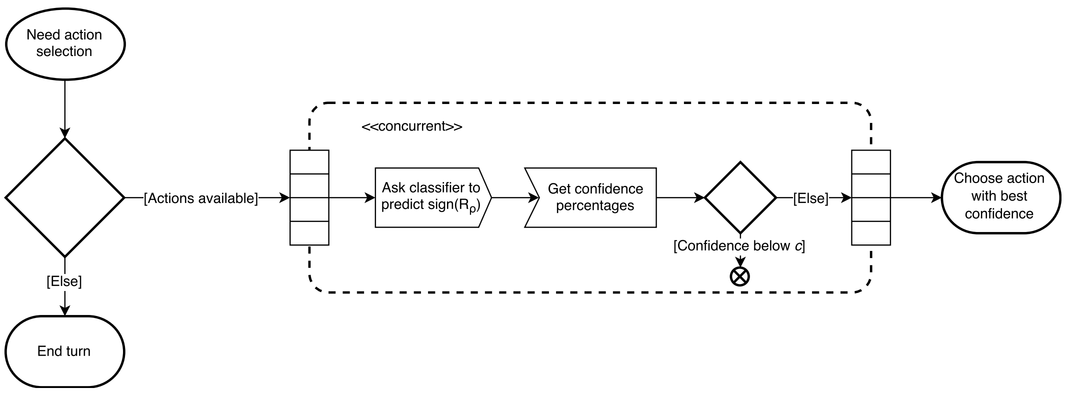

Need action selection
[Actions available]
[Else]
End turn
<<concurrent>>
Ask classifier to predict sign(R_ρ)
Get confidence percentages
[Else]
[Confidence below c]
Choose action with best confidence

Figure 3.2: The action selection process

Chapter 4

Application

Summary

The purpose of this chapter is to present an application of the theory developed in the previous chapter. It first introduces the application selected, by enumerating the motivations behind our choice. Next, it describes the game configuration, or in other words what we limited ourselves to in this game. This chapter then goes on with the designed game representation, such that it fitted the format introduced in the problem statement (Section 3.1). Finally, it presents the methodologies followed for training and testing the resultant model.

4.1 Why *Hearthstone: Heroes of WarCraft*?

The theory developed in the previous chapter was chosen to be applied to the game *Hearthstone: Heroes of WarCraft*. The following reasons motivated this choice.

4.1.1 *Hearthstone* Has Interesting Characteristics for the AI Field

There is no scientific papers to date presenting nor analyzing the behavior of autonomous agents for this game. We found it a good challenge to be pioneers in the development of an AI for *Hearthstone*. Moreover, the game is extremely complex, in a different way state-of-the-art games like Go are: its unknown action space, partial observability, randomness and state representation are difficulties one does not face in those well-studied games. Card games typically exhibit a large amount of hidden information, so they quickly massively generate branching factors in their game tree which may make even that of Go look small.

We found it thus even more interesting to see if machine learning in general could help us designing an intelligent agent for it based on some expert knowledge on the game, with as goal to obtain the level of a beginner human player.

4.1.2 *Hearthstone* is a Modern, Fun Game to Study

As a lesser goal, we also wanted to introduce a new, fun benchmarking tool for the field of games AI, more in the spirit of the times as far as video games are concerned. Plus, in order to do so we needed to find or create an open source simulator all practitioners could use to test their research against. Creating a simulator from scratch for this complex game was an interesting experience to conduct as conclusion of the master in computer science and engineering.

4.2 Game Configuration

The game features were limited such that the state and action spaces remains explorable in a relatively fast time. As these spaces dimension is mainly dictated by the available card list, we advise you to give a look at Appendix A which presents the card list the decks used for training and testing were based on. This list will give the reader some insight about the type of actions that could occur in simulated games.

Players always used the *Mage* class for their heroes. The Mage has the *Fireblast* special power, which costs 2 MP and deals 1 damage to a character (enemy or ally, hero or minion) selected by the player.

4.3 State Vector Representation

For *Hearthstone*, the world vectors $w \in \mathcal{W}$ should represent all the information a player sees on a board like the one presented back in Figure 2.1. However, this complete feature space was reduced such that the state vector $s \in \mathcal{S}$ contained only information that *Hearthstone* experts deemed relevant. These relevant features are summarized in Table 4.1.

4.3.1 About the Difficulty of Representing a State

Problems arose with the state representation: how to handle the missing values in the state? For instance, take the features describing the minions on the board. There can be zero to fourteen minions on the board: how to represent the empty spots? The same problem went for the cards in the player's hand, as one may have zero to ten cards in hand.

For the latter issue, because each card in the hand is represented by its ID only, we were able to circumvent the problem by enumerating all possible card identifiers as features of the state vector, whose value would be set to the number of cards having this ID. This solution however made the state vectors length grow linearly with the number of cards in the available card list.

Solving the other problem was not as easy because minions on the board are characterized by more than just their ID: their remaining HP, their current ATK, etc. We chose to represent each of the fourteen spots for minions on the board by vectors of 7 values, which are described in Table 4.2. For empty spots, missing value marker would be put in those vectors, while occupied spots would contain the information relative to the minion it held.

The spot used by a minion is not important, but its position amongst its neighbors is, in the case where area-of-effect actions target a minion and its direct neighbors. The list of occupied spots is thus always contiguous and its order is the same than the order of the minions on the board.

Because it would not be desirable the algorithm would have learned that first spots of each side of the board are the most important (its pretty common to have only 2-3 minions on each side, and rarer to have more), the contiguous list of spot vectors were put at a random position.

4.3.2 Feature Vector Design

It has to be noted that the variable importances given by extremely randomized trees were not used to extract valuable features from the world vector, because we did not have the time to pursue a precise analysis of these importances. Rather, we used our own knowledge of the game and of how we defined the ρ function to select them.

4.3.3 Incorporating Identifiers in the Features

Because features whose values would describe the text of cards could not be designed (recall that it can be anything), we wanted to have a way to "encapsulate" its meaning in some kind of "meta-feature" and learn it implicitly from this feature. We made the hypothesis that unique identifiers for cards could possibly be enough for the classifiers to grasp their influence on the game.

4.4 State Score Function

The state score function ρ we used was designed to exhibit a "board control" characteristic. It thus does not have the ambition to lead to an absolute winning strategy by itself, but rather to a piece of strategy that could be used along with others to ensure at the very least that the agent can have a board control behavior. Because board control is still a pretty good indicator of "*how much a player is winning in a given state* $s \in \mathcal{S}$" (as defined in Section 3.1.3), we chose to select this measure for the state score function.

What is interesting with this particular scoring scheme is that, in a fully observable scenario, it is symmetric – if a player has a positive board control score, her opponent is going to see the situation with an exactly opposite feeling, which would have the same magnitude than the positive one of the controlling player.

In the real case where the game state is only partially observable, this symmetry is broken. Nevertheless, it is still strongly balanced: for a clear situation of board control by one of the players, the other will still see the situation at his disadvantage. However, the exact symmetry is broken by the fact that nothing prevents him to have a "game changing card", even though seen from the other side he appears in a bad position. Thus, as a consequence of this partial observability, we have to balance the scoring function such that it is more cautious when estimating the level of board control. We did that by tweaking the symmetrical computation, integrating some nuances based on visible data from which we could infer possible hidden threats.

4.4.1 Board Control Strategy Description

Board control is the process of gaining control of the board and preventing any attempt of the opponent to take it. Some *Hearthstone* experts also refer to this strategy as *leading the game tempo*.

Whether a player has control over the board depends of several factors, and this score is a weighted sum relative to their expression. Having the control of the board is a tricky notion to describe, and we will try to state it as clearly as possible in what comes next by enumerating the factors we deemed the most relevant when talking about board control strategies.

Factors Influencing the Board Control Score

Minion advantage. First and simplest factor of all, the more minions a player controls compared to her opponent, the more she controls the board. This factor is pretty straightforward: with more minions than her enemy, she can more easily prevent him from putting minions for himself. Indeed, each time he will play a minion then end his turn, she will be able to easily kill the minion so that her opponent is incapable of attacking with it. If killing the minion was made at the expense of loosing one or even two of hers, because she had more minions than him, she keeps her advantage: by playing a number of minions greater or equal to the number of minions lost in the killing, she will keep her dominant position.

Feature index	**Feature description**	**Definition domain**
0	The number of minions controlled by the current player	$\mathbb{N}$
1	The number of minions controlled by the next player	$\mathbb{N}$
2	The number of minions with **Taunt** controlled by the current player	$\mathbb{N}$
3	The number of minions with **Taunt** controlled by the next player	$\mathbb{N}$
4	The number of cards in the current player's hand	$\mathbb{N}$
5	The number of cards in the next player's hand	$\mathbb{N}$
6	The current player's remaining MP	$\mathbb{N}$
7	The current player's maximum MP	$\mathbb{N}$
8	The maximum MP the next player will have on its next turn	$\mathbb{N}$
9	The current player's remaining HP	$\mathbb{Z}$
10	The next player's remaining HP	$\mathbb{Z}$
$[11, 108]$	14 vectors of length 7 representing the minions on the board	(Table 4.2)
$[109, 109+N[$	Description of the current player's hand; feature $i \in [109, 109+N[$ is the number of cards of ID $(i-109)$ the current player has in her hand	$\mathbb{N}^N$

Table 4.1: State vector representation. N stands for the number of distinct cards in the game, collectible *as well as* non-collectible. The "current player" denotes the player whose turn it is when the state vector is requested, the "next player" her opponent.

Feature index	**Feature description**	**Definition domain**
0	The minion unique ID	$\mathbb{N}$
1	Whether the minion can attack or not	$\{0, 1\}$
2	The minion current HP	$\mathbb{N}$
3	The minion current ATK	$\mathbb{N}$
4	Whether the minion is enchanted or not	$\{0, 1\}$
5	Whether the minion is silenced or not	$\{0, 1\}$
6	Whether the minion has **Taunt** or not	$\{0, 1\}$

Table 4.2: Minion vector representation.

Tough minion advantage. A second factor, more difficult to apprehend but also more determining than the previous one, is how much *tough* – or *strong* – minions (minions with more than 4 HP) she has compared to her opponent. In the rich universe of *Hearthstone*, there are many MP-costly spells that are able to clear a player's board at once. We think for example to the *Flamestrike* spell:

”*Deal 4 damages to all enemy minions.*”

As you see, even though a player may have her side of the board full of minions, if they are all *weak* minions, this is not considered as having the control. There are way too many chances that the adversary will cast a devastating spell that he reserved for this kind of situation where he is overwhelmed by many but weak creatures. As *Hearthstone* experts, we do not count the number of times this happened, and it made us learn that even though having a lot of minions is good, having *strong* ones is much more better – it is for instance better to play one strong minion than three weak ones...!

The next factor considered when evaluating the board control strategy is complementary to this one.

Hand advantage. The real caveat of having many weak minions compared to have less but stronger minions is the *number of cards* you lost playing for nothing. Indeed, the more minions you have on your side of the board, the more your opponent is tempted to kill them all in one deadly blow. You may say that the opponent can not have so much devastating cards... Nevertheless, even if you take the extreme case where he has one of them that he can play, while you have just played seven 1/1 minions, he lost one card while you lost seven. This is linked to the *card advantage* factor which will be presented later.

Moreover, the cards a player has in her hand are very representative of her being in control or not. However, this is a hidden information, thus all we can make here are assumptions, but rather strong assumptions as *Hearthstone* is a pretty well balanced game. We thus assume that the more cards she has in her hand, the more the chances she is able to react to her opponent's tentatives to take control of the board.

In the other way around, if she has very few cards in her hand, say one or two, and if she has less minions (weak or not) than her opponent, she is in a really bad position. Unless she is very fortunate in her next card draws, she will not be able to handle the pressure her adversary will be pleased to exercise, knowing he his leading the tempo of the game. She lost the tempo (or conversely, she let her opponent take it) at the moment she had not enough cards anymore to counter his tentatives to get it.

The number of cards one has compared to her enemy is thus another factor that should be taken into account for evaluating board control. Notice that if the game was fully observable, this factor would be evaluated according to the very cards seen in the opponent's hand rather than just the number of cards. Because this information is hidden, we just assumed that the number of cards in a player's hand was an indicator on the quality of her hand, thus that those variables were correlated at some level.

Trade advantage. We wanted the board control score to also reflect how much executing an action facilitated the possible subsequent killings of enemy minions by a given player. This can be crystallized by what we call *trade advantage*.

Trade advantage is a subset of a more general factor, namely *card advantage*. Card advantage is ultimately an expression of the difference between the rate at which a player gains cards, and the rate at which they lose cards [Gamepedia, 2014a]. For killing a tough minion on the board for example, maybe the opponent will have to use two spell cards, leading to a generation of a “2 for 1” card advantage for its owner.

Unfavorable combat is the principal source of card advantage for the opponent. Because of this, we chose to only evaluate this facet of the card advantage factor to simplify the scoring function computations.

Trading is the action of a player making one or more of his minions attack one of the enemy's. The trade advantage is the card advantage generated (or lost) once the trading is done.

We chose to model the influence of trade advantage in the scoring function by the difference between the number of card points a player *would be able* to make the other lose and the number of card points this player would lose himself when considering (an approximation of) the best trades available.

After a trade, the card points it grants to its owner depend on whether (some of) the attacker(s) and/or the target was killed in the fight. If a minion did not die in a trade, it grants its owner the difference between the HP lost in the fight and the damage dealt to its opponent. On the other hand, if the minion died in the trade, it makes its owner lose a total of card points equal to the sum of its ATK and HP before the fight. In this way, we take into account in the trade results that the dead minion will not be able to attack anymore.

The process of finding the best trade is not trivial, as an exhaustive search would result in evaluating all possible combinations of possible attack orders – we thus designed a heuristic able to find approached solutions. We will however not further describe its functioning nor calculation in this thesis.

Expression of the State Score Function

The state score function is a weighted sum of the scores given above:

$$\begin{aligned} \rho(s) \quad = & \quad 1.75 \times \text{Minion advantage in } s \\ + & \quad 2.50 \times \text{Tough minion advantage in } s \\ + & \quad 1.00 \times \text{Hand advantage in } s \\ + & \quad 3.75 \times \text{Trade advantage in } s. \end{aligned}$$

The weights were set based on expert knowledge.

4.5 Action Representation

In this game, the players choose to play cards (that can have *any* effect on the game, not known in advance), end their turn, make their minions attack, select a character for a targeted spell, deathrattle or battlecry,... which are all *actions*. The thing is, we do not make any assumption on the cards available, nor on the variety of the battlecries, spells and deathrattles.

Because of the diversity of these possible actions, action representation in *Hearthstone* is another challenge. As stated in Section 3.1: Problem Statement, the actions available in this game should be representable as fixed vectors of numbers for the theory to be applicable. Nevertheless, we had here a problem: all actions do not have the same structure. Some need features that are irrelevant for others, so unless wanting to handle missing values and things alike, the vectors structure is not fixed for all actions.

Consequently, we could not directly apply the theory developed in the previous chapter. So, we came up with the idea of a *divide and conquer* approach, breaking this problem into subproblems for which the theory would be applicable. The division would be made on the action space: we separate it in disjoint subspaces where each subspace contains actions of the same type. The key is that all actions belonging to the same subspace would use the same fixed vector representation.

Practically, we had to define as few action vector structures as possible such that these covered the entire action space of *Hearthstone*. We managed to determine three main action groups:

1. playing a card,

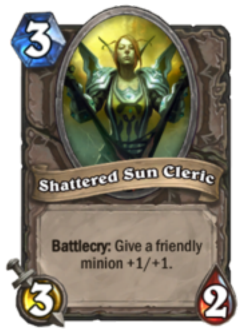


Figure 4.1: The *Shattered Sun Cleric* minion

2. selecting a target,
3. making a minion attack.[1]

Now, this divide and conquer approach means that we will not have one binary classifier for the supervised learning approach but three. We took the liberty of assuming that these action spaces were not related, in which case the theory is still valid. However, these action spaces *are* related, and taking this relation into account would greatly improve the behavior of the resulting agent. This will be discussed more thoroughly in the conclusion of this thesis.

In the following sections, the vector representation of actions in each group will be described. Also, the way the action space was separated will be justified.

4.5.1 Play Cards

Play actions constitute the main type of actions a player might choose to execute. The only problem here is how to handle the variety of cards that can appear in *Hearthstone*. Because we do not make any assumption on the card list available to the game, we can not design a representation that would describe what the card is and does. Of course, you have only minion and spell cards; however some minions might have special abilities, trigger powers, battlecry and deathrattle – describing those using a fixed structure is unrealistic. The same reasoning goes for spells: these can do pretty much everything, thus it is intractable to design a common representation for them.

Nevertheless, by encapsulating what the card is and does in what we call a *meta-feature* – a single feature summarizing all the information about something – we circumvent all the difficulties mentioned above. A card identifier, unique for each card, should be enough: eventually, the agent will learn to understand the meaning of these identifiers and the effects they have on the game environment.

The *Shattered Sun Cleric* minion (Figure 4.1) is an example that made us think to this reasoning. For instance, we thought that the agent will eventually learn that playing the card whose identifier refers to this minion is better when there is another friendly minion on the board; in this way the bonus its battlecry brings is not lost. It should also eventually learn that playing a card with this identifier has more value than one whose identifier refers to a minion with no card text.

This is the kind of reasoning that lead us thinking that summarizing a card by a unique identifier should be sufficient, as long as the agent training was "complete enough" to grasp those subtleties. Hopefully, this approach seems to have payed off (see Section 5.2: Qualitative Experiments).

[1] The attentive reader will notice that the "end turn" action does not fit in any of those sets. We chose to handle the end turn action manually, playing it only when the agent "does not want" to do anything else.

$$\begin{pmatrix} \text{played card ID} \end{pmatrix}$$

Table 4.3: Vector representation of play actions.

Figure 4.2: The *Fireball* spell

Now that we know how to represent a card, the action of playing a card can be represented as a vector of length 1 containing the identifier of the card that is going to be played (Table 4.3).

4.5.2 Selecting Targets

Hearthstone asks the player to select a target when she takes the decision of playing a targeted spell or power (*Fireball* for instance, Figure 4.2). However, it can also ask her to do so in response to an event (for instance, in response to a minion's battlecry). We call the actions requiring such a target selection process, other than attacks, *targeted actions.*

To represent a targeted action instance, at least two pieces of information are required: what the target is and what action will be executed on this target.

First, the target. It is always a character (minion or hero). It will be represented by its unique identifier (the same than the one used for the play actions) and some variables describing its current state: current HP, ATK, whether it is silenced or not, whether it has **Taunt** or not,...

The identifier is required to encapsulate the target's card text meaning. This is done for the same reason than that of the previous section. The current state variables allow the agent to discriminate between targets sharing the same identifier, but whose current characteristics differ. For instance, targeting a damaged, 6 HP *Ironbark Protector* (Figure 4.3) with *Fireball* is not the same action than targeting an undamaged 8 HP one: in the first case the spell shall kill the minion, while in the other the targeted minion shall remain alive.

Now, the action that will be executed on the target is also important. Take the following simple example: the process of selecting a target for the *Fireball* spell is quite different than the one for the *Swipe* (Figure 4.4) spell. Targeted actions thus have to include some information about the action that will be executed on the target.

We encounter here the same difficulties than in the previous section: the effects of a spell, battlecry, deathrattle or triggered power can be pretty much anything. Similarly to what was done for play actions, we chose to encapsulate the meaning of these effects into action identifiers. A unique identifier is given to each and every possible targeted action of the game, with the hope that the agent will eventually learn their meaning during the training.

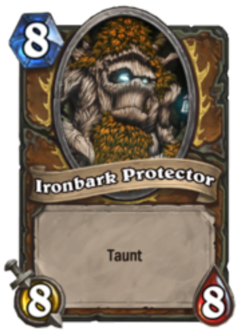


Figure 4.3: The *Ironbark Protector* minion

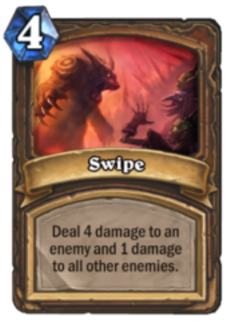

4
Swipe
Deal 4 damage to an enemy and 1 damage to all other enemies.

Figure 4.4: The *Swipe* spell

The detailed vector representation of a targeted action is summarized in Table 4.4.

4.5.3 Make Characters Attack

Attack actions are the last kind of actions in *Hearthstone*. One could wonder why they were not considered as targeted actions, which they are in principle. The answer to this question is simple: the targeted action ID could not summarize the current state of the attacking character. However, this is a crucial information: attacking with a damaged, 1 HP *Ironbark Protector* should not be considered the same action than attacking with an undamaged one!

In fact, almost the same amount of information about the attacker than about the target is required. This is why we chose to have this third and last action subspace. Table 4.5 shows the content of an attack action vector.

$$\begin{pmatrix} \text{Targeted action ID} \\ \text{Target ID} \\ \text{Ally target?} \\ \text{Target HP} \\ \text{Target ATK} \\ \text{Target silenced?} \end{pmatrix}$$

Table 4.4: Vector representation of targeted actions. The variables featuring a question mark are boolean values. The remaining variables are natural numbers.

$$\begin{pmatrix} \text{Attacker ID} \\ \text{Attacker HP} \\ \text{Attacker ATK} \\ \text{Attacker silenced?} \\ \text{Target ID} \\ \text{Ally target?} \\ \text{Target HP} \\ \text{Target ATK} \\ \text{Target silenced?} \end{pmatrix}$$

Table 4.5: Vector representation of attack actions. The variables featuring a question mark are boolean values. The remaining variables are natural numbers.

	$R_\rho(s,a) > 0$	$R_\rho(s,a) < 0$	$R_\rho(s,a) = 0$
Play Actions classifier	400,000	400,000	0
Attack Actions classifier	400,000	400,000	0
Targeted Actions classifier	400,000	400,000	0

Table 4.6: Distribution of samples in the classifiers training sets

4.6 Supervised Learning Classification

All supervised learning algorithms were obtained using the Scikit-Learn library [Pedregosa et al., 2011]. We also used this library to get the ROC and PR curves of our models. The program linked to the *Hearthstone Simulator* library and responsible of testing the agent against random, scripted and human players uses C++/Python bindings in order to interact with the trained models.

Figure 4.5 is an activity diagram of the action selection process for play and attack actions.

4.6.1 Dataset Generation

We wrote a multi-threaded program dynamically linked to the *Hearthstone Simulator* library, able to simulate a large amount of games simultaneously and using the logging capability of the library to generate log files.

We generated huge datasets by simulating 640,000 games with random players playing with random (but legal) decks. The card set however was fixed and contained 56 different cards, among which 40 minions and 16 spells (see Appendix A for the detailed card list). Finally, there is a chance that if they would want to use their special power, this action would not be selected and another one would be taken. This last point is enforced to ensure they do not use it too often, as it is always available for 2 MP, what would bias the samples distribution regarding the frequency at which a given action was taken.

The distribution of positive and negative samples in the sets used to train the classifiers is summarized in Table 4.6. Samples for which the R_ρ function would evaluate to zero were excluded from the training sets.

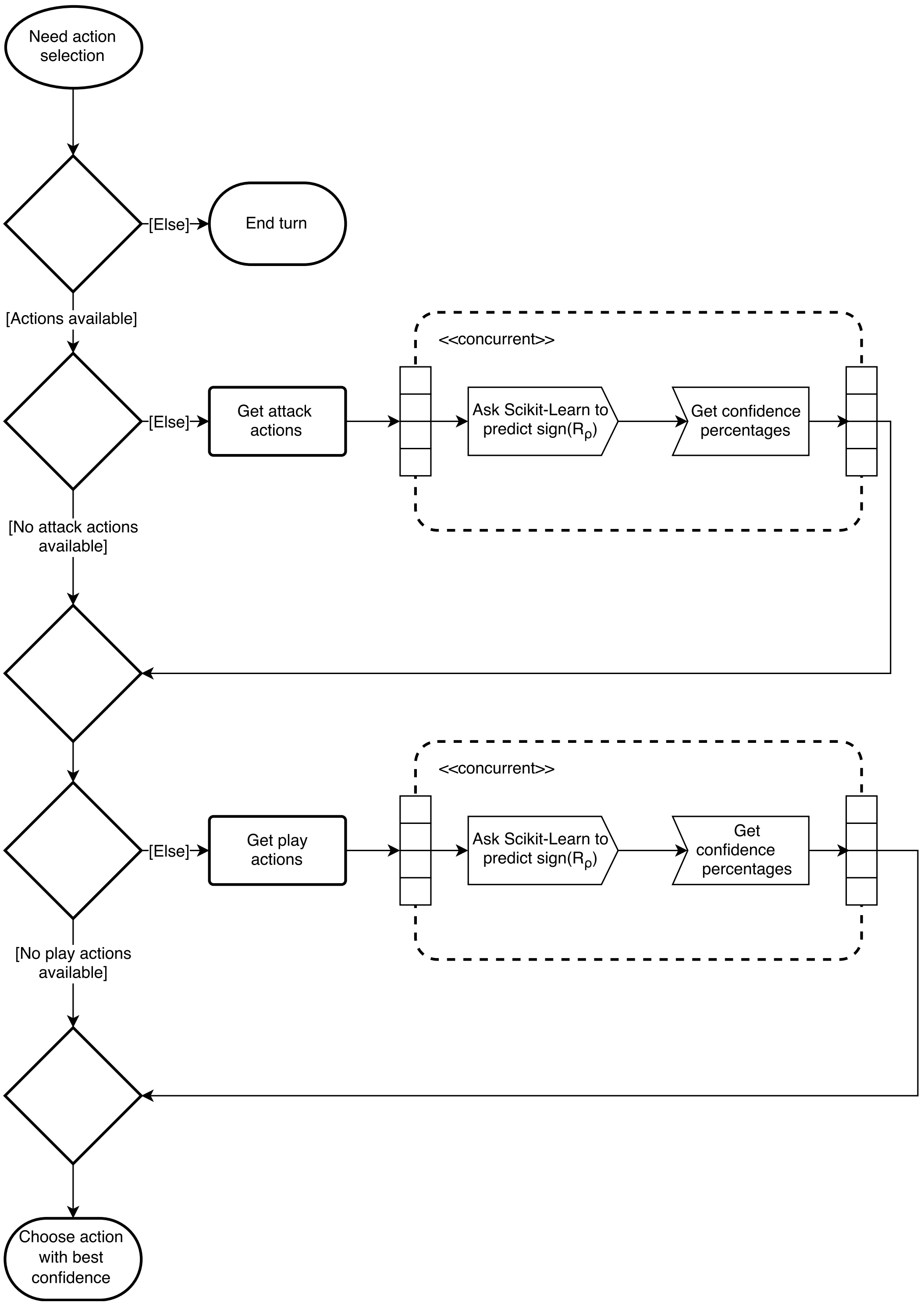

Need action selection
[Else]
End turn
[Actions available]
<<concurrent>>
[Else]
Get attack actions
Ask Scikit-Learn to predict sign(R_ρ)
Get confidence percentages
[No attack actions available]
<<concurrent>>
[Else]
Get play actions
Ask Scikit-Learn to predict sign(R_ρ)
Get confidence percentages
[No play actions available]
Choose action with best confidence

Figure 4.5: The action selection process for play and attack actions

4.6.2 Assessing the Classifiers Performance

As underlined in Section 3.2.3: Assessing the Classifier Performance, it is important to look at both the ROC and PR curves of the three classifiers to evaluate their performances.

Methodology

To get the ROC and PR curves of a trained classifier, one must have a representative test set to evaluate the classifier against. The test set we used was distributed exactly as the training set (50% positive samples, 50% negative samples – see Table 4.6), had the same size, but of course was generated from games with a different random seed than the one used for generating the training set.

Classification Performance

The objective is to maximize the *Area Under the Curve* (AUC) for both the PR and ROC curves [Hanley and McNeil, 1982]. Equivalently, the goal is to have ROC (resp. PR) curves the most in the top left (resp. top right) corner of their graphs.

Figures 4.6 and 4.7 show them, with the objective already well achieved. Indeed, the AUC for both PR and ROC, and for all classifiers, is fairly good for a “first” shot.

As a side note, it should be considered that these curves were obtained with a minimal work on the feature selection and expression and on the board control scoring function design. The same holds for the extremely randomized trees classifiers parameters, that were simply set to the default values recommended by Geurts et al. [2006]. Only a hundred trees were used for each classifier.

For the example, before obtaining those curves, we made some others where the board control scoring function (Section 4.4) did not take into account the *minion advantage* factor. Moreover, the random players used to generate the training and test sets were not enforced to play actions as long as they could, and thus had a chance to prematurely end their turn. The curves we obtained at that time are presented on Figures 4.8 and 4.9.

As you can see, by simply taking into account the minion advantage factor and by tuning a little bit the way the training set was generated, we already made a huge improvement on the “Play Actions” classifier – just by looking at it qualitatively, we see its AUCs are increased in both the ROC and PR graphs.

We had similar results with the addition of a new important feature that we did not included at first, leading to another increase in the AUC of all curves. However, we will not further present those early development curves in this thesis. We just wanted to show that the quality of the classifiers could be improved rather easily by common sense and expert knowledge about the game, without even touching to the extremely randomized trees parameters.

No Thresholding

Recall that the classifier outputs *class affiliation probabilities*, or in other words the percentage of confidence in classifying a sample in the positive class. Thresholding can be used to discriminate between what is considered positive and what is not.

At first, we used the ROC and PR curves to determine the best threshold of confidence such that we avoided having too many judgment errors. However, using thresholding, we never had significant results. We noticed however that the lower the threshold, the higher the average win rate of our agent, the extreme case where the threshold was put to zero being the best. This might seem strange but is in fact logical.

First, in *Hearthstone*, it is *much* worse to lose an opportunity than to mistakenly do something. This already partially explains why the win rate grows with the reduction of

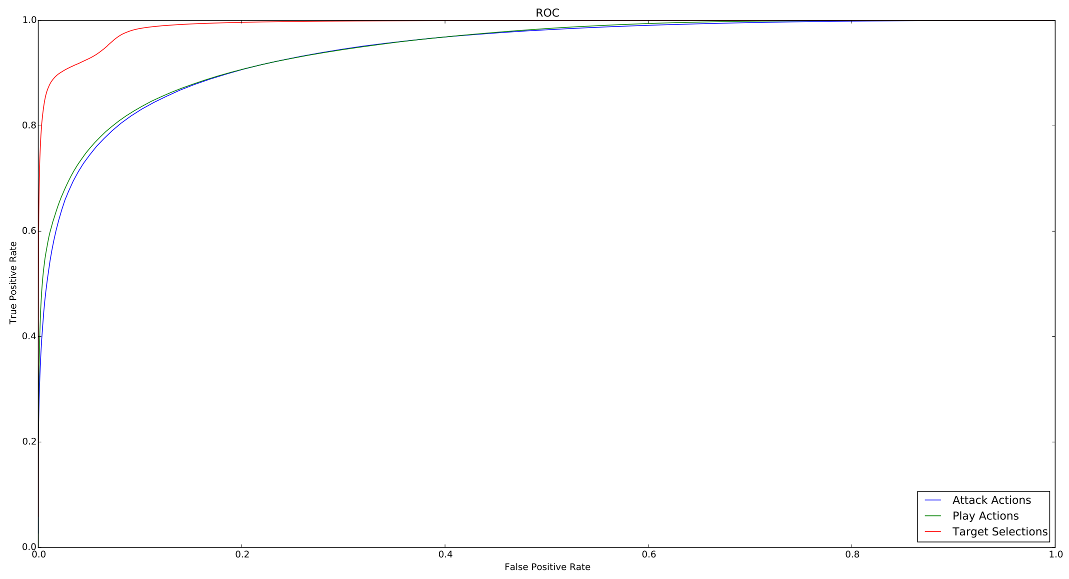


Figure 4.6: ROC curves of the agent (TPR against FPR). Please ignore the concavity change in the knee of the red curve; this is an artifact caused by a small bug in the Scikit-Learn library.

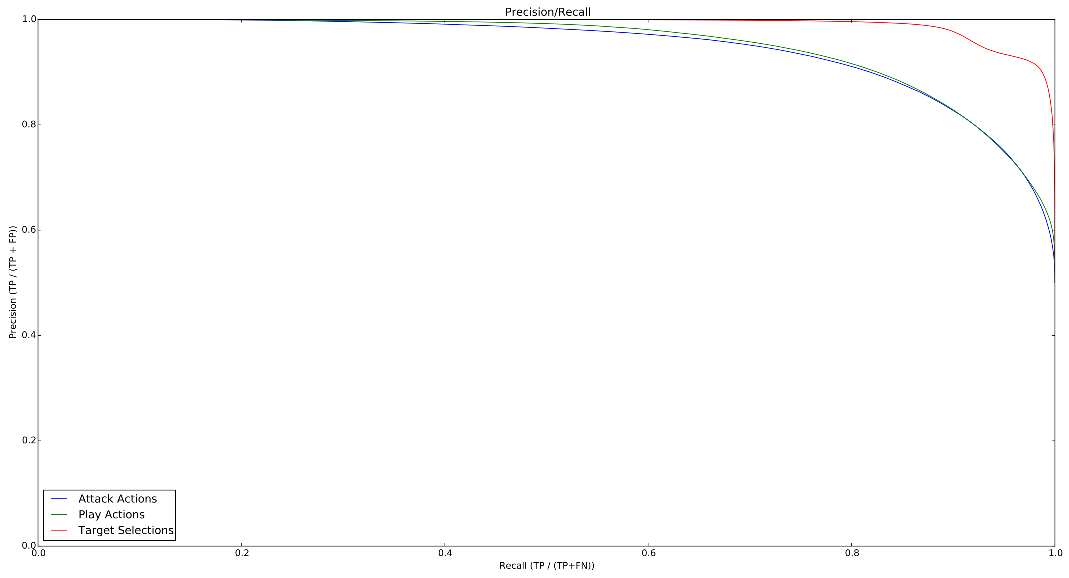


Figure 4.7: PR curves of the agent (precision against recall). Please ignore the concavity change in the knee of the red curve; this is an artifact caused by a small bug in the Scikit-Learn library.

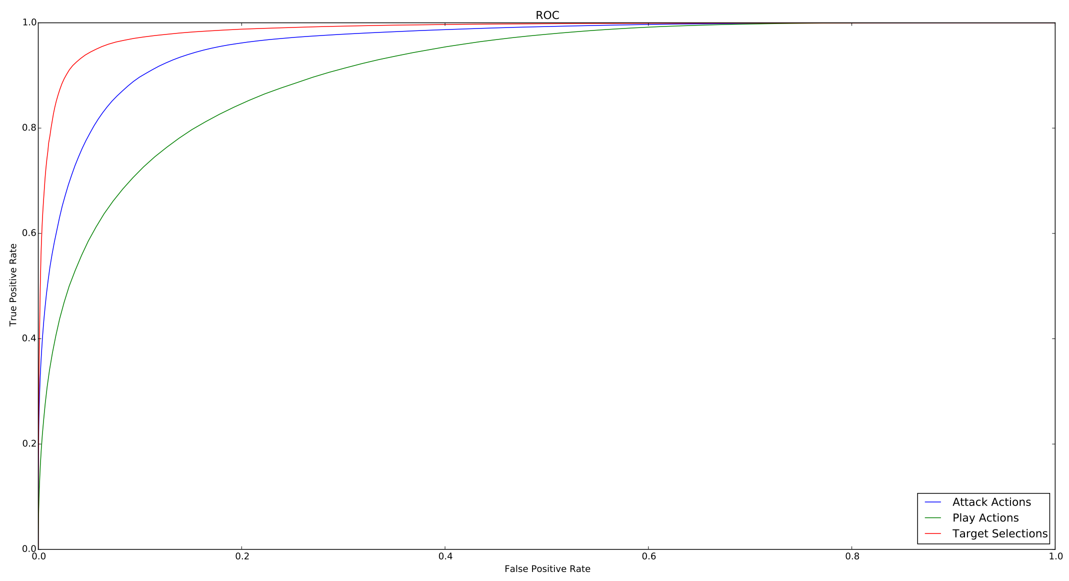


Figure 4.8: ROC curves of the agent (TPR against FPR)

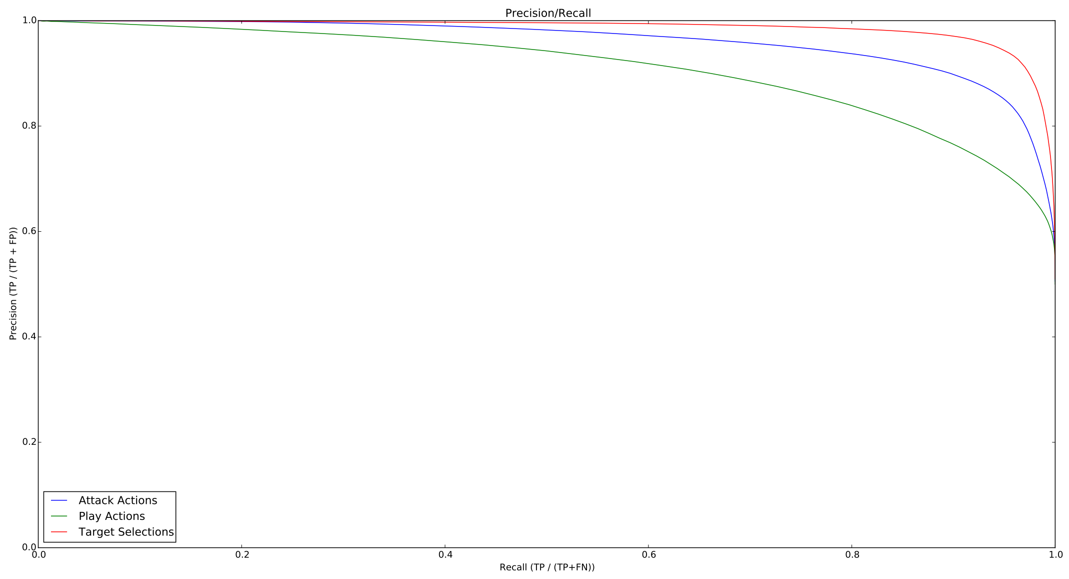


Figure 4.9: PR curves of the agent (precision against recall)

the threshold: with a smaller threshold, the precision (fraction of true positives that we did not miss) is bigger at the expense of a higher false positive rate. The number of missed opportunities thus decreases.

Second, we have to keep in mind that the agent learning is based on a scoring function which does not take into account time steps later than the next one. It would therefore be silly to try to make it hold cards for later turns or prevent it to attack on a turn, as it is incompatible with the scoring function it used to be trained with.

In the light of these two reasons, is is consequently obvious that our application does not need any thresholding.

Memory Usage

Because we did not tune the extremely randomized trees classifiers parameters, those classifiers were built unnecessarily large (approximately 14 gigabytes of RAM are required to hold the three classifiers in memory). Because of this, it is unlikely any player's personal computer will be able to run the agent.

Chapter 5

Results

Summary

This chapter regroups the tests carried on the supervised learning trained agent (called Nora*) and their results. It is divided into two parts: a quantitative experiment and a qualitative one.*

5.1 Quantitative Experiment

The goal of this experiment was to be able to objectively assess whether Nora had learned something from its rough training. This can be done by confronting her to a random player.

5.1.1 Experimental Protocol

Nora was faced against

1. a random player, which plays cards and makes its minions attack as long as it can, choosing to end its turn only when no other actions are available;
2. a scripted player, which implements the strategy of a medium-level player, already familiar with the game mechanics and balance.

Nora and her opponents shared the same deck composition, to prevent lucky decks from biasing the results – see Appendix A for the detailed composition of this deck. For each opponent, 10,000 games were simulated. For the sake of comparison, a random player has also faced the scripted player using the same protocol.

We chose to have a simulation of 10,000 games because we empirically showed that the win rate had converged past this value to something sufficiently accurate for our discussion. Figures 5.1 and 5.2 confirm this behavior, respectively for the random and scripted players.

Finally, to show that the average win rates do not depend on having trained a good classifier set[1] "by luck", we trained four more classifier sets on the same database, with distinct random seeds, all different from the one for which detailed results are given. These four classifier sets are then confronted to the random and scripted players for 10,000 games each (one simulation per classifier set here, as we have shown that it yields a sufficiently accurate average).

[1]The extremely randomized trees classifiers depend on a random seed given at the beginning of their training.

	Win rate	**Loss rate**	**Tie rate**
Simulation 1	92.90%	6.90%	0.20%
Simulation 2	92.57%	7.29%	0.14%
Simulation 3	92.92%	6.96%	0.12%
Simulation 4	92.88%	6.90%	0.22%
Simulation 5	92.72%	7.12%	0.16%
Average	92.798%	7.034%	0.168%

Table 5.1: Results of Nora against the **random** player. Each simulation was made on 10,000 games, with a distinct random seed from a simulation to another.

	Win rate	**Loss rate**	**Tie rate**
Simulation 1	10.70%	89.18%	0.12%
Simulation 2	10.25%	89.55%	0.20%
Simulation 3	10.25%	89.52%	0.23%
Simulation 4	10.82%	89.10%	0.08%
Simulation 5	10.35%	89.46%	0.19%
Average	10.474%	89.362%	0.164%

Table 5.2: Results of Nora against the **scripted** player. Each simulation was made on 10,000 games, with a distinct random seed from a simulation to another.

5.1.2 Win Rate

Figures 5.1 and 5.2 show the results of our intelligent agent against respectively the random and scripted players. The details of the win, loss and tie rates are presented in Tables 5.1 and 5.2.

As a comparison, Table 5.3 gives the win, loss and tie rates of a random player playing against the scripted player. This last table definitely proves that Nora learned to do some "reasonings" the random player does not.

Tables 5.4 and 5.5 finally compare the average win, loss and tie rates for the four additional classifier sets trained with distinct random seeds faced against the random and scripted players respectively. As you may notice, these results are quite similar to those of Tables 5.1 and 5.2, and confirm that they were not dependent on the very classifier set we used.

	Win rate	**Loss rate**	**Tie rate**
Simulation 1	1.01%	98.96%	0.03%
Simulation 2	0.79%	99.19%	0.02%
Simulation 3	1.01%	98.95%	0.04%
Simulation 4	1.01%	98.95%	0.04%
Simulation 5	0.85%	99.10%	0.05%
Average	0.934%	99.03%	0.036%

Table 5.3: Results of the random player against the **scripted** player. Each simulation was made on 10,000 games, with a distinct random seed from a simulation to another.

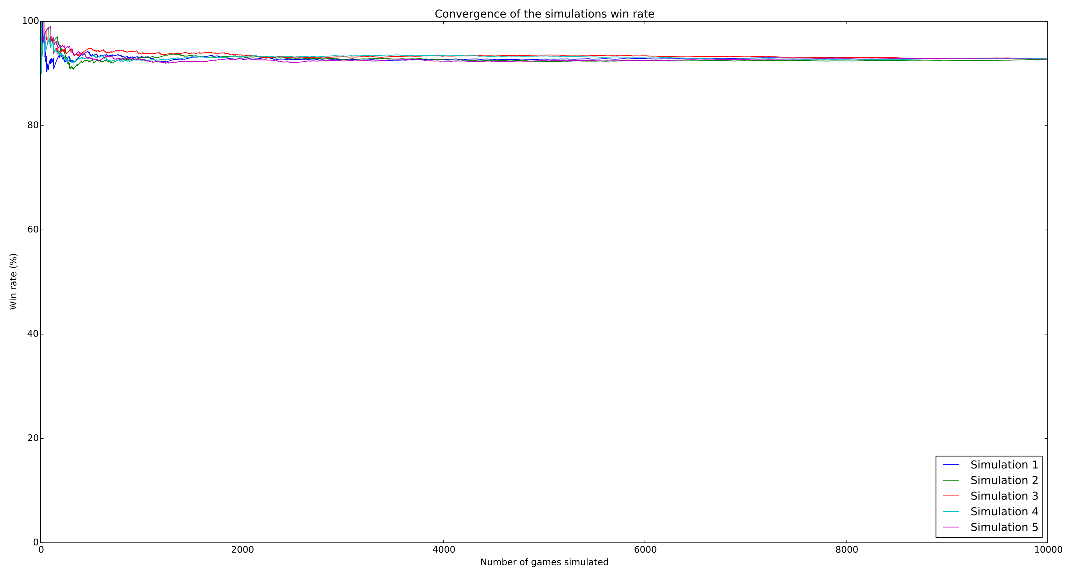


Figure 5.1: Convergence of the simulation win rates of Nora against the random player. Each curve represent the win rate at the different moments of the simulation. Sufficient convergence is attained near 10,000 games.

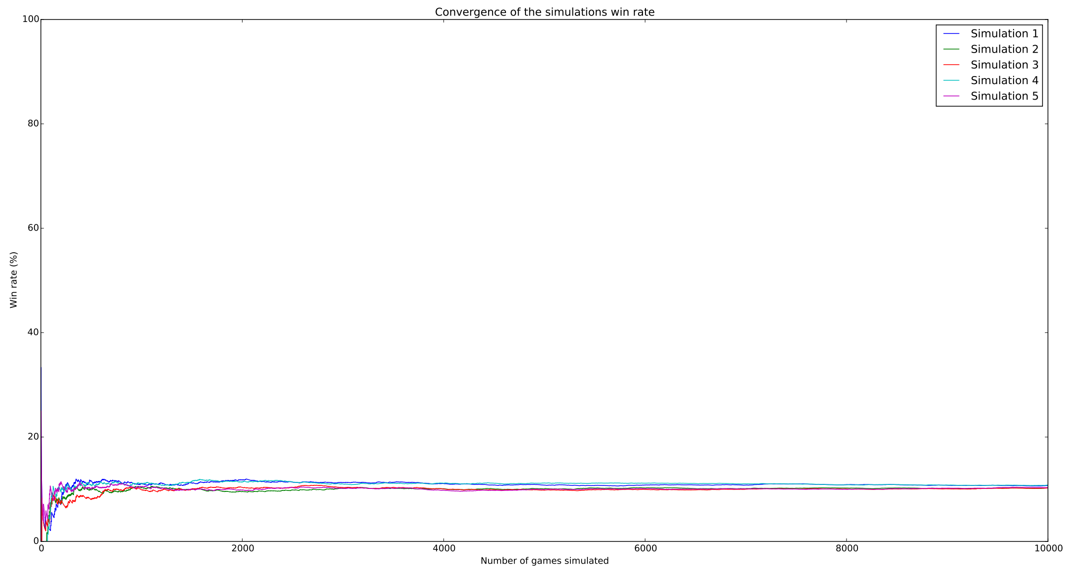


Figure 5.2: Convergence of the simulation win rates of Nora against the scripted player. Each curve represent the win rate at the different moments of the simulation. Sufficient convergence is attained near 10,000 games.

	Win rate	**Loss rate**	**Tie rate**
Random seed 1	93.31%	6.50%	0.19%
Random seed 2	93.45%	6.37%	0.18%
Random seed 3	93.18%	6.68%	0.14%
Random seed 4	92.75%	7.06%	0.19%

Table 5.4: Average results of Nora – trained with different random seeds – against the **random** player. Numbers obtained by simulating over 10,000 games for each classifier set.

	Win rate	**Loss rate**	**Tie rate**
Random seed 1	10.96%	88.87%	0.17%
Random seed 2	10.96%	88.84%	0.20%
Random seed 3	10.45%	89.33%	0.22%
Random seed 4	11.39%	88.34%	0.27%

Table 5.5: Average results of Nora – trained with different random seeds – against the **scripted** player. Numbers obtained by simulating over 10,000 games for each classifier set.

```
Nora: My possible actions:
    Make Chillwind Yeti attack Abomination (David) (Confidence: 16%)
    Play Fireball (Confidence: 52%)
    Play Flamestrike (Confidence: 54%)
    Play Fireblast (Confidence: 70%)
Nora: I choose to Play Fireblast
Nora: I need to choose a target.
Nora: I can target:
    Jaina (Nora) (Confidence: 44%)
    Jaina (David) (Confidence: 49%)
    Boulderfist Ogre (David) (Confidence: 72%)
    Chillwind Yeti (Nora) (Confidence: 38%)
    Chillwind Yeti (David) (Confidence: 48%)
    Abomination (David) (Confidence: 51%)
Nora: I select Boulderfist Ogre (David) and I am 72% confident in this choice.
```

Figure 5.3: An example of entry that can be found in Nora's log. "Jaina" is the name of the hero both player use. The name between parenthesis is the owner of the target.

5.2 Qualitative Experiments

The experiments presented below were conducted by testing Nora against us, analyzing her log to see what "crossed her mind" while playing. An entry in the log consists in the list of actions she can take, with her level of confidence for each, followed by the action she took. An example of such log is given in Figure 5.3.

The qualitative conclusions given in this section constitute a crystallization of our general feeling about Nora. These are no absolute truths, rather some overall characteristics any player could give to Nora if asked to describe her behavior after playing some games against her.

Nora does not harm her minions intentionally. When she needs to select a target for an offensive targeted action, she usually never targets her characters. Evenly, when selecting a target for giving it a bonus, she chooses an ally character most of the time.

Nora is expert when it comes to playing *Fireblast*. This is a perfect example of successful targeted action learning. Indeed, the target selected when Nora played *Fireblast*[2] was usually never an ally except in rare and isolated cases, even though the game allows to select one. Moreover, her selection was often an excellent choice, regarding the current state of the game.

This is especially true for situations where there are minions with huge ATK but 1 HP (for instance, a 8 ATK/1 HP minion): if the board contains such a minion, she would give the *Fireblast* play action a large confidence value and therefore choose to play it. During the

[2]As you might recall, *Fireblast* is the heroes' special power. It costs 2 MP and deals 1 damage to any character of the player's choice. It can be used only once per turn.

Nora's side		4 ATK/2 HP – 38%	
Our side	4 ATK/5 HP – 48%	7 ATK/1 HP – 72%	4 ATK/4 HP – 51%

Figure 5.4: Situation of the board. The percentages represent how confident Nora is in selecting one or the other target for its *Fireblast*.

target selection process, the possible targets confidence values were often low for all minions and much higher for the enemy ones that could be killable by the special power. Figure 5.4 shows an example of such situation.

It seems Nora learned by herself, based on the identifier of the *Fireblast* targeted action, that this action deals exactly 1 damage or at least, that it deals few damages. Indeed, she almost never tries to use it on a creature with more than 1 HP, unless there is no other choice. The more the HP, the less she seems confident in targeting the character. This gives us the feeling that she did manage to extract some of the targeted actions meaning from their identifiers.

It is not a surprise that she has a pretty accurate behavior with this targeted action. As this action was always available (once per turn) for the random players that helped creating the training dataset, it has statistically more chances to appear in the dataset tuples along with all sorts of game environments, more than other targeted actions. Because of this, there is more information related to this targeted action than to others in the training set, leading to Nora having a better understanding of this action compared to others.

Nora exhibits traces of board control behaviors. As you might have noticed with the two first observations, Nora has learned some ways of acquiring the board control. When the *Hearthstone* experts who helped us design the ρ function analyzed her logs while playing against her, they were pleased to see that she was showing board control behaviors. They usually said that, according to the limited knowledge put into the ρ function and what she could learn from the R_ρ objective, she was doing as they expected.

Chapter 6

Conclusion & Future Work

6.1 Conclusion

In this thesis, we presented a generic approach to design intelligent agents for complex video games. This approach has been successfully applied to *Hearthstone: Heroes of WarCraft*, a popular two-player and partially observable card game. The results were encouraging, just as much quantitatively as qualitatively. They clearly showed that Nora, the agent designed following our theory, had learned to apply some strategies a random player did not. Indeed, Nora wins approximately 93% of the time against a random player, and 10% of the time against a medium-level scripted player. The latter result might seem poor, but it has to be compared to the win rate a random player obtains against the same opponent, which is less than 1%. This confirms that Nora succeeded in extracting valuable moves in all states, from a data set composed of random moves. From a data mining perspective, this result is fairly encouraging.

However, by applying the developed theory, we noticed two main weaknesses:

Memory usage. The goal to have an agent able to fit in any player's personal computer was not attained, because of the large amount of memory the trained models need. Memory is not theoretically bounded by the approach, and we showed with a counterexample – our application to *Hearthstone* – that it effectively is so.

Representations and state score function. For complex games, the process of defining the representation of actions and states can be tough. Defining the state score function (that represents the likelihood of a player to win) is even harder and requires much expert knowledge.

In order to apply our theory to *Hearthstone: Heroes of WarCraft*, we implemented a modular and extensible simulator of this game, which is able to let researchers play against their agents, or let agents play against themselves.

JARS, a card representation language we designed, was also introduced. This language allows the library users to define their own cards in a user-friendly way, without having to recompile anything.

The simulator features an almost complete version of *Hearthstone*, where only a fraction of the game mechanics were left aside as future development work. At the start of this work, there were no viable alternatives so the implementation of this simulator was a requirement. Therefore, we designed this simulator with a long-term vision, to allow further research work based on *Hearthstone*. However, we advise researchers interested in designing autonomous agents for *Hearthstone* to use the *Fireplace* open-source project [Leclanche, 2015], which is a full-fledged *Hearthstone* clone coded in Python, and which recently released a stable build.

6.2 Future Work

6.2.1 Improving the Application to *Hearthstone*

The way we created the dataset Nora is trained on is not perfect. The problem is linked to the fact that Nora consists in three classifiers, one for each type of actions, especially targeted actions and play actions. Because of this, it means that the process of playing a targeted spell like *Fireball: Deal 6 damages to a character* is decoupled between the process of playing the card and the one of selecting the spell target. Therefore, the value of R_ρ predicted by Nora is sullied by errors. Indeed, as the dataset was created by random players, she learned to predict the value of R_ρ when playing the *Fireball* spell card *based on random target selection*! As the policy she effectively applies is the result of another classifier and not random selection, the way she predicts the value of play actions that need target selection is **wrong**: she constantly underestimate the value of playing such actions, as on average the random player selects worse targets than her.

This behavior can easily be fixed by a two-step training. First, we would create a dataset as before, by letting random players play. Based on this dataset, we would train the classifier responsible of the target selection process. Then, we would create a second dataset, by letting *semi*-random players play: as usual, they would select at random actions to take, but when it comes to target selection, would use the classifier from the previous step in place of the random selection policy. This way, the *play actions* classifier would be trained on data where the target selection policy is the same than the one that will be used.

Note that the *attack actions* classifier is independent from the others, and thus does not suffer from this caveat. It can therefore be trained either on the first dataset or the second, without distinction.

Unfortunately, we did not have the time to constitute the required datasets, so this improvement is left as future work.

6.2.2 Using Fitted Q Iteration to Integrate Long-Term Information in R_ρ Predictions

Originally, we thought of another approach than the one solely based on supervised learning and binary classifiers, which is based on the Fitted-Q Iteration (FQI) algorithm [Ernst et al., 2005]. Unfortunately, we had no time to implement it and get the results thus it was left as future work. Let us however explain the idea.

Instead of only predicting whether R_ρ will be positive or negative, we want to predict something based on R_ρ that integrates some long-term nuances, such that the predicted values take into account future possible actions and their foreseen rewards. Consequently, the predicted values based on which the agent will decide what action to take do not approximate the R_ρ function itself, but rather a discounted cumulative sum of subsequent values of R_ρ that are sensed to be granted when choosing an action.

Fitted Q Iteration

The FQI algorithm [Ernst et al., 2005] was one of the first batch mode reinforcement learning algorithms to be published. It is a very popular algorithm mainly because of its outstanding inference performances [Fonteneau, 2011]. As such, there are many successful applications of FQI in various domains – robotics [Riedmiller et al., 2009; Lange and Riedmiller, 2010], power systems [Ernst et al., 2009], image processing [Ernst et al., 2006], water reservoir optimization [Castelletti et al., 2010] and dynamic treatment regimes [Ernst, 2005] are examples of such domains. Last but not least, this algorithm is the basis of a recent technique [Mnih et al., 2013] developed to learn to play *Atari 2600* arcade games with only their raw pixel streams as input.

Appendix C details the functioning of FQI.

Regression Algorithm

We choose for the regression algorithm $\mathcal{RA}$ required by FQI to work again with extremely randomized trees, because the benefits of using them as presented in Section 3.2.1 still apply. Moreover, it was shown by Ernst [2005] that tree-based ensemble methods were particularly suited for FQI. Indeed, if we keep the tree structures constant at each iteration of FQI, and only the values of the trees leaves are refreshed, convergence of FQI is ensured.

Application to *Hearthstone*

We run into troubles when trying to apply this idea to *Hearthstone*. As we said, actions can not be all represented using the same structure, what lead us to design three classifiers in the final approach based on supervised learning. However, in its reward update process for state s_t, FQI needs to use the (unique) model computed so far to predict values of actions that can be taken in state s_{t+1}. This can not be done, as the actions that can be taken can have up to three different representations!

Moreover, the concept of “available actions at timestep $t+1$” is fuzzy in *Hearthstone*. It is straightforward for play actions, but less clear for target selection actions.

Nevertheless, we came up with the following reasoning:

1. Get a classifier as in the other approach for target selection. The classifier is trained on a dataset created by letting random players play without constraints.
2. Train a regressor through FQI, based on a partially-random dataset: let random players play, but make them use the classifier of step 1 as target selection policy. This regressor will be used to predict the value of playing a given card in a given state. Train another regressor through FQI, based on the same dataset, but applied only to *attack actions*.
3. Define two meta-actions:
   (a) play some card,
   (b) make some character attack.

   Create a dataset by letting players choose at random between these three meta-actions, but using the classifier and FQI regressors created before for selecting the precise action to take. The representation of actions here is the same for the two meta-actions (just their identifier for instance), as the only goal is to find a policy able to switch optimally between these two processes. Actions available in the next state are also well-defined.
4. Train a last FQI regressor based on this dataset.

As you notice, the target selection policy is the same as in the supervised learning approach presented in this work. This is because it performs already quite well, but mostly because it can not be handled as it by FQI. Indeed, defining what target selection actions are available after selecting a target has no sense.

The idea is to use a two-level FQI. By subdividing the action space, we again make the assumption that these are independent. It means that when asked to predict the value of playing a card, the FQI responsible of play actions will not take into account subsequent possible *attack actions*. This also stands the other way around.

By embracing both regressors into a more general one only answering the question “what to do next: play a card or make a character attack?”, we try to artificially link those action spaces back together.

6.2.3 Simulation-Budgeted MCTS

In the introduction of this work, we argued that MCTS approaches were out of scope for large-scale video games, as side-trajectories are difficult to simulate. Even though some attempts to use simulation-based techniques were presented, it was underlined that they circumvent the difficulty by solving only a subproblem of the game they were applied to and not the game itself.

In order to simulate a large-scale video game completely, as there are usually practically no ways to implement some "undo" mechanic for those games, the simulation should be done by cloning the game completely and execute the side-trajectory simulation on this clone. Often, the cloning would be the bottleneck regarding computing performances because much data would have to be copied each time a simulation is required. However, we thought that future work based on an explicitly budgeted MCTS was a good track to follow. By instantiating a "pool" of N games at the start in addition to the true game (and thus allocating memory all at once, which is faster than piece by piece), we could update all N games by taking the actions effectively taken in the true game. When a simulation is required, it would be made on a game from the pool which would be destroyed after its result was retrieved. It would be very interesting to work with a simulation-budgeted MCTS, in order to evaluate the bond between budget and the resulting agent, to find out whether MCTS could practically be applied to large-scale video games.

Appendix A

Card Set

This appendix contains the list of available cards in the figures A.1, A.2 and A.3. We used this card set for generating both the test and training sets in the various steps of this thesis. Random decks were based on it, as well as our predefined deck.

Notice that non-collectible[1] cards like *The Coin* or *Murloc Scout* are not represented here even though the game uses them.

The images from figures A.1, A.2 and A.3 are the property of Blizzard Entertainment. They are used here for illustration purposes only.

Predefined Deck

This deck is the one we used in the simulations for evaluating the win rate of our agent against the random player.

2x Abomination

2x Abusive Sergeant

1x Acolyte of Pain

1x Arcane Explosion

2x Arcane Intellect

1x Arcane Missiles

2x Bloodfen Raptor

1x Bluegill Warrior

2x Boulderfist Ogre

2x Chillwind Yeti

2x Fireball

2x Flamestrike

2x Frostbolt

2x Gurubashi Berserker

1x Ironfur Grizzly

2x Mana Wyrm

2x Senjin Shieldmasta

2x Shattered Sun Cleric

[1]Cards that can not be used when building a deck

Figure A.1: Card Set (1/3) © Blizzard Entertainment

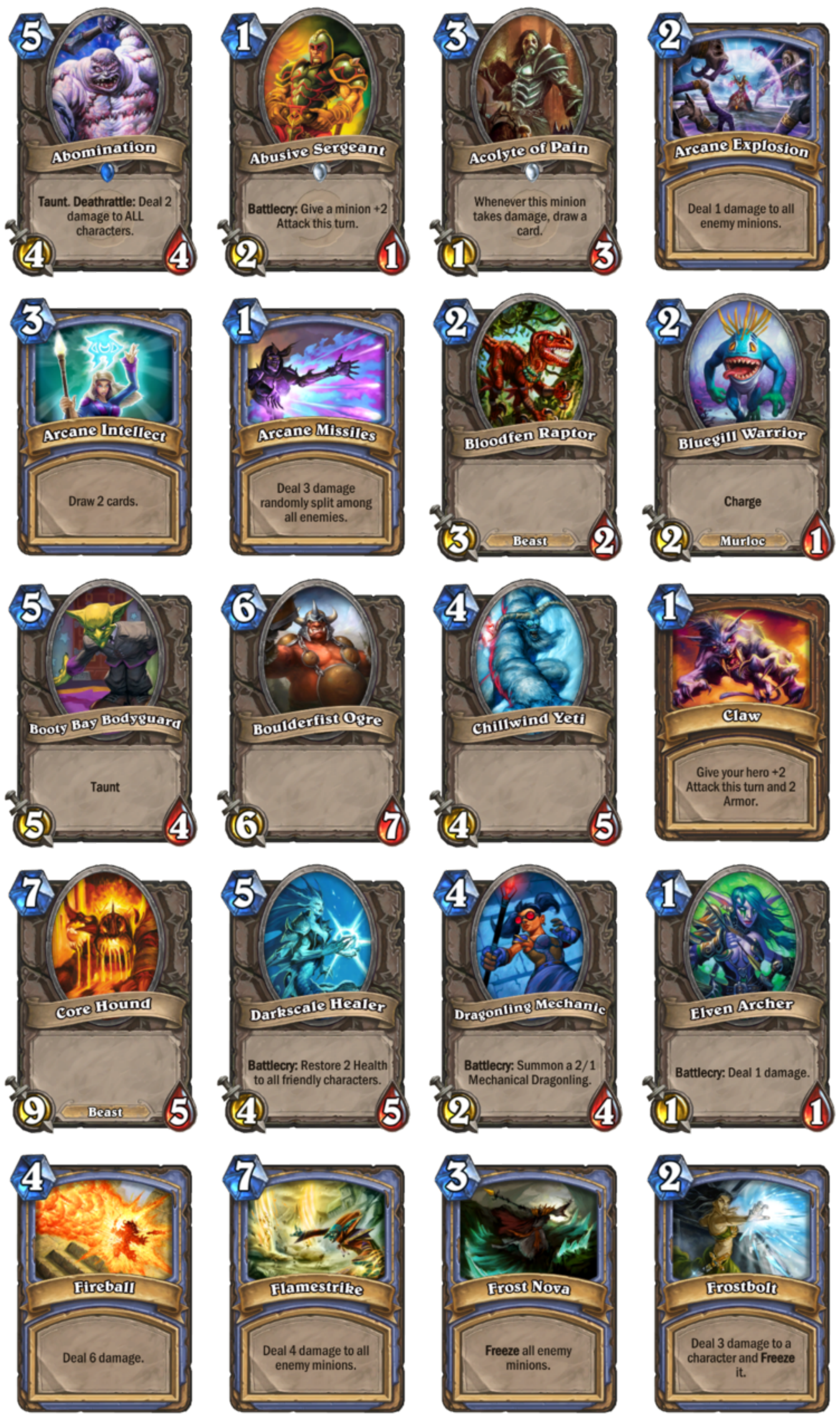

5 Abomination Taunt. Deathrattle: Deal 2 damage to ALL characters. 4 4
1 Abusive Sergeant Battlecry: Give a minion +2 Attack this turn. 2 1
3 Acolyte of Pain Whenever this minion takes damage, draw a card. 1 3
2 Arcane Explosion Deal 1 damage to all enemy minions.
3 Arcane Intellect Draw 2 cards.
1 Arcane Missiles Deal 3 damage randomly split among all enemies.
2 Bloodfen Raptor 3 Beast 2
2 Bluegill Warrior Charge 2 Murloc 1
5 Booty Bay Bodyguard Taunt 5 4
6 Boulderfist Ogre 6 7
4 Chillwind Yeti 4 5
1 Claw Give your hero +2 Attack this turn and 2 Armor.
7 Core Hound 9 Beast 5
5 Darkscale Healer Battlecry: Restore 2 Health to all friendly characters. 4 5
4 Dragonling Mechanic Battlecry: Summon a 2/1 Mechanical Dragonling. 2 4
1 Elven Archer Battlecry: Deal 1 damage. 1 1
4 Fireball Deal 6 damage.
7 Flamestrike Deal 4 damage to all enemy minions.
3 Frost Nova Freeze all enemy minions.
2 Frostbolt Deal 3 damage to a character and Freeze it.

Figure A.2: Card Set (2/3) © Blizzard Entertainment

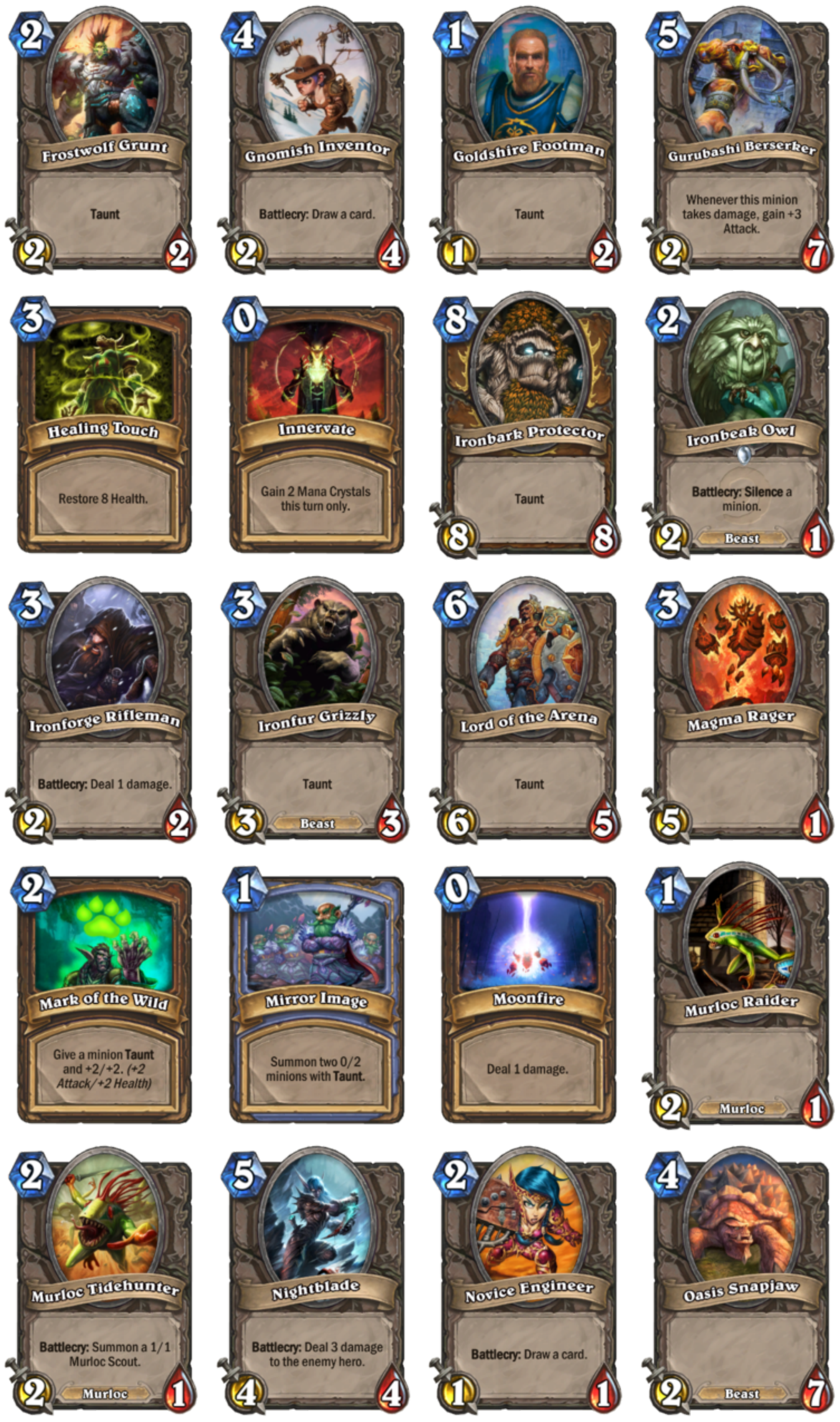
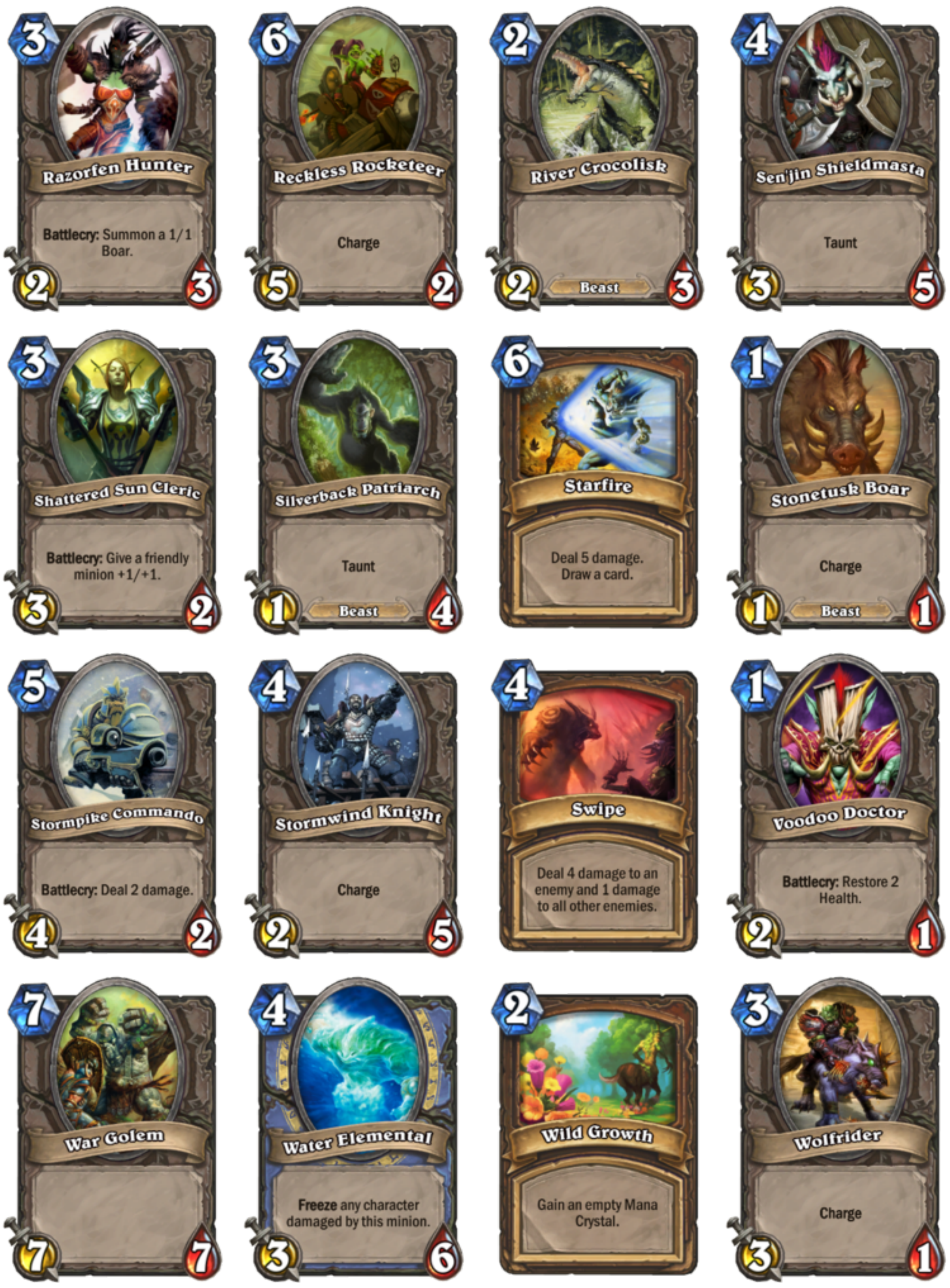

2 Frostwolf Grunt Taunt 2 2
4 Gnomish Inventor Battlecry: Draw a card. 2 4
1 Goldshire Footman Taunt 1 2
5 Gurubashi Berserker Whenever this minion takes damage, gain +3 Attack. 2 7
3 Healing Touch Restore 8 Health.
0 Innervate Gain 2 Mana Crystals this turn only.
8 Ironbark Protector Taunt 8 8
2 Ironbeak Owl Battlecry: Silence a minion. Beast 2 1
3 Ironforge Rifleman Battlecry: Deal 1 damage. 2 2
3 Ironfur Grizzly Taunt Beast 3 3
6 Lord of the Arena Taunt 6 5
3 Magma Rager 5 1
2 Mark of the Wild Give a minion Taunt and +2/+2. (+2 Attack/+2 Health)
1 Mirror Image Summon two 0/2 minions with Taunt.
0 Moonfire Deal 1 damage.
1 Murloc Raider Murloc 2 1
2 Murloc Tidehunter Battlecry: Summon a 1/1 Murloc Scout. Murloc 2 1
5 Nightblade Battlecry: Deal 3 damage to the enemy hero. 4 4
2 Novice Engineer Battlecry: Draw a card. 1 1
4 Oasis Snapjaw Beast 2 7

Figure A.3: Card Set (3/3) © Blizzard Entertainment


3 Razorfen Hunter Battlecry: Summon a 1/1 Boar. 2 3
6 Reckless Rocketeer Charge 5 2
2 River Crocolisk Beast 2 3
4 Sen'jin Shieldmasta Taunt 3 5
3 Shattered Sun Cleric Battlecry: Give a friendly minion +1/+1. 3 2
3 Silverback Patriarch Taunt Beast 1 4
6 Starfire Deal 5 damage. Draw a card.
1 Stonetusk Boar Charge Beast 1 1
5 Stormpike Commando Battlecry: Deal 2 damage. 4 2
4 Stormwind Knight Charge 2 5
4 Swipe Deal 4 damage to an enemy and 1 damage to all other enemies.
1 Voodoo Doctor Battlecry: Restore 2 Health. 2 1
7 War Golem 7 7
4 Water Elemental Freeze any character damaged by this minion. 3 6
2 Wild Growth Gain an empty Mana Crystal.
3 Wolfrider Charge 3 1

Appendix B

Extremely Randomized Trees

This appendix summarizes briefly the extremely randomized trees algorithm. For details about its implementation (impurity reduction measures, default parameters,...), we redirect the reader to the original paper on extremely randomized trees published by Geurts et al. [2006].

The training of this algorithm is done by building $M \in \mathbb{N}_0$ regression (resp. decision) trees and by averaging (resp. outputting the majority class of) their predictions. All trees are individually built based on the complete original training set. To create a test at a node, this algorithm picks $K \in \mathbb{N}_0$ features at random from the feature space and for each of these features selects a cut-point at random. It then computes the "impurity reduction" each of the K tests brings and chooses the test that maximize this reduction. The algorithm stops splitting a node when the number of elements in this node is less than some parameter $n_{\min} \in \mathbb{N}_0$.

The algorithm therefore has three parameters: the number M of trees in the forest, the number K of random tests to evaluate at each node during tree building and the minimal number $n_{\min}$ of elements in a leaf node.

Appendix C

Fitted Q Iteration

This appendix reviews the basis of the Fitted Q Iteration (FQI) algorithm as published by Ernst et al. [2005].

We can see games as systems having discrete-time dynamics described by

$$\tau : \mathcal{S} \times \mathcal{A} \to \mathcal{S} \mid (s_t, a) \mapsto s_{t+1} \text{ with } a \in \mathcal{A}_{s_t}, \quad t = 0, 1, ...$$

We also associate to the transition from t to $t+1$ the instantaneous reward

$$r_t = r(s_t, a_t).$$

Let now

$$h : \mathcal{S} \to \mathcal{A} \mid s \mapsto a \in \mathcal{A}_s$$

be a stationary control policy and $J^h_\infty(s_0)$ denote the expected return obtained over an infinite time horizon when starting from the initial state $s_0 \in \mathcal{S}$ and when using the policy h, or in other words when $\forall t, a_t = h(s_t)$.

Finally, for a given initial condition $s_0 \in \mathcal{S}$, $J^h_\infty(s_0)$ is defined as

$$\forall s_0 \in \mathcal{S}, \quad J^h_\infty(s_0) = \lim_{N\to\infty} \left(\underset{0 \le t < N}{\mathbb{E}} \left[\sum_{t=0}^{N-1} \gamma^t r(s_t, h(s_t)) \right] \right)$$

where $\gamma \in [0, 1[$ is a discount factor that weights short-term rewards more than long-term ones, and where the conditional expectation is taken over all trajectories starting with the initial state s_0.

The goal is to find an optimal stationary policy h^*, i.e. a stationary policy that maximizes $J^h_\infty(s_0)$:

$$h^* \in \underset{h}{\operatorname{argmax}}\, J^h_\infty(s_0).$$

Algorithm 1 details the functioning of FQI. It computes an approximation $\tilde{Q}^*$ of the optimal state-action value function $Q^* : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ from a sample of system transitions

$$\mathcal{F}_n = \left\{ \left(s^l_t, a^l, r^l, s^l_{t+1}\right)^n_{l=1} \right\}$$

where $\forall l \in \{1, ..., n\}$,

$$r^l = r(s_t^l, a^l)$$

and

$$s_{t+1}^l = \tau(s_t^l, a^l).$$

A near-optimal stationary policy $\tilde{h}^*$ can then be derived as follows:

$$\forall s \in \mathcal{S}, \quad \tilde{h}^* = \operatorname*{argmax}_{a \in \mathcal{A}_s} \tilde{Q}^*(s, a).$$

Algorithm 1: The Fitted Q Iteration algorithm

```
Input:
    a set of one-step system transitions $\mathcal{F}_n = \left\{ \left(s_t^l, a^l, r^l, s_{t+1}^l\right)_{l=1}^{n} \right\}$
    a regression algorithm $\mathcal{RA}$
    a discount factor $\gamma \in [0, 1[$
Output: a near-optimal state-action value function from which a near-optimal
        control policy can be derived
/* Initialization                                                          */
1 $N \leftarrow 0$;
2 Let $\tilde{Q}_0$ be equal to 0 all over the state-action space $S \times \mathcal{A}$;
/* Procedure                                                               */
3 while Stopping conditions are not reached do
4     $N \leftarrow N + 1$;
5     Build the dataset $D = \left\{ \left(i^l, o^l\right)_{l=1}^{n} \right\}$ based on the function $\tilde{Q}_{N-1}$ and on the full
      set of one step system transitions $\mathcal{F}_n$:
          $i^l = (s_t^l, a^l)$
          $o^l = r_l + \gamma \max_{a \in \mathcal{A}_{s^l}} \tilde{Q}_{N-1}(s_{t+1}^l, a)$
      ;
6     Use the regression algorithm $\mathcal{RA}$ to infer from $D$ the function $\tilde{Q}_N$:
          $\tilde{Q}_N = \mathcal{RA}(D)$
      ;
7 end
```

Bibliography

Broderick Arneson, Ryan B Hayward, and Philip Henderson. Monte carlo tree search in hex. *Computational Intelligence and AI in Games, IEEE Transactions on*, 2(4):251–258, 2010.

Radha-Krishna Balla and Alan Fern. Uct for tactical assault planning in real-time strategy games. In *IJCAI*, pages 40–45, 2009.

Christian Bauckhage, Christian Thurau, and Gerhard Sagerer. Learning human-like opponent behavior for interactive computer games. In *Pattern Recognition*, pages 148–155. Springer, 2003.

Yngvi Bjornsson and Hilmar Finnsson. Cadiaplayer: A simulation-based general game player. *Computational Intelligence and AI in Games, IEEE Transactions on*, 1(1):4–15, 2009.

Blizzard. Hearthstone: Heroes of warcraft - building the fire, March 2013. URL `https://www.youtube.com/watch?v=vF_PdZybRJE`.

Leo Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):1–43, 2012.

Razvan Bunescu, Ruifang Ge, Rohit J Kate, Edward M Marcotte, Raymond J Mooney, Arun K Ramani, and Yuk Wah Wong. Comparative experiments on learning information extractors for proteins and their interactions. *Artificial intelligence in medicine*, 33(2): 139–155, 2005.

Michael Buro. Orts: A hack-free rts game environment. In *Computers and Games*, pages 280–291. Springer, 2003.

A Castelletti, S Galelli, M Restelli, and R Soncini-Sessa. Tree-based reinforcement learning for optimal water reservoir operation. *Water Resources Research*, 46(9), 2010.

GMJB Chaslot, Jahn-Takeshi Saito, Bruno Bouzy, JWHM Uiterwijk, and H Jaap Van Den Herik. Monte-carlo strategies for computer go. In *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, Namur, Belgium*, pages 83–91. Citeseer, 2006.

Pierre-Arnaud Coquelin and Rémi Munos. Bandit algorithms for tree search. *arXiv preprint cs/0703062*, 2007.

Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *Computers and games*, pages 72–83. Springer, 2007.

Peter I Cowling, Colin D Ward, and Edward J Powley. Ensemble determinization in monte carlo tree search for the imperfect information card game magic: The gathering. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(4):241–257, 2012.

Joseph Bockhorst Mark Craven. Markov networks for detecting overlapping elements in sequence data. *Advances in Neural Information Processing Systems*, 17:193, 2005.

Jesse Davis and Mark Goadrich. The relationship between precision-recall and roc curves. In *Proceedings of the 23rd international conference on Machine learning*, pages 233–240. ACM, 2006.

Demilich. Metastone — hearthstone simulator written in java, April 2015. URL `http://www.demilich.net/metastone/`.

Thomas G Dietterich. Ensemble methods in machine learning. In *Multiple classifier systems*, pages 1–15. Springer, 2000.

Damien Ernst. Selecting concise sets of samples for a reinforcement learning agent. In *3rd International Conference on Computational Intelligence, Robotics and Autonomous Systems (CIRAS 2005)*, 2005.

Damien Ernst, Pierre Geurts, and Louis Wehenkel. Tree-based batch mode reinforcement learning. In *Journal of Machine Learning Research*, pages 503–556, 2005.

Damien Ernst, Raphaël Marée, and Louis Wehenkel. Reinforcement learning with raw image pixels as input state. In *Advances in Machine Vision, Image Processing, and Pattern Analysis*, pages 446–454. Springer, 2006.

Damien Ernst, Mevludin Glavic, Florin Capitanescu, and Louis Wehenkel. Reinforcement learning versus model predictive control: a comparison on a power system problem. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 39(2):517–529, 2009.

Raphael Fonteneau. *Contributions to batch mode reinforcement learning.* PhD thesis, University of Liège, 2011.

Frederik Frandsen, Mikkel Hansen, Henrik Sørensen, Peder Sørensen, Johannes Garm Nielsen, and Jakob Svane Knudsen. *Predicting player strategies in real time strategy games.* PhD thesis, Master's thesis, 2010.

Gamepedia. Card advantage, January 2014a. URL `http://hearthstone.gamepedia.com/Trade`.

Gamepedia. Hearthstone wiki, January 2014b. URL `http://hearthstone.gamepedia.com/`.

Sylvain Gelly and Yizao Wang. Exploration exploitation in go: Uct for monte-carlo go. In *NIPS: Neural Information Processing Systems Conference On-line trading of Exploration and Exploitation Workshop*, 2006.

Quentin Gemine, Firas Safadi, Raphaël Fonteneau, and Damien Ernst. Imitative learning for real-time strategy games. In *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, pages 424–429. IEEE, 2012.

Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine learning*, 63(1):3–42, 2006.

Mark Goadrich, Louis Oliphant, and Jude Shavlik. Learning ensembles of first-order clauses for recall-precision curves: A case study in biomedical information extraction. In *Inductive logic programming*, pages 98–115. Springer, 2004.

Bernard Gorman and Mark Humphrys. Imitative learning of combat behaviours in first-person computer games. *Proceedings of CGAMES*, 2007.

James A Hanley and Barbara J McNeil. The meaning and use of the area under a receiver operating characteristic (roc) curve. *Radiology*, 143(1):29–36, 1982.

Justin Haywald. Hearthstone passes 20 million players, what do you want to see next?, September 2014. URL `http://www.gamespot.com/articles/hearthstone-passes-20-million-players-what-do-you-/1100-6422336`.

Tin Kam Ho. The random subspace method for constructing decision forests. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 20(8):832–844, 1998.

John Laird and Michael VanLent. Human-level ai's killer application: Interactive computer games. *AI magazine*, 22(2):15, 2001.

Sascha Lange and Martin Riedmiller. Deep learning of visual control policies. In *ESANN*. Citeseer, 2010.

Jérôme Leclanche. Fireplace — a hearthstone simulator and implementation, written in python, May 2015. URL `https://github.com/jleclanche/fireplace`.

Chang-Shing Lee, Mei-Hui Wang, Guillaume Chaslot, J-B Hoock, Arpad Rimmel, F Teytaud, Shang-Rong Tsai, Shun-Chin Hsu, and Tzung-Pei Hong. The computational intelligence of mogo revealed in taiwan's computer go tournaments. *Computational Intelligence and AI in Games, IEEE Transactions on*, 1(1):73–89, 2009.

Gilles Louppe and Pierre Geurts. Ensembles on random patches. In *Machine Learning and Knowledge Discovery in Databases*, pages 346–361. Springer, 2012.

Riot Lyte. Detecting and banning extreme intentional feeders, August 2014. URL `http://boards.na.leagueoflegends.com/en/c/miscellaneous/oH6zB1eg-detecting-and-banning-extreme-intentional-feeders`.

Jeffrey Matulef. Destiny has more than 16 million registered users, February 2015. URL `http://www.gamespot.com/articles/hearthstone-passes-20-million-players-what-do-you-/1100-6422336`.

Jean Méhat and Tristan Cazenave. Combining uct and nested monte carlo search for single-player general game playing. *Computational Intelligence and AI in Games, IEEE Transactions on*, 2(4):271–277, 2010.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Oyachai. Hearthsim — generic hearthstone game simulator and ai for testing and understanding the values of various game mechanics and cards, May 2015. URL `https://github.com/oyachai/HearthSim/`.

Amit C Patel and Mia K Markey. Comparison of three-class classification performance metrics: a case study in breast cancer cad. In *Medical imaging*, pages 581–589. International Society for Optics and Photonics, 2005.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Foster J Provost, Tom Fawcett, and Ron Kohavi. The case against accuracy estimation for comparing induction algorithms. In *ICML*, volume 98, pages 445–453, 1998.

Martin Riedmiller, Thomas Gabel, Roland Hafner, and Sascha Lange. Reinforcement learning for robot soccer. *Autonomous Robots*, 27(1):55–73, 2009.

Arpad Rimmel, F Teytaud, Chang-Shing Lee, Shi-Jim Yen, Mei-Hui Wang, and Shang-Rong Tsai. Current frontiers in computer go. *Computational Intelligence and AI in Games, IEEE Transactions on*, 2(4):229–238, 2010.

Firas Safadi, Raphael Fonteneau, and Damien Ernst. Artificial intelligence in video games: Towards a unified framework. *International Journal of Computer Games Technology*, 2015, 2015.

Franisek Sailer, Michael Buro, and Marc Lanctot. Adversarial planning through strategy simulation. In *Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on*, pages 80–87. IEEE, 2007.

Dennis Soemers. Tactical planning using mcts in the game of starcraft1. Master's thesis, Maastricht University, 2014.

Antonio Sutera. Characterization of variable importance measures derived from decision trees. Master's thesis, University of Liège, 2013.

Fabien Teytaud and Olivier Teytaud. Creating an upper-confidence-tree program for havannah. In *Advances in Computer Games*, pages 65–74. Springer, 2010.

Christian Thurau, Christian Bauckhage, and Gerhard Sagerer. Imitation learning at all levels of game-ai. In *Proceedings of the international conference on computer games, artificial intelligence, design and education*, volume 5, 2004.

H Jaap van den Herik. The drosophila revisited. *ICGA journal*, 33(2):65–66, 2010.

Colin D Ward and Peter I Cowling. Monte carlo search applied to card selection in magic: The gathering. In *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, pages 9–16. IEEE, 2009.

Daniel Yule. Hearthbreaker — machine learning and data mining of blizzard's hearthstone: Heroes of warcraft, May 2015. URL `https://github.com/danielyule/hearthbreaker`.

David TARALLA Academic year 2014 – 2015

MASTER THESIS SUBMITTED FOR THE DEGREE OF
MSC IN COMPUTER SCIENCE & ENGINEERING

Learning Artificial Intelligence in Large-Scale Video Games

— A First Case Study with *Hearthstone: Heroes of Warcraft* —

Over the past twenty years, video games have become more and more complex thanks to the emergence of new computing technologies. The challenges players face now involve the simultaneous consideration of many game environment variables – they usually wander in rich 3D environments and have the choice to take numerous actions at any time, and taking an action has combinatorial consequences. However, the artificial intelligence (AI) featured in those games is often not complex enough to feel natural (*human*). Today's AI is still most of the time hard-coded, but as the game environments become increasingly complex, this task becomes exponentially difficult.

To circumvent this issue and come with rich autonomous agents in large-scale video games, many research works already tried and succeeded in making video game AI *learn* instead of *being taught*. This thesis does its bit towards this goal.

In this work, supervised learning classification based on extremely randomized trees is attempted as a solution to the problem of selecting an action amongst the set of available ones in a given state. In particular, we place ourselves in the context where no assumptions are made on the kind of actions available and where action simulations are not possible to find out what consequences these have on the game. This approach is tested on the collectible card game *Hearthstone: Heroes of WarCraft*, for which an easily-extensible simulator was built. Encouraging results were obtained when facing Nora, the resulting Mage agent, against random and scripted (medium-level) Mage players. Furthermore, besides quantitative results, a qualitative experiment showed that the agent successfully learned to exhibit a *board control* behavior without having been explicitly taught to do so.