University of Liège

Faculty of Applied Sciences

Montefiore Institute

Thesis for the Degree of Master in Computer Engineering

Imitative learning for designing intelligent agents for video games

Author:
Quentin Gemine

Supervisor:
Prof. Damien Ernst

Academic Year 2011-2012

Acknowledgements

I would like to acknowledge my supervisor Prof. Damien Ernst who let me pursue one of my main interests by approving the topic of this master's thesis and who gave me crucial advice.

I wish to thank Firas Safadi, PhD student in computer science at the University of Liège, who has been very helpful and assisted me in many ways. His work on the scientific paper resulting of this master's thesis has been very decisive.

I am also grateful to Raphael Fonteneau, postdoctoral researcher at Inria Lille - Nord Europe, for his help with the scientific.

Abstract

Over the past decades, video games have become increasingly popular and complex. Virtual worlds have gone a long way since the first arcades and so have the artificial intelligence (AI) techniques used to control agents in these growing environments. Tasks such as world exploration, constrained pathfinding or team tactics and coordination just to name a few are now default requirements for contemporary video games. However, despite its recent advances, video game AI still lacks the ability to learn. In this work, we attempt to break the barrier between video game AI and machine learning and propose a generic method allowing real-time strategy (RTS) agents to learn production strategies from a set of recorded games using supervised learning. We test this imitative learning approach on the popular RTS title StarCraft II® and successfully teach a Terran agent facing a Protoss opponent new production strategies.

Contents

Chapter 1

Introduction

1.1 Video Games

Video games started emerging about 40 years ago. Their purpose is to bring entertainment to the people by immersing them in virtual worlds. The rules governing a virtual world and dictating how players can interact with objects or with one another are referred to as game mechanics.

The first video games were very simple because they were limited to small 2-dimensional discrete spaces and to only a few mechanics with one or two players at most. Developing agents capable of autonomously playing these games required thus no more than some simple scripted procedures. However, recent video games feature large 3-dimensional spaces, hundreds of mechanics and allow many players and agents to play together. These games seem therefore to be useful for assessing the performance of autonomous agents in complex environments.

1.2 RTS Games

Real-time strategy (RTS) is one of the most complex genre of video games. The game area is a battlefield where - to achieve victory - players have to develop in parallel their economy, military power and technology advancement. A player's economy dictates the gathering rate of his resources which are required for every production or upgrade. A player does not have an infinite amount of resources and he thus has to make compromises to spend them. Investing in economy leads to an increase in the future production capacity but improving the army is required

too to be able to face opponents' attacks and to destroy their bases. Technological development cannot be forgotten because it provides key advantages against less evolved opponents.

Even if the final goal is the military superiority, neglecting either of these three sectors would induce lacks that are often decisive in the race to victory. Players usually try to gain a substantial advantage in military power and technology during a short time window to beat their opponents. As it requires a lot of investment, a player that would not have made enough damage during his time window would most probably fall behind and be overwhelmed in the following of the game. The management of the production and the strategical decisions are part of what is called macromanagement in RTS terminology.

On the other hand, micromanagement refers to operational needs relating to individual units or to small squads. While some units are very strong against others, they can be weak when they have to face specific counters. Positioning units on the battlefield during a fight is therefore a key component of the micromanagement. A player has to move his units precisely and quickly to save damaged ones in order to benefit for as long as possible from their firepower. A lack of focus on unit management could result in a defeat despite a numeric superiority.

As the game evolves in real-time, decisions and executions have to be performed as quickly as possible to avoid wasting precious time. It is one of the more challenging genres because elaborate skills such as multitasking, observation, decision-making and dexterity are required. Recent RTS games are so complex that scientists have acknowledged them as a new tool for the study of cognitive processes [1].

1.3 Bots in RTS Games

When developing agents for RTS games (bots), the common way to deal with the complexity of the environment is to restrict the number of behaviors that the artificial agent can have. Indeed, as each situation and action has to be scripted, limiting the number of possible situations in which the agent can be in is inevitable. Early game is usually handled by choosing randomly between a small set of sequences of production orders that are known to have a decent efficiency against a significant number of strategies. However, some key events cannot be neglected by the agents and a trigger-based approach has to be used in parallel to the sequence of production orders. Some thresholds are defined on measures of relevant game elements in order to trigger a reaction in the agent's behavior. As

every game situation cannot be considered in advance, a lot of information specific to the game has to be discarded to enable a generic processing of these events. As a result, all these abstractions lead to strongly game-dependent mechanisms and a lack of efficiency.

Developing agents in a fully scripted way restricts them to follow only a small subset of possible strategies. As experienced human players can easily recognize which strategy is followed by an opponent if it does not change often enough, the behavior of bots is therefore quite deterministic. Furthermore, new counters are often discovered in such complex games and some strategies become obsolete while others are found. Agents with static behavior are thus quickly outdated and even less challenging over time.

Artificial agents in RTS games are outperformed by the human cognitive abilities. While human players cannot compete with the impressive multi-tasking and dexterity skills of the agents, they still defeat computer players thanks to their strong strategic capacities. In addition to the limited challenge that such agents offer, their consistency is also quite limited as a lot of elements are not taken into account.

1.4 Goal

This work suggests and tests a generic design able of learning to handle all the production orders of an artificial agent for RTS games. A production order can target a building, a unit or a technology which means that the production manager significantly influences the behavior of the agent. Indeed, the production of workers and harvesting structures determines its economy while the construction of production buildings determines its production capacity. Moreover, the production of units defines the size of the army and its composition.

While micromanagement can efficiently be handled by scripted agents with a sufficient number of rules, strategic aspects have still not been efficiently addressed by current bots. This lack is probably due to the absence of any substitute to the human cognitive abilities. Imitative learning based on supervised learning is attempted as such a substitute for production decisions.

For testing purposes, the designed agent is then implemented for the popular RTS title StarCraft II® and the results are discussed.

1.5 Related Work

Lately, the video game industry has attracted substantial research work for the purpose of developing new technologies to boost entertainment and replay value or simply because modern video games have become an alternate, low-cost yet rich environment for assessing machine learning algorithms.

We could distinguish two main goals in video game AI research. Some work aims at creating agents with properties that make them more fun to play with such as human-like behavior. This is usually attempted on games for which agents capable of challenging skilled human players already exist. This is necessary because agents usually manage to rival human players due to unfair advantages such as instant reaction time or perfect aim. These features increase performance at the cost of frustrating human opponents. For more complicated games, agents do not have any chance against skilled human players and improving their performance is the priority. Performance similar to what humans can achieve can therefore be seen as a prerequisite to entertainment. Indeed, facing a too weak or too strong opponent is not usually diverting. This concept is illustrated in Figure 1.1. In both cases, video game AI research advances towards the ultimate goal of mimicking human intelligence. It has actually been suggested that human-level AI can be pursued directly in these new virtual environments [2].

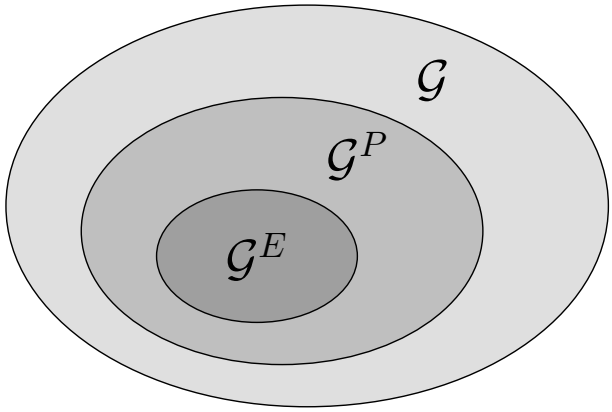


Figure 1.1: Agent set structure for a video games : $\mathcal{G}^E \subset \mathcal{G}^P \subset \mathcal{G}$ where $\mathcal{G}^E$ is the set of agents the player finds entertaining, $\mathcal{G}^P$ is the set of agents that can rival the player's performance and $\mathcal{G}$ is the set of all agents.

Developing an agent with a human-like behavior has been attempted in first-person shooter (FPS) games, most notably the popular and now open-source game Quake II®, using imitative learning. Using clustering by vector quantization to

organize recorded game data and several neural networks, more natural movement behavior as well as switching between movement and aim was achieved in Quake II [3]. Human-like behavior was also approached using dedicated neural networks for handling weapon switching, aiming and firing [4]. Further work discussed the possibility of learning from humans at all levels of the game, including strategy, tactics and reactions [5].

While human-like agent behavior was being pursued, others were more concerned with performance issues in genres like real-time strategy (RTS) where the action space is too large to be thoroughly exploited by generic triggers. Classifiers based on neural networks, Bayesian networks and action trees assisted by quality threshold clustering were successfully used to predict enemy strategies in StarCraft [6]. Case-based reasoning has also been employed to identify strategic situations in Wargus, an open-source Warcraft II® clone [7, 8, 9]. Other works resorted to data mining and evolutionary methods for strategy planning and generation [10, 11]. Non-learning agents were also proposed [12]. By clearly identifying and organizing tasks, architectures allowing incremental learning integration at different levels were developed [13].

Although several different learning algorithms were applied in RTS environments, none were actually used to dictate agent behavior directly. In this work, we use imitative learning to teach a StarCraft II agent to autonomously pass production orders. The created agent building, unit and technology production is entirely governed by the learning algorithm.

1.6 Structure of the Thesis

This thesis is structured as follows. Chapter 2 (Theory) describes how to model the environment of the agent and how to benefit of this representation to use a supervised learning algorithm. The application of the proposed design to StarCraft II is then detailed in Chapter 3 where the chosen learning procedure is also specified. In Chapter 4 (Results), the performance of the agent is analyzed and the similarity between the strategies in the dataset and those that have been performed by the trained agent is measured thanks to clustering. We finally conclude and discuss future work that could enhance the design and its applications (Chapter 5).

Moreover, an appendix describes the implementation details of the agent for Stracraft II and another collects screenshots of the agent playing a game. An archive containing the Matlab® code, the data files used for learning purposes

and the source code of the developed agent can be found at `http://www.gemine.net/tfe.zip`.

Given the innovative aspect of this work and the quite interesting results that were achieved with the proposed design, a scientific paper has been written to present the generic design developed in this work. The paper has been submitted to CIG 2012, a IEEE conference in Computational Intelligence in the video games environment. This paper is joined as an appendix at the end of this document.

Chapter 2

Theory

2.1 Problem Statement

This section presents a way to model as a vector the agent's environment in a RTS game. Starting from the whole game state, this vector is incrementally refined to limit information to that which is interesting for a production manager. The problem of learning production strategies is finally formalized thanks to the modeling that has been established.

2.1.1 State Vector

At a given time, the state of a game can be described by a vector with enough components to contain all the information about the game. We call it a world vector $\boldsymbol{w} \in \mathcal{W}$ with $\mathcal{W}$ the entire game space in which every possible state can be represented. In other words, it should be possible to resume a game at any time considering only the world vector recorded at this time. Given the complexity of current RTS games, such a space is likely to be very large.

However, a player does not usually have access to the entire game state because some information is exclusive to each player. The observation space $\mathcal{O}$ representing the observable information for a player is therefore a subspace of $\mathcal{W}$. We thus define an observation vector $\boldsymbol{o} \in \mathcal{O}$ with $\mathcal{O} \subset \mathcal{W}$. Futhermore, a player cannot always observe all the information he has potentially access to. The current knowledge of a player is actually limited by his vision of the field provided by his units given their limited area of view. Some components of an

observation vector specific to a player may therefore no longer be up to date with the actual game state according to his current knowledge.

Considering the whole observation space does not seem relevant for learning production orders because all the components of this vector are not correlated with production. As we focus on the production management, the state space $\mathcal{S}$ taken into account to learn and predict can therefore be limited to a subspace of the observation space deemed relevant to production. We define the state vector

$$\boldsymbol{s} = (s_1, s_2, ..., s_n)$$

as the projection of $\boldsymbol{o}$ in $\mathcal{S}$.

2.1.2 Production Variables

Some components of the state vector must necessarily relate directly to the buildings and units the player has to produce and to the technologies he has to research. We define these components as natural numbers that indicate the current number of every building type that the player currently has and the cumulative number of each unit type he has produced since the beginning of the game. Components dedicated to technologies are binary variables that indicate whether they have been researched or not.

If the cumulative number is chosen concerning units, it is because the current number would be too much correlated to events unrelated to production. The battles that occur during a game have indeed a high impact on the current number of units. Having only a few units of a given type can be due to a small production or to many losses in battle.

We called these variables *production variables* and we denote them by s_{p_i} to distinguish them among the components s_i of $\boldsymbol{s}$. Considering a state vector that contains $m \in \mathbb{N}$ production variables, we have

$$s_{p_i} \in \mathbb{N} \text{ and } s_{p_i} \in \{s_k\}$$

with $i \in \{1, 2, ..., m\}$ and $\{s_k\}$ the set of the components of $\boldsymbol{s}$.

2.1.3 Variables Relating to Opponents

A number of components of the state vector have to relate to the opponent. As mentioned earlier, some of these components could be inconsistent with the real

state of the game if their value changed since it was last observed by the player. That is why this information should be represented in a robust way.

To reach this goal, the information should be restricted to abstract measures of the opponent state. In this matter, a technological tree seems to be a suitable representation of the knowledge referring to the opponent. Nodes of a technological tree represent a building, unit or technology of the player and it enables to highlight his technological development (Figure 2.1). Indeed, some buildings, units and technologies are usually only available if some other requirements have been built, produced or researched. As there can be several requirements to unlock one node, technological trees are actually directed acyclic graphs.

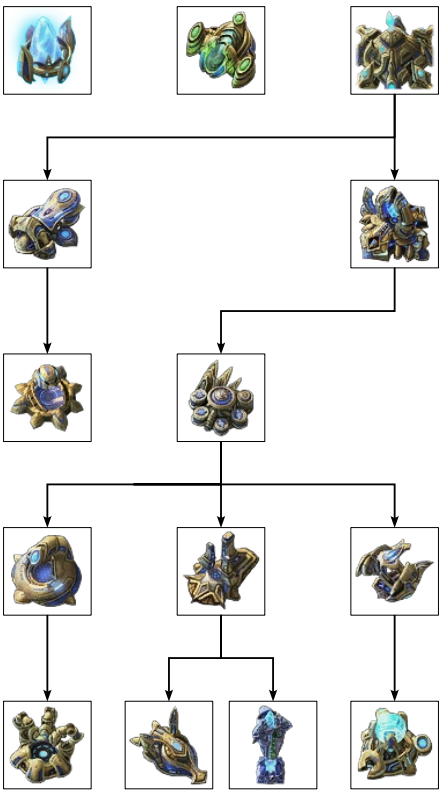


Figure 2.1: Technological tree restrained to buildings for the Protoss race in StarCraft 2.

Such a representation can be stored in a vector by using a component for each node. If a node has been reached by a player, it can be represented by 1 or else by 0. The structure of the acyclic graph does not have to be stored in the state vector because this information does not change during a game.

The technological tree will therefore be updated every time that the player observes a new building, unit or technology belonging to his opponent. As development is all about compromise, knowing the current technological tree gives a good estimation of the strategy followed by the opponent. The time at which a node has been reached is a crucial information because it shows how much a player has spent of his resources in that direction of development. Given that, it is in the agent's interest to collect as frequently as possible information on his opponent.

2.1.4 Other Variables

The remaining components of $\boldsymbol{s}$ correspond to relevant player information. Besides production variables, knowing what is the current gathering rate of each resource is obviously interesting to choose what has to be produced or researched. The time elapsed since the beginning of the game is a crucial information too. Other variables will depend on the specific game for which the agent is designed.

2.1.5 Action Vector

As production variables are the amount of every object in the game relating to production strategy, they are the components of $\boldsymbol{s}$ that have to be modified by the agent's production manager. If the agent wants to produce a new building or unit for which its current amount is the value of s_{p_k}, it will therefore have to perform an action to increment s_{p_k}. If it wants to research a new technology, it will have to set the corresponding production variable to 1.

Considering $m \in \mathbb{N}$ production variables, the production orders can thus be represented by an *action vector*

$$\boldsymbol{a} = (a_1, a_2, ..., a_m)$$

defined in an action space $\mathcal{A}$ which has as many dimensions as there are production variables. Every component a_k of the action vector is actually a natural number that indicates by how much s_{p_k} needs to be increased.

2.1.6 Problem Formalization

The problem of learning production strategies in a RTS game can be formalized as finding a relation

$$P : \mathcal{S} \rightarrow \mathcal{A}$$

that determines an action vector given the state vector of the game. Indeed, if the agent can determine $\boldsymbol{a} = P(\boldsymbol{s})$, it knows how many buildings and units of each type it has to produce and which technologies it has to research. This action vector has to be transformed in actual game actions that result in an update of the production variables within the state vector as follows

$$s_{p_k} \leftarrow s_{p_k} + a_k$$

with $k \in \{1, 2, ..., m\}$.

2.2 Learning Architecture

2.2.1 Hypothesis

We will assume that a set of recorded games is available for the learning process. Each recorded game has to contain state vectors recorded at many different times. These state vectors can result from a state dump at a fixed and short interval which is the same for every recorded game.

2.2.2 Imitative Learning

For the learning of production strategies, we consider that the available games have been played by an expert. No further assumptions are made about the quality of this dataset. The objective is therefore to learn a relation $P : \mathcal{S} \rightarrow \mathcal{A}$ that fits at best the production strategies used in the games.

During a game, a player has to manage consistently every aspect of the game to carry out a strategy. This creates correlation between components of $\boldsymbol{s}$ because fixing the value of some s_i will restrain the possible values of other ones. Knowing this, it might be possible to determine - or at least to estimate - the value of some s_i thanks to $\{s_k | k \neq i\}$. We thus propose a procedure based on supervised learning with only the state vectors to perform imitative learning on the production strategies.

Given a vector

$$\boldsymbol{s}_{-p_j} = (s_1, s_2, ..., s_{p_j-1}, s_{p_j+1}, ..., s_n)$$

which is the state vector without the component s_{p_j}, we define a function P_j such that

$$P_j(\boldsymbol{s}_{-p_j}) = s_{p_j}$$

for every production variable ($j \in \{1, ..., m\}$). The mapping P can therefore be expressed as

$$\begin{aligned} P(\boldsymbol{s}) &= \boldsymbol{a} = (a_1, a_2, ..., a_j, ..., a_m) \\ &= (P_1(\boldsymbol{s}_{-p_1}) - s_{p_1}, P_2(\boldsymbol{s}_{-p_2}) - s_{p_2}, ..., \\ &\quad P_j(\boldsymbol{s}_{-p_j}) - s_{p_j}, ..., P_m(\boldsymbol{s}_{-p_m} - s_{p_m})) \end{aligned}$$

Using this approach, we learn the production strategy of the dataset by learning m P_j functions to estimate the production variables given the remaining state variables. As described in Figure 2.2, each P_j is learned separately using supervised learning.

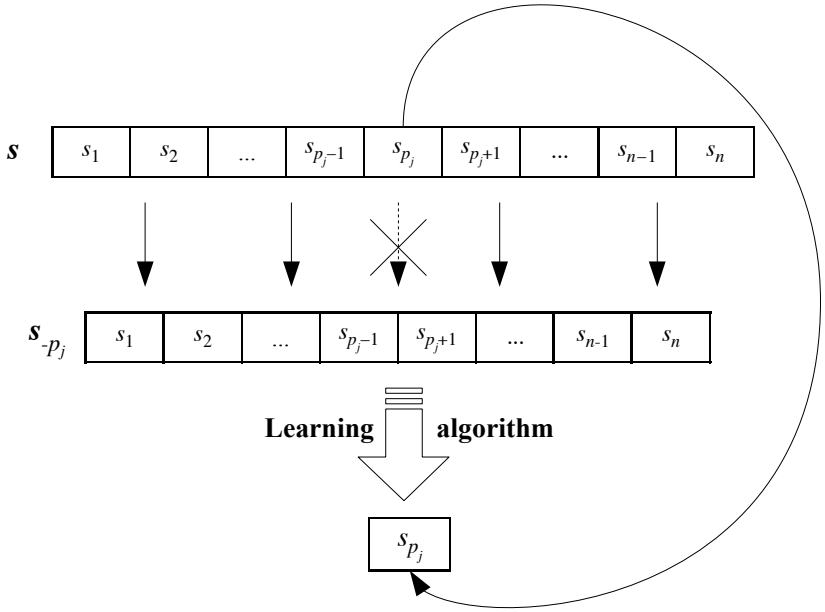


Figure 2.2: Learning the jth model

2.3 Prediction Process

As the value of each a_j indicates how many production orders targeting the jth element are required for the production variable s_{p_j} to become consistent with the remainder of the state vector, non null components of $\hat{\boldsymbol{a}} = \hat{P}(\boldsymbol{s})$ correspond to buildings or units that have to be produced and to technologies that have to be researched.

However, the use of statistical models to represent the learned production strategy $\hat{P}$ means that the computed action vectors are actually estimations. Of course, errors can occur as $\hat{\boldsymbol{a}}$ is an estimation and they can sometimes induce some inconsistencies that have to be handled by the agent. The following sections describe these inconsistencies and how to manage them.

2.3.1 Consistency

Some of the incorrect components of $\hat{\boldsymbol{a}}$ can indicate to produce or to research something that is not available given the current development of the agent's technological tree. To make the prediction more robust, $\hat{\boldsymbol{a}}$ is filtered by setting to zero every non null component relating to an inaccessible target.

2.3.2 Predictability and Resource Constraints

Ideally, if the production strategy was learned perfectly, the amount of resources that the agent owns would always be sufficient to perform every requested production order. Indeed, the dataset was recorded within the same resource constraints than those applied to the agent. In practice, due to errors that occur on some components of $\hat{\boldsymbol{a}}$, the cost of all the requested productions can exceed available resources. This sometimes leads to production orders that can not be performed.

As the behavior of the agent should not be too predictable, the components of $\hat{\boldsymbol{a}}$ should not be treated in the same order for every game. Indeed, if the necessity to build a *building A* is always computed before a *building B*, it would favor *building A* every time the available resources would be insufficient to build both buildings. Of course, the same applies to the units and technologies. A solution to avoid this problem is to randomize the computation order of the components of $\hat{\boldsymbol{a}}$ every time the agent starts a new game.

2.3.3 Distinct Development Paths

Sometimes, when the game state $\boldsymbol{s}$ is not consistent with any strategy from the dataset, there could be several candidate strategies that could be chosen in order to get the state consistent. Since some of these strategies can imply different production orders, the action vector predicted for $\boldsymbol{s}$ might request production orders for several strategies at once. This is why $\boldsymbol{s}$ has to be updated every time a $P_j(\boldsymbol{s})$ is computed and after its value has been translated into a production

order if needed. By doing that, a production order will be taken into account for future predictions as soon as it is computed.

As mentioned earlier, computing the components of the action vector in a fixed order would cause the agent to be quite predictive. Indeed, every time several strategies are available, the production manager would choose the same one. Here again, randomization of the computation order of the components of $\hat{\boldsymbol{a}}$ allows each strategy to be potentially chosen.

Chapter 3

Application

3.1 StarCraft II

Today, StarCraft II®, Blizzard Entertainment's successor to genre patriarch StarCraft®, is one of the top selling RTS games. Featuring a full-fledged game editor, it indisputably is the ideal platform to assess this new learning breed of agents.

Players start at a given location on a map with a main building and six workers. They have to develop and to expand (Figure 3.1) on the map in order to gather more resources in several locations. However, they also must be able to defend their bases and to pressure the opponent to limit his own expansion (Figure 3.2). The usual development of a player is to increase his number of production facilities when his economy grows to be able to spend the harvesting rate of the resources. Indeed, a production capacity greater than the opponent is often a key to victory.

Two resources are available in the game. The first one is *mineral*, it is the main resource in StarCraft II and it can be harvested as soon as a game starts. Mineral is required for almost every production from the most basic unit up to the most technologically advanced research. The second resource available is *vespene gas*, it is required for more advanced buildings, units and technologies. In order to gather vespene gas, a player has to build a refinery on a vespene geyser.

There are three different races in StarCraft II, each involving a unique play style with exclusive units and technologies. The Terrans are masters of adaptation and have learned to survive in the most hostile environments. The Zerg is an alien race and can overwhelm its enemies with massive swarms. Finally, the Protoss are a humanoid species with unmatched individual fighting prowess.

Figure 3.1: A Protoss player just taking a new expansion on the map to harvest more resources.

Figure 3.2: A Terran player (red) using dropships to send units behind enemy lines and hurt the economy of his opponent.

3.2 Game Configuration

In an effort to avoid unnecessary complexity, the agent will be limited to the particular scenario of a one-on-one, Terran versus Protoss matchup type. The matchup is also restricted to the Metalopolis map (Figure 3.3) and the starting locations of the players are fixed in order to limit the amount of data required for learning purposes.

Figure 3.3: Top view of the Metalopolis map of StarCraft II.

3.3 State Vector

In Chapter 2, some instructions were given in order to define a relevant modeling to manage production in an RTS game. This chapter follows these recommendations to build an adequate state vector $\boldsymbol{s}$ for a Terran agent in StarCraft II.

3.3.1 Production Variables

Production variables have to relate to all buildings, units and technologies available for a Terran player. A component defined as a natural number is thus dedicated to every building and unit type. Here are the production variables for a Terran player :

1. $s_{p_{1\to 19}} \in \mathbb{N}$: current number of each building type in the game belonging to the agent.
2. $s_{p_{20\to 31}} \in \mathbb{N}$: cumulative number of units produced since the beginning of the game for every unit type.

3. $s_{p32} \in \mathbb{N}$: current number of Space Construction Vehicles (SCV) belonging to the player. An SCV is used as a worker for a Terran player, it handles the construction of buildings and the harvest of resources. We chose to represent this unit with its current number instead of its cumulative one because it is the main contributor to the agent's economy. If some SCVs were killed by the opponent, they would have to be replaced as quickly as possible and independently of the number produced since the beginning of the game.

4. $s_{p33 \to 59} \in \{0, 1\}$: indicates for each technology if it is researched (1) or not (0).

There are therefore 59 production variables and the action vector must have the same number of components. These production variables are those that will be learned individually given the remainder of the state.

3.3.2 Opponent's Technological Tree

The technological tree of a Protoss player is shown in Figure 3.4. Dotted nodes represent buildings and units that obviously belong to the opponent and that do not need to be in the state vector. There are thus 45 nodes in the tree that require a component in the state vector. It is the number of components relating to the opponent. When a node is reached by a player, the corresponding component will be set to 1. Otherwise the value will stay 0. As specified in the design section, the structure of the tree does not need to be stored in the state vector because it will always remain the same as in Figure 3.4.

3.3.3 Other Relevant Variables

Now that we have defined the variables of the state vector that was clearly specified in the design section, we still need to determine those that we believe are relevant for the production manager. Here is a list of those four variables :

1. $time \in \mathbb{N}$: time expressed in seconds since the start of the game.

2. $population \in \mathbb{N}$: population is a measure provided by the game of the amount of units belonging to a player. Typically, basic units have a value of 1 and this value increases when the unit is more powerful and more expensive. The population value of a Terran unit varies from 1 to 6.

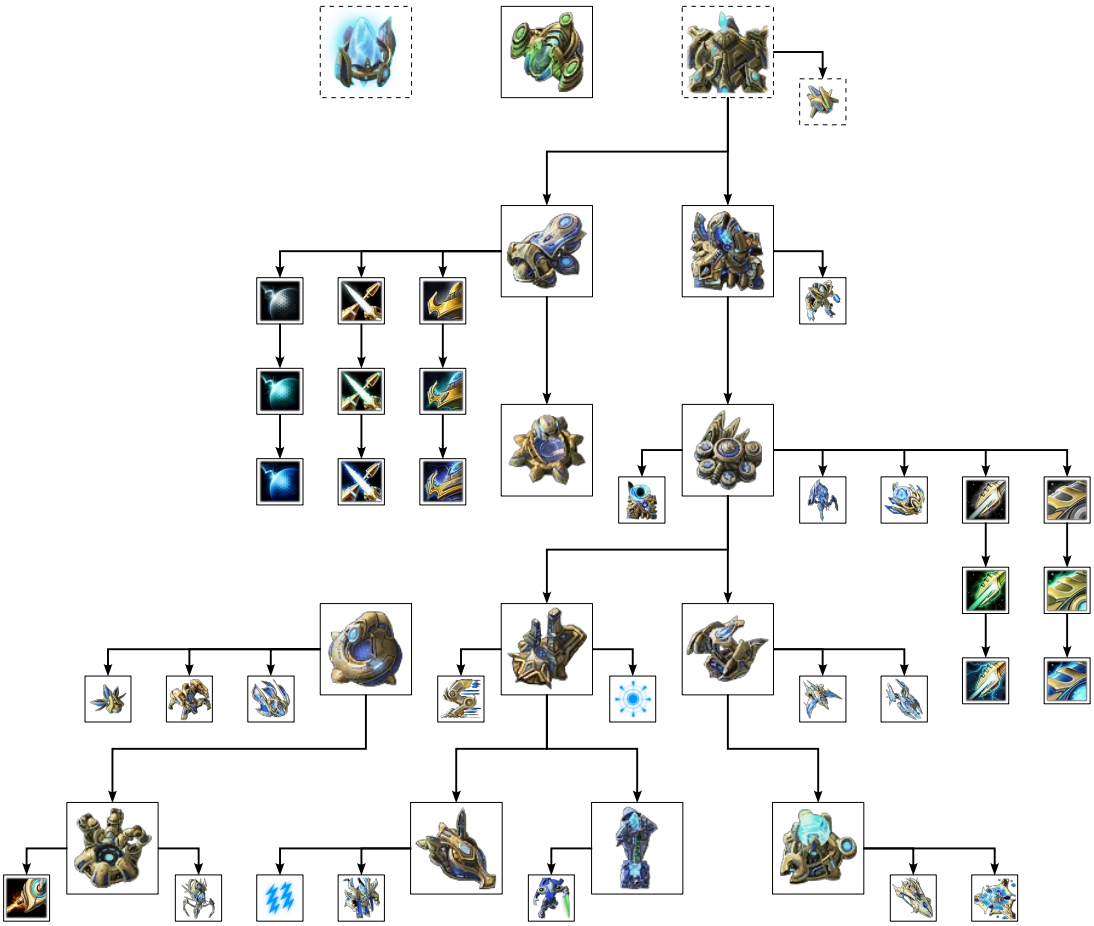


Figure 3.4: Protoss technological tree, large nodes are buildings and small ones are units and technologies.

3. *mineral harvest rate* $\in \mathbb{N}$: amount of mineral harvested by minute by the player.
4. *gas harvest rate* $\in \mathbb{N}$: amount of vespene gas harvested by minute by the player.

3.4 Recording States

The 108-component state vector $\boldsymbol{s}$ has to be updated and recorded frequently in order to gather enough information about games played by an expert. We must

first decide when and how often the variables of the state must be updated. After that, a convenient process to gather the state vectors from the game has to be set up.

3.4.1 State Update

The update frequency of the state vector has to be a compromise between precision, which requires as many updates as possible, and performance. A small break is indeed needed between two refreshes otherwise the game would be slowed down by this process.

Furthermore, a too high update frequency would generate - even for a few games - a huge dataset and require a lot of computational resources to train the agent. We chose a frequency of one refresh every 5 seconds. It is slow enough to allow the game to run smoothly and fast enough to account - on a strategical point of view - for any change in the game.

In addition to update frequency, we have to determine when a change in the game effectively affects the production variables of the state vector. Indeed, when a player gives the order leading to a production or research, it starts a process that will sequentially set the request in several states before reaching its final creation. Those states are the following :

1. *Queued* : a request is queued if its production has not yet started. This happens when an SCV has not yet reached the location where it must build a building but it is moving towards this goal. A unit or a technology request is queued when it is waiting in the queue of a building to be processed because the building is busy producing something else.
2. *In Progress* : state of a request that is being produced. The state of a building request is in progress when an SVC is handling its construction. On the other hand, a unit or a technology is in progress when its dedicated facility is producing it.
3. *Complete* : state of any request once its production is over.

To choose since which state a production order must imply a modification of the state vector, we have to remember that the agent will request a production once the learned production strategy $\hat{P}$ computes an action vector with a non null component. Since an agent's production request has to follow the same steps than a human player's one, the agent must be notified of a production necessity as soon as a human player decides to take such a decision. A unit, building or

technology has therefore to be included in the count of its dedicated production variable as soon as it is in a *queued* state.

3.4.2 Log Files

Unfortunately, StarCraft II does not provide any straightforward way to export data collected during a game. The few mechanisms that are available to store data are provided only to save a few variables and are limited by the size of the saved file, their number and the structure of the data. None of these mechanisms are suited to export state vectors in a convenient way.

However, it is possible to write any string that we want into a freely named log file for debugging purposes. We therefore abuse this service to create a log file for each game and to write the state vectors in it. A line is dedicated for each state vector and components are separated by a space :

$$\begin{array}{cccccc} s_1^a & s_2^a & & . \ . \ . & & s_{108}^a \\ s_1^b & s_2^b & & . \ . \ . & & s_{108}^b \\ . & . & & & & . \\ . & . & & & & . \\ . & . & & & & . \end{array}$$

Such a file format has been chosen because it is very simple and it can easily be imported into Matlab® which will be used for learning purposes.

3.5 Dataset Generation

Of course, there is no dataset already available respecting our requirements. This section describes what the dataset should be in an ideal way and what it really is, due to some unavoidable constraints.

3.5.1 Ideal Dataset

It has already been stated that experienced human players are by far the best StarCraft II players among human and artificial existing players. The dataset should therefore be generated by recording from an expert - or some of them - that is an experienced human player. Ideally, he should even be one of the best

StarCraft II players in the world. Such an expert would indeed be ideal for our two criteria : human-like behavior and amazing skills.

Each time a Stracraft II game is played, a *replay file* is recorded by the game engine. Thanks to the e-sport community, a huge database of recorded games is available on the Internet. Unfortunately, the only purpose of these files is to be played in StarCraft II. They thus only store a sequence of events while ignoring any state information.

Even worse, the file format is proprietary and there is no service provided by Blizzard or tool currently available that lets a third-party developer to be aware of what is actually the game state at a given time. Performing a full reverse engineering of the file format and developing a tool able to compute the game state would be equivalent to rewriting the whole game engine of StarCraft II. This amount of work is inaccessible within the limits of this work, an alternative solution has therefore to be found.

3.5.2 Chosen Dataset

The chosen dataset has to be available or not too hard to generate and relevant to evaluate the level of the agent at the same time. Recording game logs from human players would need a significant amount of time to plan interesting electronic sport events in order to involve enough people to gather a dataset of decent size. Such an organization would require a financial and organizational support that is not accessible for this work.

Our solution to gather a dataset is to use the built-in artificial intelligence of StarCraft II by configuring computer players to generate the data for us. Of course, the level of those agents does not reach our criteria concerning human-like behavior and efficiency but it still allows us to measure the ability of the design to learn production strategies. Moreover, the level of the agents of StarCraft II remains approximately the same for every game which means that it will be easier to evaluate the impact of the learning on the agent's performances.

As the expert should still be better than the average, we will test the design on a dataset of 372 games (64571 state vectors) generated by letting a "very hard" Terran player play against a "hard" Protoss player. The goal of the learned agent will therefore be to beat the "hard" Protoss computer player as much as the "very hard" Terran computer player while following its strategies as closely as possible.

Details concerning the procedure that has been set up to generate the games can be found in Section A.1.

Figure 3.5: Fight opposing a "very hard" Terran computer player (in red) against a "hard" Protoss computer player (in blue).

3.6 Learning Algorithm

This section is about the learning algorithm used to learn the function P. Requirements for learning and predicting production strategies are presented and the chosen solution is discussed on several aspects.

3.6.1 Requirements

For a Terran player in StarCraft II, 59 production variables have to be learned. This number is quite high and we want therefore to avoid the necessity to spend a lot of time on each variable to improve the quality of the learned model. This is why the learning algorithm has to be flexible enough to be able to train decent models for the fifty nine different production variables without requiring specific configuration for each one.

The scripting language of the StarCraft II editor is quite limited. While these limitations will be described in the appendix, we can already state that it will not provide us enough flexibility to allocate and to modify dynamically a data structure. This constraint led us to look for a learning algorithm that computes models as static as possible in their structure. The structure of the model should always be the same and only some parameters should change from one model to

another one. Decision trees, for example, should be avoided because the shape of the tree is not always the same.

The last requirement is of course the computational cost induced by the learning algorithm, both to learn and to predict. The learning should be fast enough to allow us train a lot of different models but it is not the main preoccupation as it has only to be done once. On the other hand, predicting has to be performed in real time during games which is far more crucial. Indeed, the prediction of fifty nine different models have to be computed in a few milliseconds only to avoid slowing down the game which requires most of the available computational resources on its own. Nearest-neighbors approaches are therefore prohibited.

3.6.2 Neural Networks

Given these requirements, going for feedforward neural networks seems quite reasonable. Neural networks are composed of many interconnected units called neurons (Figure 3.6). Every neuron computes a dot product between its own weight vector and an input vector. The result is then used as input of an activation function σ and the final value is sent forward in the network to be one of the inputs of another neuron.

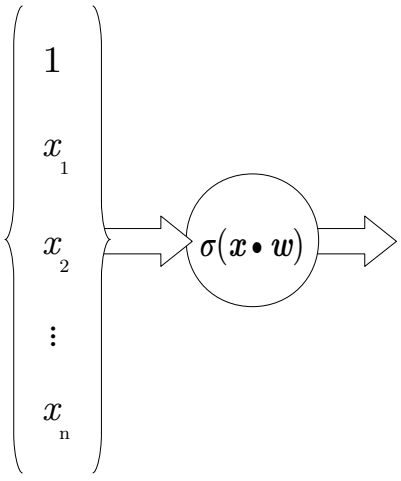


Figure 3.6: An artificial neuron. A bias component is typically added to the input vector, it is the role played by the "1" component in the figure.

Considering a single hidden-layer network (Figure 3.7), the complexity of the

model can easily be adjusted by changing the number of neurons in this layer only. As a static structure is required, the size of this layer should be the same for every model. This size must first be determined to be appropriate with respect to the considered problem. The solution will necessarily be a compromise between all the components of P.

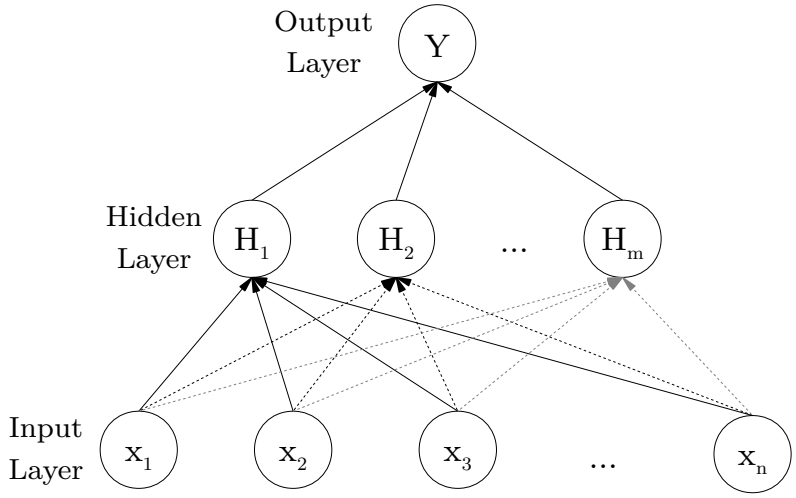


Figure 3.7: Feedforward neural network with a single hidden layer.

By fixing the size of the networks, we fulfill the objective of a static structure. The only variations between two models will indeed be restrained to the weight values. It will thus be possible to implement a unique prediction procedure and to switch between network weights.

The training process minimizes mean square error between predictions and the dataset. As long as the activation functions are all differentiable, the training process can benefit from non-linear optimization techniques. The time dedicated to the learning process is therefore a compromise between the speed and the quality of the solution of the optimization problem.

Concerning computational cost, feedforward neural networks are very fast when performing a prediction as they only compute dot products and the activation function. Of course, the computation time of the activation function has to be reasonable.

3.6.3 Learning Parameters

To train neural networks, we used Matlab® with its Neural Network Toolbox™. The considered neural networks have a 15-neuron hidden layer. This number

is a compromise between the model's complexity, the time required for training purposes and the amount of data to import into the game due to the number of weights.

The activation function used in hidden layer is a typical tangent sigmoid function. This function is defined as

$$tansig(x) = \frac{2}{(1 + e^{-2x})} - 1$$

and it is plotted in Figure 3.8. The identity function is used for the single neuron of the output layer as it is a good choice for regression problems [14].

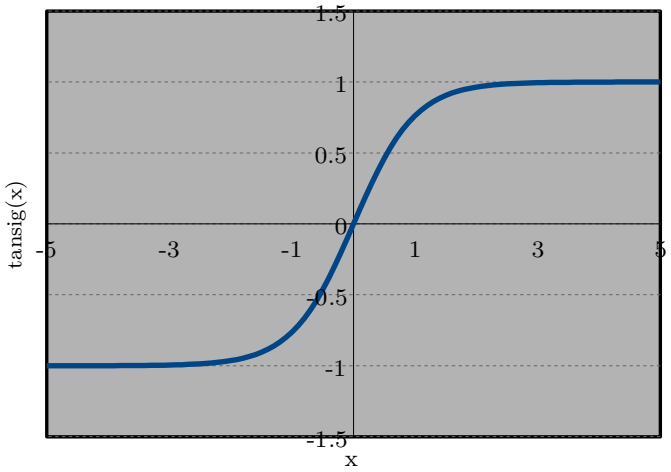


Figure 3.8: Tangent-sigmoid activation function.

3.6.3.1 Scaling

The necessity of scaling inputs of a neural network is theoretically not required. Indeed, bias and weights can scale inputs on their own. However, in some practical situations, scaling can decrease the training time and improve the initialization of the network in order to reduce the probability to be stuck in a local minimum [15]. We thus chose to scale inputs and output to the range [-1;1].

3.6.3.2 Weight Updating

Training a feedforward neural network consists in finding weights and bias values that minimize the mean-square error on the dataset. Non-linear optimization

theory provides many algorithms for that purpose. Choosing one of them is a compromise between convergence speed and solution quality. Indeed, a fast convergence is often associated with a higher probability to stop on a local minimum.

We chose the Levenberg-Marquardt algorithm which adaptively switches between a gradient descent and a Gauss-Newton method. The gradient descent updates weights in the direction that decreases the most the mean square error. Its convergence rate is only linear but it is a good choice if the current iterate is far from the optimum [16]. On the other hand, the Gauss-Newton method assumes that the function to minimize is approximately quadratic. It can give almost a quadratic convergence rate when the iterate is close to the solution [17]. Therefore, the Levenberg-Marquardt algorithm relies on a gradient-descent behavior when far from the solution or on the Gauss-Newton method otherwise.

3.6.4 Model Performance

Errors when predicting can be cause by the limited complexity of the model or by a lack of information within the state vector. Of course, every production variable does not require the same complexity or the same information and some of them may be harder to fit than others. Therefore, the magnitude of the errors varies with the considered production variable. In order to demonstrate this fluctuation, Figure 3.9 counts - for 3 different models - the state vectors from the dataset given the magnitude of the error that is made when predicting.

We can see that the models are very accurate at predicting the number of command centers and the number of barracks required to reach a consistent game state. However, the percentage of wrong predictions is higher for the number of marines and the magnitude of the errors is also greater. It has to be taken into account that the number of marines is quite different than the two previous variables because its value is usually much higher. Moreover, players themselves are often less accurate at producing marines than barracks or command centers because it is not crucial to be as much precise.

3.7 Strategy Classification

In this section, we present a procedure that clusters games given the strategy of a player. It is therefore required to define a similarity measure between two games. It must be taken into account that each game usually does not last the

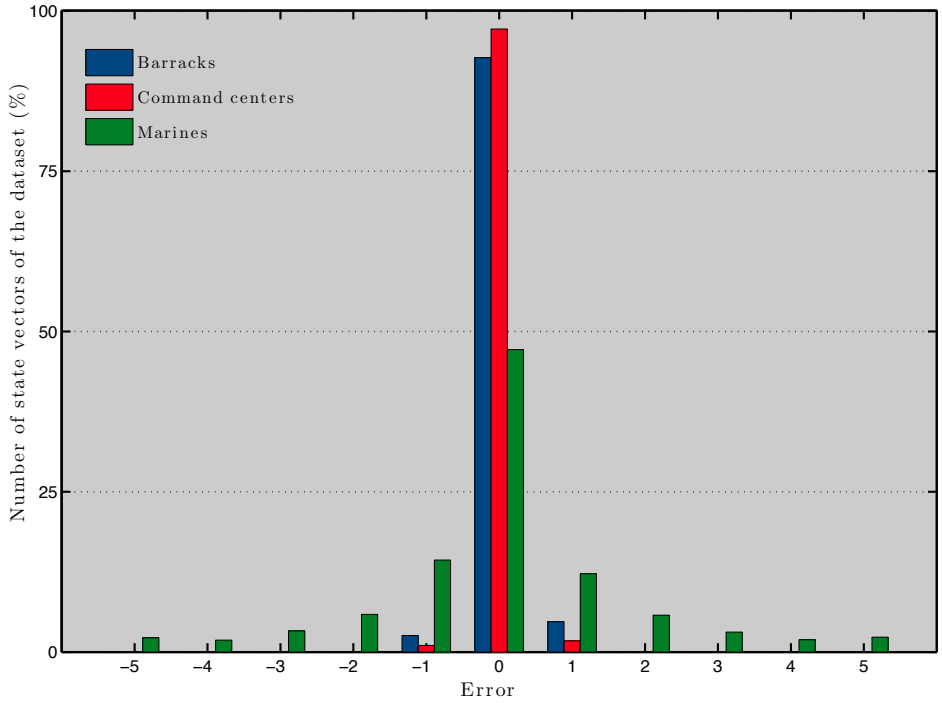


Figure 3.9: Histogram of the number of state vectors of the dataset (%) as a function of the prediction error.

same time than others. Moreover, some of the components of the state vectors in the dataset might not be relevant to determine the similarity between two states.

3.7.1 Build Order

It has been stated that players usually follow in early-game a sequence of production orders known to be efficient for reaching military superiority during a brief time window. Such a sequence of productions is called a *build order* in RTS terminology. Choosing a build order determines the strategy in the beginning of a game but it also restrains what kind of strategies can be followed later in the game. Indeed, every build order favors only some development paths in the technological tree of a player. This limits the player to go later for alternative technologies because corresponding development paths have been neglected.

Therefore, we try to distinguish build orders to be able to cluster games given the overall strategy. By doing so, the problem of the game time variation is solved as build orders relate to early-game only. We chose the first eight minutes of play to distinguish build orders. Indeed, build orders in Stracraft II rarely last longer

than 8 minutes and all the games in the dataset have been played during more than that.

3.7.2 Clustering Space

In order to compute a distance between two states, we need to consider a subspace of the state space $\mathcal{S}$. Indeed, build orders relate only to production orders. We can therefore restrict the state vectors to a subset of the production variables. All the production variables are not required because some of them do not concern early game. We selected 29 relevant variables within a state vector.

Of course, a state space only concerns a game at a fixed time. As build orders are sequences of productions, it seems reasonable to compare games based on the state records of the first eight minutes. To reach this goal, we define a clustering space $\mathcal{C}$ to represent the possible successions of all the state records of the early game. A state vector is recorded every 5 seconds and we select 29 components among every vector during 8 minutes. The dimensionality of the clustering space $\mathcal{C}$ thus amounts to

$$(\frac{8 \times 60}{5} - 1) \times 29 = 2755$$

and each game is represented by a 2755-component vector.

3.7.3 Clustering Algorithm

We went for a hierarchical clustering approach because we do not know the relevant number of clusters in order to distinguish efficiently strategies within a set of games. Ward's linkage has been used to agglomerate clusters : the two merged clusters at each step are those that give the cluster with the smallest inner square distance to its centroid. Therefore, this algorithm consists in minimizing the variance within each cluster which is the objective here. Indeed, if a set of games have a small variance, each clustering vector is close to the mean of all the clustering vectors within the set and we can approximate the build order as being this mean vector.

As we chose to go with Ward's algorithm, the only distance that can be used is the Euclidean distance. However, as we do not want any component of the clustering vectors to have a higher impact than other ones, it is required to standardize components of the clustering vectors on the whole dataset. Every component of the vectors finally used to compute distances between build orders has therefore a zero mean and a standard deviation of 1.

3.7.4 Clustering Procedure

The whole clustering procedure is summarized in the following steps :

1. Each game is loaded as a matrix where lines are the state vectors.
2. The 29 relevant components are selected by keeping the corresponding columns only.
3. Each resulting matrix M is reshaped into a clustering vector $\boldsymbol{c}$ given

$$c_{(i-1)\times 29+j} = M_{i,j}$$

4. The clustering vectors are then standardized for their components to have a zero mean and a standard deviation of 1.
5. The resulting vectors are those finally used to perform clustering thanks to Ward's hierarchical clustering method.

The dendrogram resulting of this procedure applied to the generated dataset is showed in Figure 3.10. Thanks to the observation of figures plotting the mean value and the standard deviation of relevant components of the state vector as function of the time, we chose a threshold corresponding to 2 clusters. This choice is a compromise between clusters with a moderate variance and each cluster representing a significantly different strategy.

This small number of strategies was expected as the dataset was generated by scripted agents that are quite predictive. However, the clustering procedure that has been used only distinguishes strategies and it does not mean that all the games within a cluster are strongly similar. They instead share some important characteristics such as the production facilities that have been built and the army composition.

The two sets of games resulting of this clustering procedure are further discussed in Section 4.3.

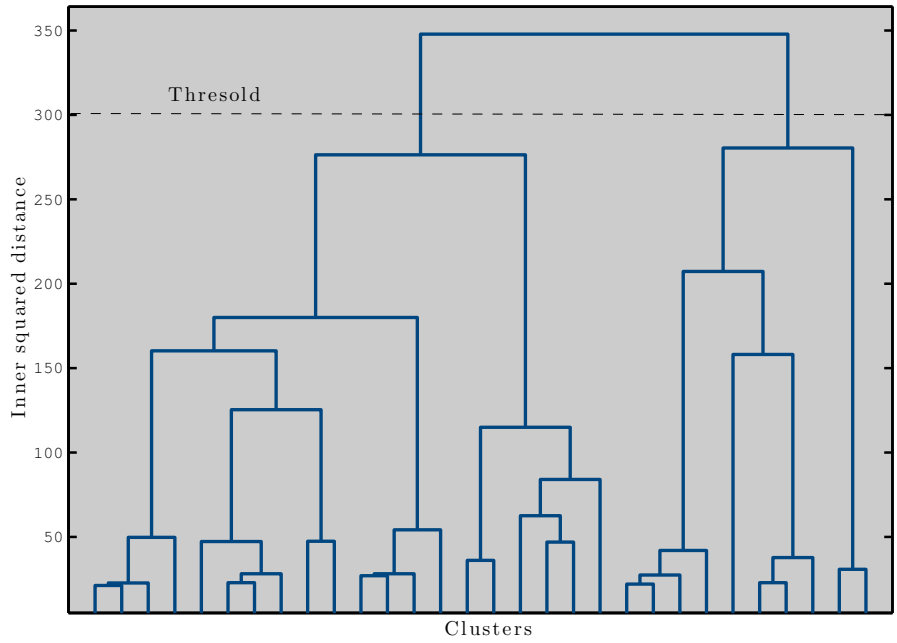


Figure 3.10: Dendrogram computed with the clustering procedure on the set of 372 games.

Chapter 4

Results

4.1 Experimental Protocol

As previously discussed, we chose the "very hard" Terran computer player of StarCraft II to be the expert. As the expert is supposed to be better than an average player, its opponent was the "hard" computer player. To remove unnecessary complexity, the map and the starting locations were fixed. The expert played 372 games with these settings and the *training set* is therefore composed of the corresponding state vectors.

For comparison purposes, the imitative learning trained agent (IML agent) played 50 games against the same opponent : the "hard" Protoss computer player. The resulting state vectors are the *test set*.

The performance of both the expert and IML agent are discussed first. The production strategies used by both agents are then measured and compared to determine if the IML agent correctly learned from the expert's games.

4.2 Win Rate Comparison

Errors when predicting from learned statistical models induce definitely a variation of the production strategies. Even without knowing the magnitude of this variation, it seems appropriate to expect a loss in efficiency resulting from the learning process. Indeed, strategies in the dataset are those deemed relevant by the expert and any barrier to a perfect imitation is likely to decrease the efficiency of the strategies.

The results are summarized in Table 4.1. It is straightforward that the performance loss of the IML agent against the "hard" Protoss player is fairly limited. Indeed, the decrease in efficiency is measured to 6.5% for our test settings. Compared to the initial win rate of 96.5%, the moderate performance loss allows the IML agent to still be much better than the opponent common to both the expert and IML agent. The table contains the results of the "hard" Terran computer player too in order to verify that the good win rate does not come from an imbalance between the Protoss and Terran races. As this last agent always lost, we can thus conclude that the good performances of the IML agent come from the knowledge he learned from the expert.

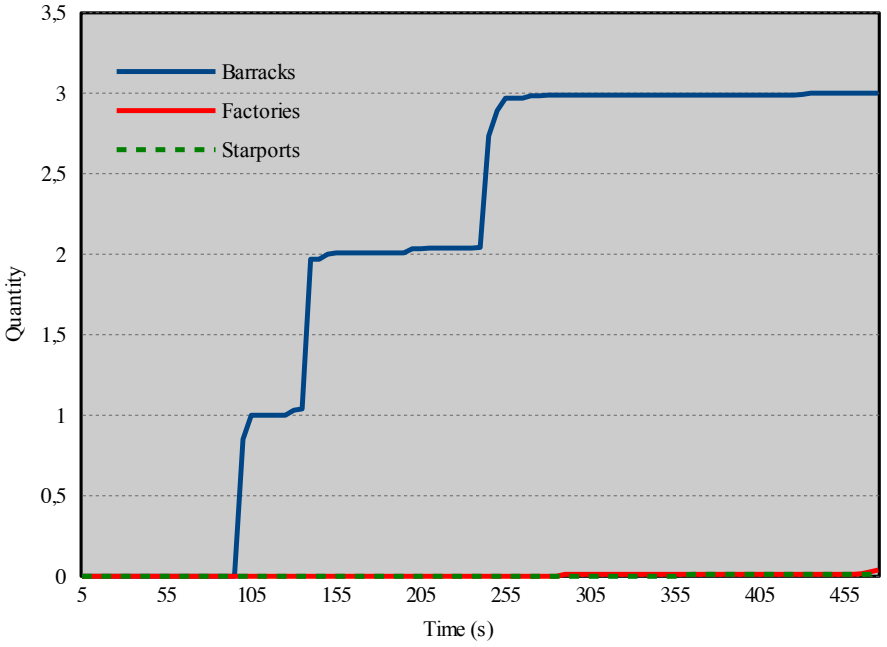
	Terran win rate	Total games
Very hard Terran	96.5%	372
IML agent	**90%**	**50**
Hard Terran	0%	50

Table 4.1: Terran performance against hard Protoss

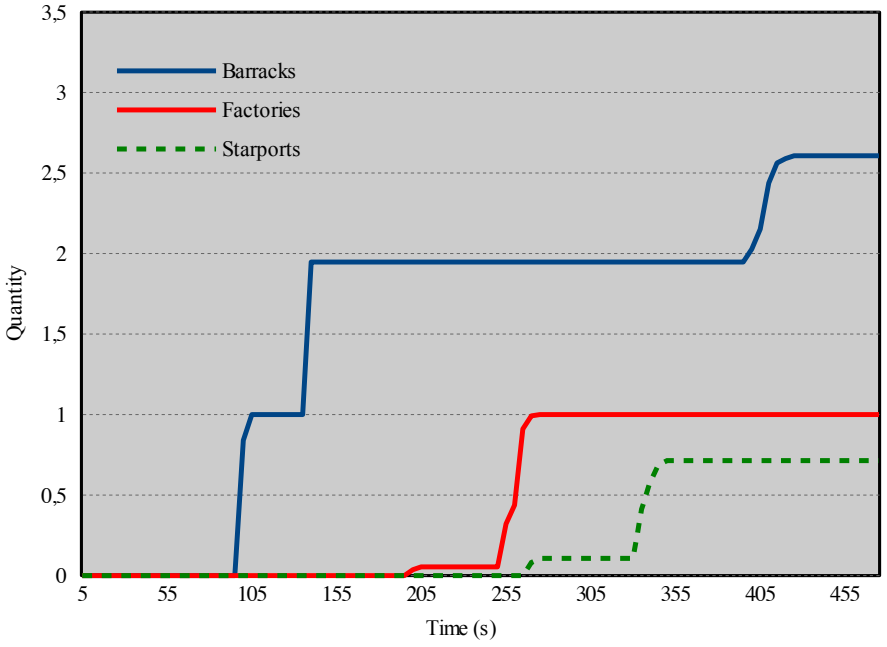
4.3 Strategy Comparison

The procedure presented in Section 3.7 is used to distinguish different strategies within the expert's dataset thanks to clustering. It allows to determine that the expert performs two main strategies. The first one (A) favors infantry at the expense of anything else while the second one (B) tends to go for a faster technological development. In order to support this statement, Figure 4.1 presents the mean number of three building types as a function of the time for both strategies. Strategy A is faster at building barracks but it does not build factories and starports in the first eight minutes of the games. On the other hand, strategy B dedicates some resources to build factories and then starports while it slows down the construction of barracks.

Afterwards, games played by the IML agent are classified within the two existing clusters. The centroid of both clusters is computed to perform this classification. Every new game is classified in the cluster that has the nearest centroid with respect to the Euclidean distance. Figure 4.2 shows - as for the initial clusters - the mean number of barracks, factories and starports as a function of the time for the two sets of games resulting of the classification. Those figures are quite similar to those of the "very hard" Terran computer player (Figure 4.1).



Strategy (A)



Strategy (B)

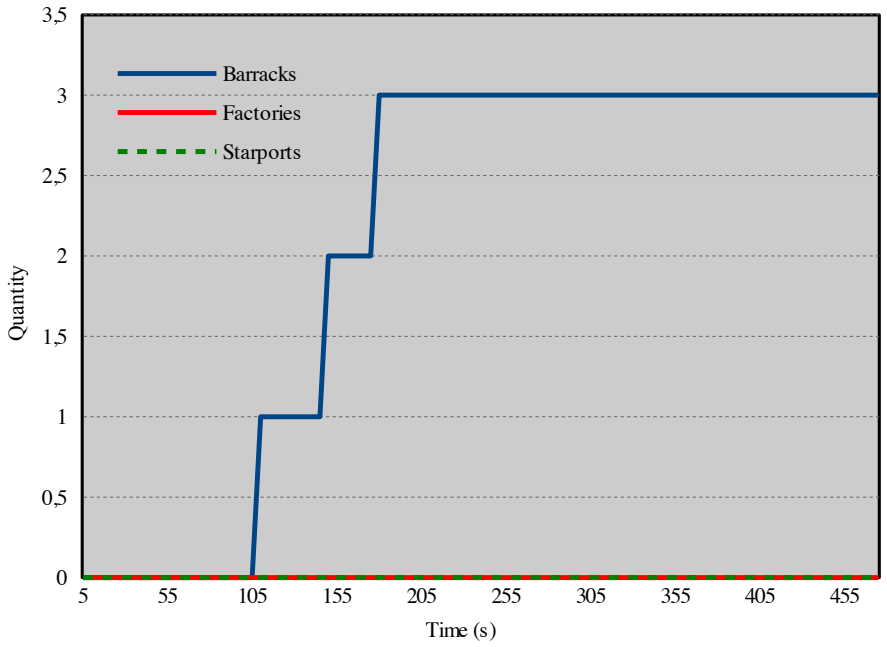
Figure 4.1: Average number of barracks, factories and starports built over time for each strategy.

This demonstrates that, in addition to maintaining a good level of efficiency, the design learned both strategies.

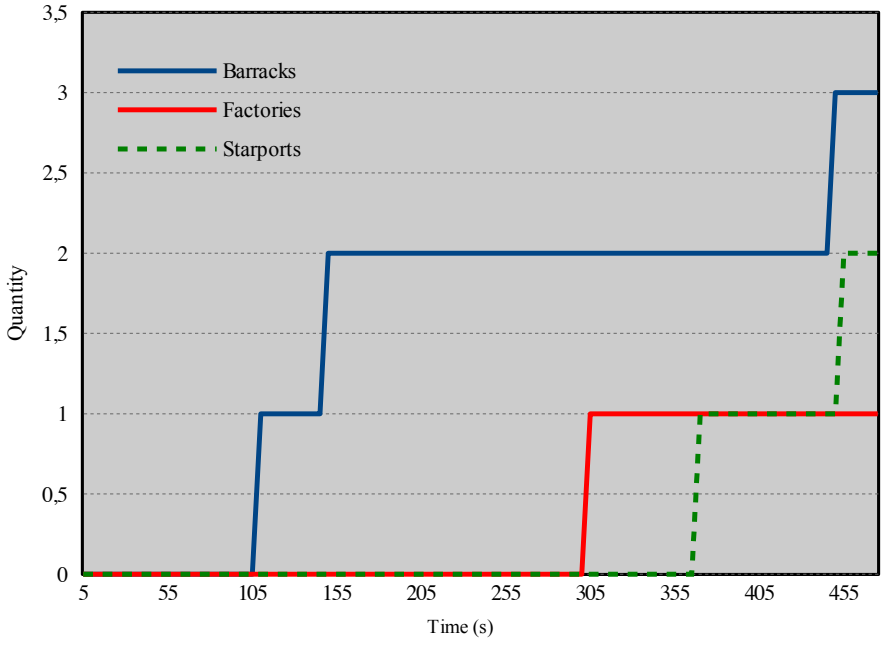
The observed variations between strategies of both agents were expected because of the use of statistical models. However, they are rather limited and we have seen that it does not much affect the performances of the agent. If the design is most likely to be a source of differences between strategies, the simplified combat manager (see Section A.5) might be partially responsible too. Indeed, it can differ by the amount of opponent information gathered and by the issues of battles which are both used to predict production orders.

In addition to evaluating the similarity of the strategies, we measured their frequency. In the Figure 4.3, we can see that the distribution of strategies is more equitable in the set of games played by the "very hard" computer player (training set) than in the one played by the IML agent (test set).

Here again, the combat manager can be a reason of this difference but the learned models have their part of responsibility too. Indeed, neural networks are limited to 15 neurons in their hidden layer for computational considerations and to expect a decent generalization performance when predicting. It means that the complexity of the models might not be sufficient to distinguish different production strategies from quite similar states. Moreover, equal states might be followed by distinct production orders in the dataset because the expert does not always go for the same strategy. When such a situation occurs, the models always favor the most frequent choice because the learning process minimizes the mean square error which is a maximum likelihood estimator [14].



Strategy (A)



Strategy (B)

Figure 4.2: Average number of barracks, factories and starports built over time for the two sets of games resulting of the classification.

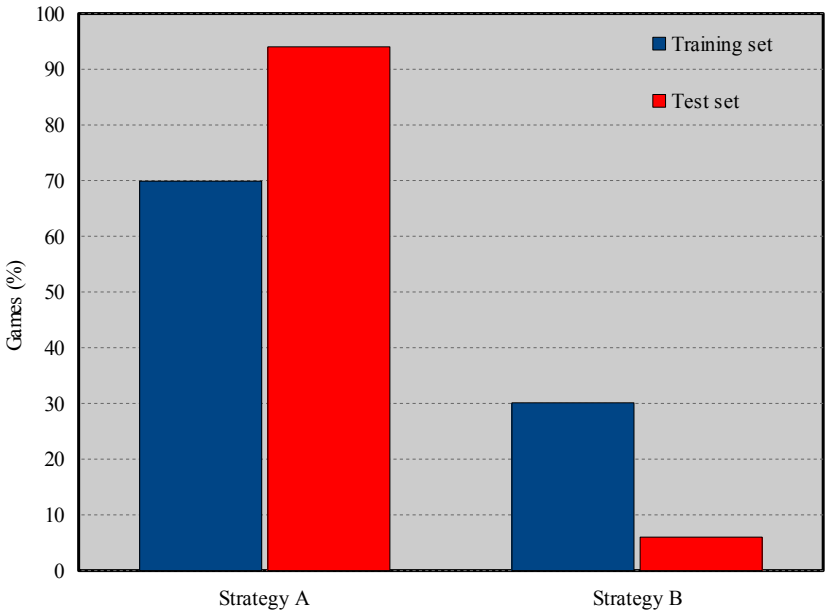


Figure 4.3: Frequency of the two strategies for both agents.

Chapter 5

Conclusion and Future Work

In this work, we proposed a generic design that uses imitative learning to handle the production orders of an autonomous agent for an RTS game. The design has been successfully applied to StarCraft II, one of the most popular and recent RTS game. Indeed, we showed that the agent resulting of the learning process has almost fully inherited the good performance of the expert and that the strategies observed in the dataset were those replicated by the new agent.

Therefore, splitting the complexity of the production manager to train statistical models that each focus on a specific target seems to be a decent solution to learn production strategies in an RTS game. If defining a relevant state for a given game is of course game-dependant, the approach consists mainly in selecting the production variables and defining the opponent's technological tree. Both of these tasks are quite straightforward and the only work that is really specific to the game is to determine the few remaining variables that are important for the production manager.

Once a state vector has been defined, the design provides to game developers the capacity to continuously change the production strategies performed by the computer agents by learning on more recent datasets. They can therefore keep them up to date despite the constant evolution of the strategies in recent RTS titles. By avoiding the necessity to script all the production sequences of the agents, a lot of time can be spared in favor of other managers such as those responsible for scouting and battles.

However, the diversity of the dataset that we used was rather limited because of the weaknesses of the expert. Moreover, even if the agent learns the production strategies of the expert, the most common one in the dataset was over represented in the test set at the expense of the less frequent production strategy. In order to

efficiently learn from richer sets collected from various sources, we suspect clustering would be required to organize records and maintain manageable datasets. Indeed, the dataset could be partitioned into several subsets corresponding to the different strategies in order to train statistical models individually for each cluster. By choosing randomly the models to load with respect to their frequency of appearance in the dataset, less frequent strategies should be better represented in the test set. Furthermore, the production strategies should be learned more precisely as the variance of the different production variables would be reduced.

Appendix A

Implementation Details

This appendix is about the implementation of the agent designed for StarCraft II thanks to the StarCraft II Editor, called Galaxy Editor™. This tool is provided to allow map creation and edition by third-party developers.

A map defines not only the visual battlefield on which players fight but also any specific mechanisms for the players to interact with the game. Map files allow third-party developers to integrate their scripts in order to take control over most of the game aspects.

The scripting language - called Galaxy - is syntactically based on C but lacks numerous features.

A.1 Automatization

The StarCraft II Editor is used to set up the games of the dataset but it is not conceivable to start manually every game. An automatic process has to be implemented to start a new game as soon as the previous one is over. As no script can be executed once a game is over, we define a criterion to determine when the game state guarantees that a player is too far behind and to state that its opponent has won the game. Thanks to a careful observation of several games, it was quite straightforward to see - in our game settings - that a player always wins once he has more than four times the population of his opponent.

To speed up as much as possible the generation of the dataset, two StarCraft II instances were launched at the same time. Using sandboxes was required to enable several instances to run on the same computer. In addition, the speed of the

game was artificially increased by forcing the value of some variables directly in the memory allocated by the game. All these mechanisms allowed us to generate 372 games in about twelve hours even if it actually represents more than 90 hours of game time.

A.2 Galaxy Editor

The main weakness of this language is probably the lack of dynamic allocation. In addition, there are no pointers or any similar concept and only the primitive types are allowed as argument to functions or return value. These limitations are obviously quite restrictive. One of the consequences is that user-defined variable types (restrained to C-like structures) and arrays can not be conveniently exchanged between two functions or code sections. The only way to share data between functions is therefore to use global variables.

Furthermore, there is no way to easily import data into the map file. Several mechanisms are provided to store data but they are restricted to StarCraft-II-related content such as unit or building properties. Moreover, all script files together can not exceed a size of about 2 MB within a map. This bound has to be compared to the amount of variables required for the agent. There are 59 neural networks and they thus require

$$59 \times (15 \times 108 + 16) = 96\ 524$$

weight and bias values. This means that it is hard to generate enough code to initialize all the variables, if only for the neural network weights. Indeed, considering that a character in a script file is encoded over one byte, this limit does not allow more than about 20 characters by variable initialization. Shortening variable names and rounding values would barely be sufficient given that the whole script has still to be written.

Those are the main technical obstacles to develop an agent in StarCraft II but the following sections show how to circumvent them. On a less technical aspect, some difficulties have been encountered concerning API documentation. Indeed, a lot of functions are available to manage the game and to gather information about it but the documentation is often missing or incomplete. Some analysis of native script files has even been required to determine which functions were required for some specific tasks.

A.3 Statistical Models

A.3.1 Importation

The importation of parameters characterizing learned models has been realized by abusing a StarCraft-2-related file format. This file format is actually defined to store what is called *conversion states*. A conversation state defines an interactive dialogue concerning an in-game character and allows a player to answer from a list of predefined replies.

Fortunately, this file format relies on XML and it is possible to store 2-dimension arrays by encapsulating tags appropriately. The Listing A.1 presents an example of such a file. Moreover, the only restriction with predefined data files is not to exceed a size of 10 MB for the whole map file once it has been compressed by the Galaxy Editor thanks to a Lempel-Ziv-Markov chain algorithm.

```
<CConversationState id="Array1">
        <Indices>
            <Id value="Index1"/>
            <InfoValue Id="Value1" Value="15.1"/>
            <InfoValue Id="Value2" Value="10.4"/>
                        <!-- {...} -->
            <InfoValue Id="ValueN" Value="35.0"/>
        </Indices>
        <Indices>
            <Id value="Index2"/>
            <InfoValue Id="Value1" Value="20.5"/>
            <InfoValue Id="Value2" Value="8.7"/>
                        <!-- {...} -->
            <InfoValue Id="ValueN" Value="3.82"/>
        </Indices>
                        <!-- {...} -->
        <Indices>
            <Id value="IndexN"/>
            <InfoValue Id="Value1" Value="52.5"/>
            <InfoValue Id="Value2" Value="75.6"/>
            <InfoValue Id="Value3" Value="1.9"/>
                        <!-- {...} -->
            <InfoValue Id="ValueN" Value="7.98"/>
        </Indices>
</CConversationState>
```

Listing A.1: Conversation-states file for StarCraft 2.

A simple script command is required to access those values from the StarCraft

II Editor. For example, the following script line loads the J^{th} value of the K^{th} index from the array number L :

```
fixed value = ConversationDataStateFixedValue("L|K", "J");
```

An array is therefore stored for every statistical model. The initial range in the dataset of every input variable and the output must be provided to be able to scale components of the state vector during a game with the same parameters as in the training process. Of course, weights and bias for each neuron of the neural network are required in order to compute predictions. The structure of the resulting array is presented in Listing A.2.

```
<CConversationState id="modelID">
        <Indices>
                <!-- Lower bounds in dataset -->
                <Id value="xMin"/>
                <InfoValue Id="0" Value="..."/>
                <InfoValue Id="1" Value="..."/>
                        <!-- {...} -->
                <InfoValue Id="106" Value="..."/>
            <InfoValue Id="out" Value="..."/>
        </Indices>
        <Indices>
                <!-- Upper bounds in dataset -->
                <Id value="xMax"/>
                        <!-- {...} -->
        </Indices>
        <Indices>
                <!-- Output neuron -->
                <Id value="output"/>
                <InfoValue Id="0" Value="-..."/>
                <InfoValue Id="1" Value="..."/>
                <!-- {...} -->
                <InfoValue Id="14" Value="-..."/>
                <InfoValue Id="bias" Value="-..."/>
        </Indices>
                <!-- {...} -->
        <!-- Hidden neurons -->
        <Indices>
                <Id value="0"/>
                <InfoValue Id="0" Value="..."/>
                <InfoValue Id="1" Value="..."/>
                        <!-- {...} -->
                <InfoValue Id="106" Value="..."/>
                <InfoValue Id="bias" Value="..."/>
        </Indices>
        <Indices>
```

```
                <Id value="1"/>
                        <!-- {...} -->
        </Indices>
                        <!-- {...} -->
        <Indices>
                <Id value="14"/>
                        <!-- {...} -->
        </Indices>
</CConversationState>
```

Listing A.2: Structure of the arrays used to store learned parameters.

In addition to storing these parameters, another array is used to specify the identifier of the model (*modelID*) within the XML file of each neural network and the index of the associated production variable within the state vector. A flag is used too to know if the production variable refers to a research or not. Indeed, technologies sometimes require different Galaxy functions.

The resulting XML file has a raw size of 2.29 MB but this size is reduced to 369 KB when compressed. The limit of 10 MB is thus not a problem.

A.3.2 Prediction

In order to compute the predictions of the statistical models, a sequence of simple steps is followed. First of all, components of the state vector have to be scaled and then the dot products between the scaled input vector and each neuron weight vector is computed. Then, the addition of the bias values of every neuron and the computation of the activation function follow. This sequence is repeated for the neuron of the output layer using outputs of previous neurons as inputs and the resulting value is unscaled. The implementation in Galaxy code of the function dedicated to this task is shown in Listing A.3.

```
int predictAmount(int nn, int toPredictIndx)
{
    fixed[HIDDENS] features; // Outputs of the hidden layer.
    fixed output = 0.0;      // Output of the output layer.
    int feat = 0;  // Neuron being processed
    int in = 0;    // Input being processed
    int delta = 0; // Used to shift input index when
                   // the targeted variable has to be skipped

    // Compute the output for the neurons in the hidden layer.
    while (feat < HIDDENS)
    {
```

```
        in = 0;
        delta = 0;
        // Compute the dot product between the state vector
        // and the weight vector.
        while (in < INPUTS)
        {
            if (in == toPredictIndx)
            {
                delta = 1;
            }
            features[feat]
                   += getScaledInput(nn, in, state[in+delta])
                   * getHiddenWeight(nn, feat, in);
            in += 1;
        }
        // Add the bias to the dot product
        features[feat] += getHiddenBias(nn, feat);
        // Compute the activation function's output
        // (= tansig(features[feat]))
        features[feat] = (2/(1+Pow(E,-2*features[feat])))-1;
        // Compute incrementally the output
        // of the whole network.
        output += features[feat] * getOutputWeight(nn, feat);
        feat += 1;
    }
    // Add the bias to the dot product of the output neuron
    output += getOutputBias(nn);
    // Predicted value is rounded because production
    //variables are natural numbers.
    return RoundI(getScaledOutput(nn, output));
}
```

Listing A.3: Function that computes the prediction of the nn^{th} neural network.

A.4 Production Management

The goal of the production manager is to translate predictions from the statistical models to production orders in the game. However, we do not want to have to script a whole new agent. The objective is therefore to benefit from the computer agent of StarCraft II and to take control over production only. For this purpose, an existing function provided by the Galaxy Editor is particularly useful :

```
void AIMakeAlways(int player, int priority,
                  int town, string objectType, int count);
```

Indeed, this function handles all the operational needs in order to start the construction of a building or the production of a unit or research for a given player. The string *objectType* identifies the building, unit or technology (object) to produce or to research while the integer *player* specifies the targeted computer player. The *count* and *town* parameters indicate respectively the number of times that the object has to be produced and the town that has to be responsible of the production. We will always use the value -1 for *town* which means that it does not matter. In order to have an agent that is robust against punctual errors of the statistical models, the production manager asks - for each object type - one production at most by prediction and the value of *count* is therefore fixed to 1. The effect of the *priority* parameter is not obvious and its usage is not documented, we will thus not make use of it and we fix its value to 0.

Some conditions have to be respected for this function to have an effect. Indeed, the request will be accepted only if the agent has enough resources and if its current technological tree allows the production of the required object. When these conditions are satisfied, the request goes into a waiting queue dedicated to the production orders of the agent. The effective production starts as soon as there is an available worker (SCV) or an available production facility suited for the request.

As stated in Section 2.3, the randomization of the order in which production variables are processed prevents the emergence of predictive behaviors. However, some mechanisms of the game engine of StarCraft II seem to induce an unequal priority to different object types. For example, a production order asking for the construction of a *Tech Lab* (building add-on) for a barracks will always be neglected in favor of a request to produce a *Marine* (infantry unit) if this latter is made within about the same second and that there is only one barracks available.

In order to avoid this favoritism, the production manager waits for two successive predictions to be greater than a current production variable before effectively asking the agent to produce the concerned object. By doing so, the requests for the different object types are sometimes alternated and the predictive behavior induced by the game engine is decreased.

While taking this precaution, the production manager iterates on the production variables and checks if the technological tree allows the production of a new object. If so, the associated neural network is used to predict the number of objects of this type that should be currently in the game. If the prediction is greater than the actual number and if the agent has enough resources, the production manager requests a new production for this object.

Sometimes, there may not be enough workers or suited production facilities to process all the requests. When such a situation occurs, the remaining requests stay in the processing queue of the agent and wait to be produced. If these requests were still pending before the next iteration of this procedure, the production manager would delete them from the queue. Indeed, time has elapsed since the last iteration and the choice that has been made might no longer be relevant given the current situation. Therefore, instead of considering pending requests as mandatory, the predictions are computed again for the new state and new production orders are given if the state is judged inconsistent.

Finally, the whole procedure is summarized in Listing A.4. During a game, this procedure is always restarted 1 second after its previous execution. This small break allows the requests to be processed by the game engine and releases enough computational resources for the game to run smoothly. Another small break is also required (about 50 ms) after the processing of each production variable to avoid short game freezes. The whole prediction procedure lasts therefore at most $1 + 59 * 0.05 +$ computations ≈ 4 seconds. However, all the production variables have not been used by the expert and some of them do not need to be predicted. Practically, this procedure lasts about 3 seconds in our learning configuration. As a threshold of 2 successive greater predictions is set, a request is therefore sent to the agent 6 seconds at worst after it should have been started with respect to the corresponding statistical model, to the amount of resources available and to the technological tree.

```
clearPendingRequests(bot)
minerals ← getMinerals(bot)
gas ← getGas(bot)
state = getGameState()

for var in productionVariables
   if allowed(var, techTree(bot))
      prediction ← predict(var.model, state);
      if prediction > var.value
         var.consec  ← var.consec + 1
         if var.consec ≥ 2
            var.consec ← 0
            if var.mineralCost ≤ minerals and var.gasCost ≤ gas
               minerals ← minerals − var.mineralCost
               gas ← minerals − var.gasCost
               AIMakeAlways(bot, 0, −1, var, 1)
               state[var] ← state[var] + 1
            end
         end
      end
```

```
    end
end
```

Listing A.4: Production procedure

A.5 Combat Management

Developing a combat manager is not the objective of this work. Therefore, our agent inherits most of its combat mechanisms from the computer agent of StarCraft II. However, as we want to take care of the production orders, we have to disable the native production manager and the one that is responsible for when and where to attack is necessarily disabled too.

To handle high level decisions responsible for launching attack waves, a simple trigger-based approach is used. The agent observes continuously the time elapsed since the start of the game, the population value of its army and the number of opposing units it has killed. If these figures suggest that the agent could have a substantial military advantage, an attack wave is launched against the opponent. The checks performed by this combat manager have the structure showed in Listing A.5, each check handling a different time window.

```
if t_i ≤ gameTime() < t_{i+1}
    if (armyPopulation() ≥ thres_a^i and nbUnitsKilled() ≥ thres_b^i)
            or armyPopulation() ≥ thres_c^i
        attack()
    end
end
```

Listing A.5: Combat manager.

Appendix B

Screenshots

This appendix contains some screenshots of the learned agent during a game.

Figure B.1: The agent building barracks in the beginning of a game.

Figure B.2: Extractors have been built to harvest vespene gas, more buildings are in construction and a substantial army has already been produced.

Figure B.3: The agent takes an expansion that will boost its resource harvesting rate once finished.

Figure B.4: The opponent retreating against the stronger army of the agent after suffering heavy losses.

Figure B.5: The agent taking advantage of its lead to push forward the opponent's base.

Figure B.6: The agent ending the game by destroying its opponent's base.

Appendix C

Scientific Paper

The following paper is based upon the material published in this thesis. It has been submitted to CIG 2012, a IEEE conference in Computational Intelligence in the video games environment (`http://www.ieee-cig.org/`).

Imitative Learning for Real-Time Strategy Games

Quentin Gemine, Firas Safadi, Raphaël Fonteneau and Damien Ernst

***Abstract*—Over the past decades, video games have become increasingly popular and complex. Virtual worlds have gone a long way since the first arcades and so have the artificial intelligence (AI) techniques used to control agents in these growing environments. Tasks such as world exploration, constrained pathfinding or team tactics and coordination just to name a few are now default requirements for contemporary video games. However, despite its recent advances, video game AI still lacks the ability to learn. In this paper, we attempt to break the barrier between video game AI and machine learning and propose a generic method allowing real-time strategy (RTS) agents to learn production strategies from a set of recorded games using supervised learning. We test this imitative learning approach on the popular RTS title StarCraft II® and successfully teach a Terran agent facing a Protoss opponent new production strategies.**

I. Introduction

Video games started emerging roughly 40 years ago. Their purpose is to bring entertainment to the people by immersing them in virtual worlds. The rules governing a virtual world and dictating how players can interact with objects or with one another are referred to as game mechanics. The first video games were very simple: small 2-dimensional discrete space, less than a dozen mechanics and one or two players at most. Coding agents capable of autonomously playing these games required no longer than a few lines. Today, video games feature large 3-dimensional spaces, hundreds of mechanics and allow numerous players and agents to play together. Among the wide variety of genres, real-time strategy (RTS) provides one of the most complex environments overall. The multitude of tasks and objects involved as well as the highly dynamic environment result in extremely large and diverging state and action spaces. This renders the design of autonomous agents difficult. Currently, most approaches largely rely on generic triggers. Generic triggers aim at catching general situations such as being under attack with no consideration to the details of the attack (i.e., location, number of enemies, ...). These methods are easy to implement and allow agents to adopt a robust albeit non-optimal behavior in the sense that agents will not fall into a state for which no trigger is activated, or in other words a state where no action is taken. Unfortunately, this type of agent will often discard crucial context elements and fail to display the natural and intuitive behavior we may expect. Additionally, while players grow more familiar with the game mechanics and improve their skills and devise new strategies, agents do not change and eventually become obsolete. This evolutionary requirement is critical for performance in RTS games where the pool of possible strategies is so large that it is impossible to estimate optimal behavior at the time of development. Although it is common to increase difficulty by granting agents an unfair advantage, this approach seldom results in entertainment and either fails to deliver the sought-after challenge or ultimately leads to player frustration.

Because the various facets of the RTS genre constitute very distinct problems, several learning technologies would be required to grant agents the ability to learn on all aspects of the game. In this work, we focus on the production problem and propose a generic method to teach an agent production strategies from a set of recorded games using supervised learning. We chose StarCraft II® as our testing environment. Today, StarCraft II, Blizzard Entertainment's successor to genre patriarch StarCraft®, is one of the top selling RTS games. Featuring a full-fledged game editor, it is the ideal platform to assess this new breed of learning agents. Our approach is validated on the particular scenario of a one-on-one, Terran versus Protoss matchup type. The created agent architecture comprises both a dynamically learned production model based on multiple neural networks as well as a simple scripted combat handler.

The paper is structured as follows. Section 2 briefly covers some related work. Section 3 details the core mechanics characterizing the RTS genre. Section 4 and 5 present the learning problem and the proposed solution, respectively. Section 6 discusses experimental results and, finally, Section 7 concludes.

II. Related Work

Lately, the video game industry has attracted substantial research work, be it for the purpose of developing new technologies to boost entertainment and replay value or simply because modern video games have become an alternate, low-cost yet rich environment for assessing machine learning algorithms.

Roughly, we could distinguish 2 goals in video game AI research. Some work aims at creating agents with properties that make them more fun to play with such as human-like behavior. This is usually attempted on games for which agents capable of challenging skilled human players already exist. This is necessary because, often in this case, agents manage to rival human players due to unfair advantages: instant reaction time, perfect aim, etc. These features increase performance at the cost of frustrating human opponents. For more complicated games, agents stand no chance against skilled human players and improving their performance takes priority. Hence, performance similar to what humans can achieve can be seen as a prerequisite to entertainment. Indeed, facing a too weak or too strong opponent is not usually diverting. This concept is illustrated in Figure 1. In either case, video game AI research advances towards the

ultimate goal of mimicking human intelligence. It was in fact suggested that human-level AI can be pursued directly in these new virtual environments [1].

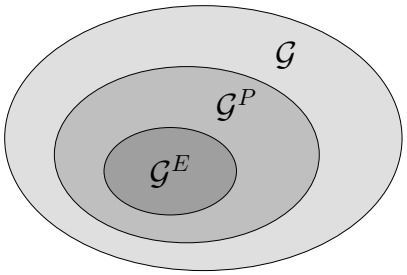


Fig. 1. Agent set structure for a video game; $\mathcal{G}^E \subset \mathcal{G}^P \subset \mathcal{G}$ where $\mathcal{G}^E$ is the set of agents the player finds entertaining, $\mathcal{G}^P$ is the set of agents that can rival the player's performance and $\mathcal{G}$ is the set of all agents.

The problem of human-like agent behavior has been tackled in first-person shooter (FPS) games, most notably the popular and now open-source game Quake II®, using imitative learning. Using clustering by vector quantization to organize recorded game data and several neural networks, more natural movement behavior as well as switching between movement and aim was achieved in Quake II [2]. Human-like behavior was also approached using dedicated neural networks for handling weapon switching, aiming and firing [3]. Further work discussed the possibility of learning from humans at all levels of the game, including strategy, tactics and reactions [4].

While human-like agent behavior was being pursued, others were more concerned with performance issues in genres like real-time strategy (RTS) where the action space is too large to be thoroughly exploited by generic triggers. Classifiers based on neural networks, Bayesian networks and action trees assisted by quality threshold clustering were successfully used to predict enemy strategies in StarCraft [5]. Case-based reasoning has also been employed to identify strategic situations in Wargus, an open-source Warcraft II® clone [6, 7, 8]. Other works resorted to data mining and evolutionary methods for strategy planning and generation [9, 10]. Non-learning agents were also proposed [11]. By clearly identifying and organizing tasks, architectures allowing incremental learning integration at different levels were developed [12].

Although several different learning algorithms were applied in RTS environments, none were actually used to dictate agent behavior directly. In this paper, we use imitative learning to teach a StarCraft II agent to autonomously pass production orders. The created agent building, unit and technology production is entirely governed by the learning algorithm and does not involve any scripting.

III. Real-Time Strategy

In a typical RTS game, players confront each other on a specific map. The map is essentially defined by a combination of terrain configuration and resource fields. Once the game starts, players must simultaneously and continuously acquire resources and build units in order to destroy their opponents. Depending on the technologies they choose to develop, players gain access to different unit types each with specific attributes and abilities. Because units can be very effective against others based on their type, players have to constantly monitor their opponents and determine the combination of units which can best counter the enemy's composition. This reconnaissance task is referred to as scouting and is necessary because of the "fog of war", which denies visibility to players over areas where they have no units deployed.

Often, several races are available for the players to choose from. Each race possesses its own units and technologies and is characterized by a unique play style. This further adds to the richness of the environment and multiplies mechanics. For example, in StarCraft II players can choose between the Terrans, masters of survivability, the Zerg, an alien race with massive swarms, or the Protoss, a psychically advanced humanoid species.

Clearly, players are constantly faced with a multitude of decisions to make. They must manage economy, production, reconnaissance and combat all at the same time. They must decide whether the current income is sufficient or new resource fields should be claimed, they must continuously gather information on the enemy and produce units and develop technologies that best match their strategies. Additionally, they must swiftly and efficiently handle units in combat.

When more than two players are involved, new diplomacy mechanics are introduced. Players may form and break alliances as they see fit. Allies have the ability to share resources and even control over units, bringing additional management elements to the game.

Finally, modern RTS games take the complexity a step further by mixing in role-playing game (RPG) mechanics. Warcraft III®, an RTS title also developed by Blizzard Entertainment, implements this concept. Besides regular unit types, heroes can be produced. Heroes are similar to RPG characters in that they can gain experience points by killing critters or enemy units to level up. Leveling up improves their base attributes and grants them skill points which can be used to upgrade their special abilities.

With hundreds of units to control and dozens of different unit types and special abilities, it becomes obvious why the RTS genre stands on top in terms of overall complexity.

IV. Problem Statement

The problem of learning production strategies in an RTS game can be formalized as follows.

Consider a fixed player u. A world vector $\boldsymbol{w} \in \mathcal{W}$ is a vector describing the entire world at a particular time in the game. An observation vector $\boldsymbol{o} \in \mathcal{O}$ is the projection of $\boldsymbol{w}$ over an observation space $\mathcal{O}$ describing the part of the world perceivable by player u. We define a state vector $\boldsymbol{s} \in \mathcal{S}$ as the projection of $\boldsymbol{o}$ over a space $\mathcal{S}$ by selecting variables deemed relevant to the task of learning production strategies.

Let $n \in \mathbb{N}$ be the number of variables chosen to describe the state. We have:

$$\boldsymbol{s} = (s_1, s_2, ..., s_n), \forall i \in \{1, ..., n\} : s_i \in \mathbb{R}$$

Several components of $\boldsymbol{s}$ are variables that can be directly influenced by production orders. Those are the variables that describe the number of buildings of each type available or planned, the cumulative number of units of each type produced or planned and whether each technology is researched or planned. If a technology is researched or planned, the corresponding variable is equal to 1, otherwise, it is equal to 0. Let m be the number of these variables and let s_{p_1}, s_{p_2}, ..., s_{p_m} be the components of $\boldsymbol{s}$ that correspond to these variables.

When in state $\boldsymbol{s}$, a player u can select an action vector $\boldsymbol{a} \in \mathcal{A}$ of size m that gathers the "production orders". The j^{th} component of this vector corresponds to the production variable s_{p_j}. When an action $\boldsymbol{a}$ is taken, the production variables of $\boldsymbol{s}$ are immediately modified according to:

$$\forall j \in \{1, ..., m\} : s_{p_j} \leftarrow s_{p_j} + a_j$$

We define a production strategy for player u as a mapping $P : \mathcal{S} \rightarrow \mathcal{A}$ which selects an action vector $\boldsymbol{a}$ for any given state vector $\boldsymbol{s}$:

$$\boldsymbol{a} = P(\boldsymbol{s})$$

V. Learning Architecture

We assume that a set of recorded games is provided. Each recorded game consists of a set of state vectors of player u. Our objective is to learn a production strategy P as close as possible to the production strategy used by player u. To achieve this, we use supervised learning to learn to predict each production variable s_{p_j} based on the remaining state $\boldsymbol{s}_{-p_j}$ defined below. We then use the predicted s_{p_j} values to deduce a production order $\boldsymbol{a}$. Since there are m production variables, we solve m supervised learning problems. Formally, our approach works as follows.

For each production variable s_{p_j}, we define the remaining state as $\boldsymbol{s}_{-p_j}$:

$$\forall j \in \{1, ..., m\} : \boldsymbol{s}_{-p_j} = (s_1, s_2, ..., s_{p_j - 1}, s_{p_j + 1}, ..., s_n)$$

For each production variable s_{p_j}, we also define a function P_j which maps each remaining state to s_{p_j}:

$$\forall j \in \{1, ..., m\} : P_j(\boldsymbol{s}_{-p_j}) = s_{p_j}$$

Knowing each P_j, we can deduce the mapping P and estimate a production order $\boldsymbol{a}$ for any given state vector $\boldsymbol{s}$:

$$\begin{aligned}\boldsymbol{a} = P(\boldsymbol{s}) = (&P_1(\boldsymbol{s}_{-p_1}) - s_{p_1}, \\ &P_2(\boldsymbol{s}_{-p_2}) - s_{p_2}, ..., P_m(\boldsymbol{s}_{-p_m}) - s_{p_m})\end{aligned}$$

Using this approach, we learn the production strategy used by player u by learning m P_j functions to estimate production variables given the remaining state variables. Each P_j is learned separately using supervised learning. In other words, we learn m models. For each model, the input for the learning algorithm is the state vector $\boldsymbol{s}$ stripped from the component the model must predict, which becomes the output. This process is illustrated in Figure 2.

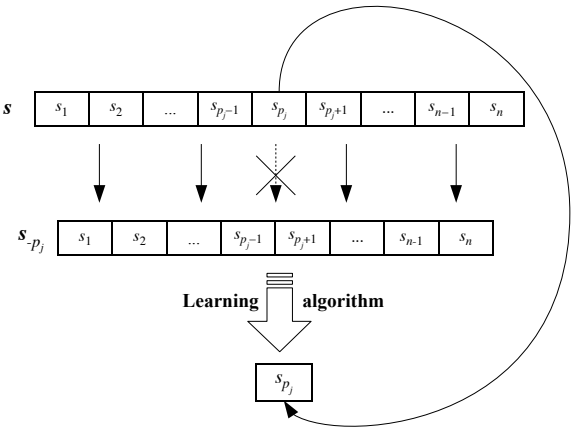


Fig. 2. Learning the kth model

It is worth stressing that the action vector $\boldsymbol{a}$ computed by the mapping P learned may not correspond to, due to the constraints imposed by the game, an action that can be taken. For example, $\boldsymbol{a}$ may send among others an order for a new type of unit while the technology it requires is not yet available. In our implementation, every component of $\boldsymbol{a}$ which is inconsistent with the state of the game is simply set to zero before the action vector is applied.

VI. Experimental Results

The proposed method was tested in StarCraft II by teaching a Terran agent facing a Protoss opponent production strategies.

A total of $n = 108$ variables were selected to describe a state vector. These state variables are:

- $s_1 \in \mathbb{N}$ is the time elapsed since the beginning of the game in seconds
- $s_2 \in \mathbb{N}$ is the total number of units owned by the agent
- $s_3 \in \mathbb{N}$ is the number of SCVs (Space Construction Vehicles)
- $s_4 \in \mathbb{N}$ is the average mineral harvest rate in minerals per minute
- $s_5 \in \mathbb{N}$ is the average gas harvest rate in gas per minute
- $s_u \in \mathbb{N}, u \in \{6, ..., 17\}$ is the cumulative number of units produced of each type
- $s_b \in \mathbb{N}, b \in \{18, ..., 36\}$ is the number of buildings of each type
- $s_t \in \{0, 1\}, t \in \{37, ..., 63\}$ indicates whether each technology has been researched
- $s_e \in \{0, 1\}, e \in \{64, ..., 108\}$ indicates whether an enemy unit type, building type or technology has been encountered

Among these, there are $m = 58$ variables which correspond to direct production orders: 12 s_u unit variables, 19 s_b building variables and 27 s_t technology variables. Therefore, an action vector is composed of 58 variables. These action variables are:

- $a_u \in \mathbb{N}, u \in \{1, ..., 12\}$ corresponds to the number of additional units of each type the agent should produce
- $a_b \in \mathbb{N}, b \in \{13, ..., 31\}$ corresponds to the number of additional buildings of each type the agent should build
- $a_t \in \{0, 1\}, t \in \{32, ..., 58\}$ corresponds to the technologies the agent should research

The Terran agent learned production strategies from a set of 372 game logs generated by letting a Very Hard Terran computer player play against a Hard Protoss computer player on the Metalopolis map. State vectors were dumped every 5 seconds in game time. Each P_j was learned using a feedforward neural network with a 15-neuron hidden layer and the Levenberg-Marquardt backpropagation algorithm to update weights. Inputs and outputs were mapped to the $[-1, 1]$ range. A tan-sigmoid activation function was used for hidden layers.

These 58 neural networks were combined with a simple scripted combat manager. During a game, the agent periodically predicts production orders. For any given building type, unit type or technology, if the predicted target value $P_j(\boldsymbol{s}_{-p_j})$ is greater than the current number s_{p_j}, a production order a_j is passed to reach the target value. This behavior is illustrated in Figure 3.

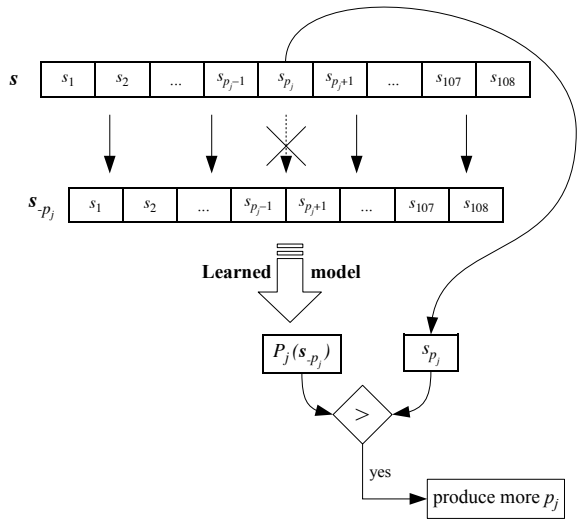


Fig. 3. Agent production behavior

The final agent was tested in a total of 50 games using the same settings used to generate the training set. The results are summarized in Table 1. With a less sophisticated combat handler, the imitative learning trained agent (IML agent) managed to beat the Hard Protoss computer player 9 times out of 10 on average while the Hard Terran computer player lost every game. This performance is not far below that of the Very Hard Terran computer player the agent learned from, which achieved an average win rate of 96.5%. In addition to counting victories, we have attempted to verify that the agent indeed replicates to some extent the same production strategies as those from the training set. Roughly, two different strategies were used by the Very Hard Terran computer player. The first one (A) primarily focuses on infantry while the second one (B) aims at faster technological development. Formally, a game is given the label Strategy A if no factories or starports are built during the first 5 minutes of the game. Otherwise it is labeled Strategy B. Figure 4 shows, for the training set, the average number of barracks, factories and starports built over time for each strategy. Two corresponding strategies were also observed for the learning agent over the 50 test games, as shown in Figure 5. For each strategy, the frequency of appearance is shown in Figure 6.

TABLE I
TERRAN PERFORMANCE AGAINST HARD PROTOSS

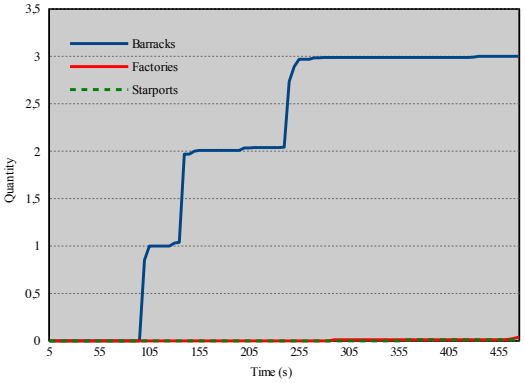
	Terran win rate	**Total games**
Very Hard Terran	96.5%	372
Hard Terran	0%	50
Learning agent	**90%**	**50**

The frequency at which each strategy is used was not faithfully reproduced on the test set. This can be partly explained by the more limited combat handler, which may fail to acquire the same information on the enemy than was available in the training set. Moreover, Strategy B seems to be less accurately replicated than Strategy A. This may be caused by the lower frequency of appearance in the training set. Nevertheless, the results obtained indicate that the agent learned both production strategies from the Very Hard Terran computer player. Subsequently, we may rightly attribute the agent's high performance to the fact that it managed to imitate the efficient production strategies used by the Very Hard Terran computer player.

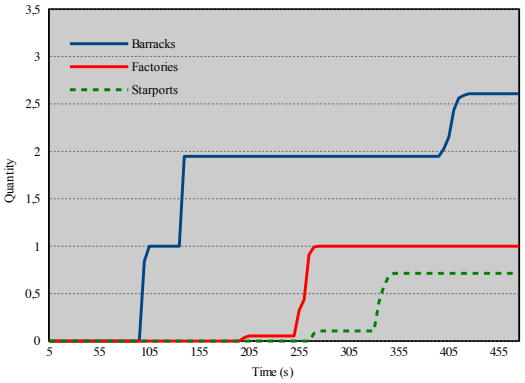
VII. CONCLUSION AND FUTURE WORK

In this paper, we have presented a method for integrating imitative learning in real-time strategy agents. The proposed solution allowed the creation of an agent for StarCraft II capable of learning production strategies from recorded game data and applying them in full one-on-one games.

Although we managed to create a learning agent for StarCraft II, the training set was relatively small and the data diversity rather limited. In order to efficiently learn from richer sets collected from various sources, we suspect clustering will be required to organize records and maintain manageable data sets. Furthermore, the manually generated training data only contained desirable production strategies. When training data is automatically collected from various sources, selection techniques will be required to filter out undesirable production strategies. Besides production-related improvements, there are other areas worth investing in to

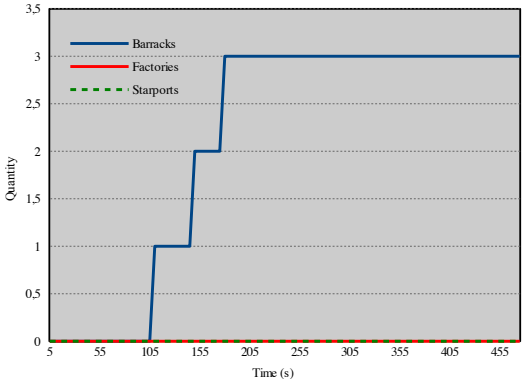


(a) Strategy A

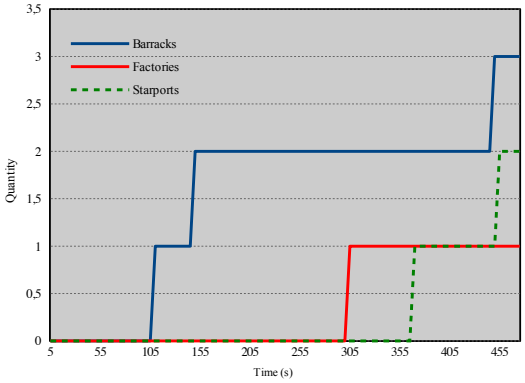


(b) Strategy B

Fig. 4. Training set strategies



(a) Strategy A



(b) Strategy B

Fig. 5. Test set strategies

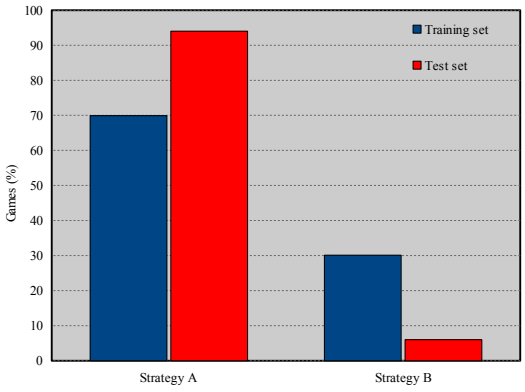


Fig. 6. Strategy frequencies

increase agent performance such as information management or combat management. Enhanced information management can allow an agent to better estimate the state of its opponents and for example predict the location of unit groups that could be killed before they can retreat or be joined by backup forces. As for combat management, it may lead to much more efficient unit handling in battle and for example maximize unit life spans.

ACKNOWLEDGMENTS

Raphael Fonteneau is a postdoctoral researcher of the FRS-FNRS from which he acknowledges the financial support. This paper presents research results of the European Network of Excellence PASCAL2 and the Belgian Network DYSCO funded by the Interuniversity Attraction Poles Programme, initiated by the Belgian State, Science Policy Office.

REFERENCES

[1] J. E. Laird and van Michale Lent, "Human-level ai's killer application: Interactive computer games," in *Proceedings of the 17th National Conference on Artificial Intelligence*, 2000.

[2] C. Bauckhage, C. Thurau, and G. Sagerer, "Learning human-like opponent behavior for interactive computer games," *Pattern Recognition*, pp. 148–155, 2003.

[3] B. Gorman and M. Humphrys, "Imitative learning of combat behaviours in first-person computer games," in *Proceedings of the 10th International Conference on Computer Games: AI, Mobile, Educational and Serious*

Games, 2007.

[4] C. Thurau, G. Sagerer, and C. Bauckhage, "Imitation learning at all levels of game ai," in *Proceedings of the International Conference on Computer Games, Artificial Intelligence, Design and Education*, 2004.

[5] F. Frandsen, M. Hansen, H. Sørensen, P. Sørensen, J. G. Nielsen, and J. S. Knudsen, "Predicting player strategies in real time strategy games," Master's thesis, 2010.

[6] S. Ontañón, K. Mishra, N. Sugandh, and A. Ram, "Case-based planning and execution for real-time strategy games," in *Proceedings of the 7th International Conference on Case-Based Reasoning (ICCBR-07)*, 2007, pp. 164–178.

[7] D. W. Aha, M. Molineaux, and M. J. V. Ponsen, "Learning to win: case-based plan selection in a real-time strategy game," in *Proceedings of the 6th International Conference on Case-Based Reasoning (ICCBR-05)*, 2005, pp. 5–20.

[8] B. Weber and M. Mateas, "Case-based reasoning for build order in real-time strategy games," in *Proceedings of the 5th Artificial Intelligence for Interactive Digital Entertainment Conference (AIIDE-09)*, 2009.

[9] B. G. Weber and M. Mateas, "A data mining approach to strategy prediction," in *Proceedings of the 5th IEEE Symposium on Computational Intelligence and Games (CIG-09)*. IEEE Press, 2009, pp. 140–147.

[10] M. Ponsen, H. Muñoz-Avila, P. Spronck, and D. Aha, "Automatically generating game tactics via evolutionary learning," *AI Magazine*, vol. 27, no. 3, pp. 75–84, 2006.

[11] J. McCoy and M. Mateas, "An integrated agent for playing real-time strategy games," in *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI-08)*, 2008, pp. 1313–1318.

[12] F. Safadi, R. Fonteneau, and D. Ernst, "Artificial intelligence design for real-time strategy games," in *Proceedings of the 2nd International Workshop on Decision Making with Multiple Imperfect Decision Makers*, 2011.

[13] D. C. Cheng and R. Thawonmas, "Case-based plan recognition for real-time strategy games," in *Proceedings of the 5th International Conference on Computer Games: Artificial Intelligence, Design and Education (CGAIDE-04)*, 2004, pp. 36–40.

[14] M. Chung, M. Buro, and J. Schaeffer, "Monte-carlo planning in real-time strategy games," in *Proceedings of the 1st IEEE Symposium on Computational Intelligence and Games (CIG-05)*, 2005.

[15] A. Kovarsky and M. Buro, "A first look at build-order optimization in real-time strategy games," in *Proceedings of the 2006 GameOn Conference*, 2006, pp. 18–22.

[16] B. G. Weber and M. Mateas, "Conceptual neighborhoods for retrieval in case-based reasoning," in *Proceedings of the 8th International Conference on Case-Based Reasoning (ICCBR-09)*, 2009, pp. 343–357.

Bibliography

[1] Sandra Upson. How a computer game is reinventing the science of expertise. *Scientific American Blog Network*, 2011. `http://blogs.scientificamerican.com/observations/2011/12/01/how-a-computer-game-is-reinventing-the-science-of-expertise-video/`. 2

[2] John E. Laird and van Michale Lent. Human-level ai's killer application: Interactive computer games. In *Proceedings of the 17th National Conference on Artificial Intelligence*, 2000. 4

[3] Christian Bauckhage, Christian Thurau, and Gerhard Sagerer. Learning human-like opponent behavior for interactive computer games. *Pattern Recognition*, pages 148–155, 2003. 5

[4] Bernard Gorman and Mark Humphrys. Imitative learning of combat behaviours in first-person computer games. In *Proceedings of the 10th International Conference on Computer Games: AI, Mobile, Educational and Serious Games*, 2007. 5

[5] Christian Thurau, Gerhard Sagerer, and Christian Bauckhage. Imitation learning at all levels of game ai. In *Proceedings of the International Conference on Computer Games, Artificial Intelligence, Design and Education*, 2004. 5

[6] Frederik Frandsen, Mikkel Hansen, Henrik Sørensen, Peder Sørensen, Johannes Garm Nielsen, and Jakob Svane Knudsen. Predicting player strategies in real time strategy games. Master's thesis, 2010. 5

[7] Santiago Ontañón, Kinshuk Mishra, Neha Sugandh, and Ashwin Ram. Case-based planning and execution for real-time strategy games. In *Proceedings of the 7th International Conference on Case-Based Reasoning (ICCBR-07)*, pages 164–178, 2007. 5

[8] David W. Aha, Matthew Molineaux, and Marc J. V. Ponsen. Learning to win: case-based plan selection in a real-time strategy game. In *Proceedings of the 6th International Conference on Case-Based Reasoning (ICCBR-05)*, pages 5–20, 2005. 5

[9] Ben Weber and Michael Mateas. Case-based reasoning for build order in real-time strategy games. In *Proceedings of the 5th Artificial Intelligence for Interactive Digital Entertainment Conference (AIIDE-09)*, 2009. 5

[10] Ben G. Weber and Michael Mateas. A data mining approach to strategy prediction. In *Proceedings of the 5th IEEE Symposium on Computational Intelligence and Games (CIG-09)*, pages 140–147. IEEE Press, 2009. 5

[11] Marc Ponsen, Héctor Muñoz-Avila, Pieter Spronck, and David Aha. Automatically generating game tactics via evolutionary learning. *AI Magazine*, 27(3):75–84, 2006. 5

[12] Josh McCoy and Michael Mateas. An integrated agent for playing real-time strategy games. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI-08)*, pages 1313–1318, 2008. 5

[13] Firas Safadi, Raphael Fonteneau, and Damien Ernst. Artificial intelligence design for real-time strategy games. In *Proceedings of the 2nd International Workshop on Decision Making with Multiple Imperfect Decision Makers*, 2011. 5

[14] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001. 26, 35

[15] Warren S. Sarle. Neural network faq, 1997. `ftp://ftp.sas.com/pub/neural/FAQ.html`. 26

[16] K. Madsen, H. B. Nielsen, and O. Tingleff. Methods for non-linear least squares problems (2nd ed.), 2004. 27

[17] Henri Gavin. The levenberg-marquardt method for nonlinear least squares curve-fitting problems. *Environmental Engineering*, pages 1–15, 2011. 27