UMONS
Université de Mons

Faculté
des Sciences

Neural-network-based AIs for grid-based computer games

Master's thesis

Thesis realised by: Anthony Rouneau
Section: Master in Computer Sciences
Academic year: 2016-2017

Under the direction of: Pr. Tom Mens
Department: Software engineering

Contents

Part I
Introduction

This masters thesis was written within the context of the last year of a Master's degree in Computer Sciences at the University of Mons (UMONS). The main goal is to evaluate the use of machine learning and neural network techniques on grid-based games. Indeed, machine learning has been an important domain of research for video games [1], [2], and neural networks have been used a lot lately to outperform the results of classical machine learning techniques in multiple domains [3], [4].

To achieve this goal, we will develop an open-source, extensive and robust game engine designed to easily prototype grid-based games and easily test artificial intelligences (AIs) on the developed games. Indeed, nowadays, most of the free game engines, like LibGDX [5] or GODOT [6], imply to deeply understand their structure and functioning in order to develop a new game using them. Hence, the objective of our engine is to provide a simple and fast way to implement a new game as well as to provide a simple way to use and test any type of AI for the developed games. To prove that our engine can accept diverse types of AIs, we will the learning power of neural networks to develop intelligent and evolutive AIs that play the games (game agents). To do so, we will develop automatic data collection algorithms using our game engine and we will use the generated data to make neural networks models learn how to play each game. Finally, we will create bots that play games using these models.

1 Goals

There are two main tasks in this thesis. The first consists of developing an extensive, robust and easy-to-use game engine in which any type of artificial intelligence can be prototyped. The second task is to evaluate the adequacy of neural networks based agents to win simple grid-based games.

1.1 Game engine

Video games are obviously an attractive and fun domain, but more importantly it is a common application area for AIs [7] as they provide environments that are easy to simulate and they offer easy-to-collect data. This is why we will develop an engine that will help to prototype AIs to play simple games. Thanks to our framework, these AIs could be developed using any type of software development techniques, such as modelling languages, traditional lines of code, and even machine learning models.

Hence, the goal here is to develop a framework that allows rapid prototyping on game development as well as on AI development. This will be made possible by developing an extensive and robust game engine specialized in simple grid-based video games. The first difficulty of making such framework is to forecast the different types of games that will be available to implement. Indeed, even if we limit ourselves to grid-based games, many parameters have to be taken into account. Indeed, there exist multiple types of grid-based games; some can be turn-based while other can run in real-time, some can be played by one player only while some others need more players, etc... We list some important points that the framework must handle in order to be extensive to the different types of games that can be developed in it:

1. Handle turn-based and real-time games.
2. Handle different numbers of players.
3. Handle team-based games.
4. Handle competitive and cooperative games.
5. Handle different game rules.

Once the framework addresses all these points, we can consider it as extensive, but we want it to be robust and easy-to-use. We set a series of technical matters that we consider important for a game framework to be robust and easy-to-use.

1. Provide a secured way for the AIs to interact with the game and prevent cheating.
2. Make the game independent of the AIs' performance (i.e. make sure a slow AI does not slow down the game).
3. Provide a complete API for the AIs to communicate with.

Matching all these points will ensure that the framework will be easily reusable and offers a way to implement multiple grid-based games of all types.

1.2 Neural networks

Machine learning has become an important research topic over the last decades [8]. It brings intelligence in complex problems which would have been nearly impossible to solve without it, like classification problems, such as optical character recognition [9], [10], face classification [11], speech recognition [12], or feature-extraction problems [13], [14] in which the machine learning solution will try to analyse and differentiate a set of given data. At the moment, the main focus of research in this domain is on neural networks and their deep architectures [15], [16], which allow non-linear learning, giving much better results than classical learning methods in complex problems [3], [4], [10], [11].

We will explore this domain by developing AIs using neural networks learning techniques to reach the goal of the game for which they were developed. We will see further in this document that the results of neural networks-based techniques mainly depend on their learning phase, and in our case, the networks will have to learn first how not to lose the game and then how to win the game. To master those concepts, the models will need samples of games from which it can learn. This is why multiple techniques of data collection will be developed to collect data for each developed game and then provide it to the learning models.

The job of the generated AIs will be to predict the next action to perform for its controlled unit, knowing which actions have been performed this far by all the players. To validate these AIs, we will benchmark them by making the AIs fight against various types of other AIs using the game engine previously mentioned.

Part II
State Of The Art

The state of the art concerning our project can be seen below. First, we will detail some existing open-source game engines and explain the purpose and the role of our framework amongst them. Secondly, the main point is to introduce the different neural network architectures that will be considered during this thesis and to give some examples of machine learning applied to video games.

1 Game engines

There exist several game engines, many are commercial and some are open-source and free. The main open-source game engines and frameworks will be referenced below, and, for each, we will explain its main purpose. We did not introduce commercial game engines as our own will be open source.

Pygame

`Pygame` [17] is a cross-platform framework using `Python` and `SDL` [18], which is a low-level library providing hardware access written in `C` and natively uses `C++`. It is not a game engine by itself as it does not provide any game mechanics, except collision detection between two sprites, but it provides crucial methods and functions to develop games, such as hardware-accelerated graphics drawing, sound handling, frames per second limitation, windows management, and many more.

GODOT

The `GODOT game engine` [6] is an open-source and high-level game engine using its own scripting language: the `GDScript` (even though it can use `C++` to develop games). `GDScript` is a dynamically typed programming language and uses a syntax similar to Python, but a bit more verbose. `GODOT` can be used to develop 2D or 3D games on any platform (even web and mobile). One of the big advantages of this engine is that it comes with a Project Manager that includes a GUI designer and a graphical 2D/3D scene editor. But one drawback of the framework, which is a matter of habit, is that it uses a specific node-based[1] programming paradigm that requires a bit of practice to handle correctly.

MonoGame

The `MonoGame` engine is based on .NET libraries. It can be used to develop cross-platform using the `C#` programming language. It is the most known[2] open-source cross-platform game engine on GitHub using `C#` and .NET packages.

[1]For more details, see the docs: http://docs.godotengine.org/en/stable/tutorials/step_by_step/scenes_and_nodes.html

LibGDX

The LibGDX [5] framework is an open-source high-level game engine that can be used in Java and Python. It provides several useful methods to develop games, including settings handling, advanced input and sensors handling (accelerometer, vibrator, ...). Games can be developed and distributed on multiple platforms (including web and mobile). LibGDX is specialized in 2D games, provides many examples and tutorial, and is the favourite 2D game engine on GitHub[2].

Cocos2d-X

Cocos2d-x [19] is a cross-platform framework based on the functioning of Cocos2d-iphone, which is one of the most known free game engine used for the iPhone. The engine can be used to develop games using C++ or JavaScript. It provides graphics and multimedia handling, a physics engine, and many other interesting features.We can note that it is the second favourite game engine on GitHub[2].

RTS engines

Real-time strategy games have got plenty of open source game engines. Some of them are modelled on a commercial video game, like openAGE [20], which is based on *Age Of Empires* or openRA [21], mimicking the engine of *Command & Conquer: Red Alert*. There are also more generic game engines specialized in RTS development, like openRTS [22] and spring [23].

Web-oriented engines

There are three main game engines focusing on web development using technologies like javascript and HTML5. The first, turbulenz [24] provides both a low-level API and a high-level API, making it rich in functionalities and extensible. It focuses on high performance and asynchronous loading. The second, texttsuperpowers [25], is mainly a intelligent game development environment that provides powerful and intuitive tools as well as a real-time collaboration kit for development. Finally, GDevelop gives a way to develop games that does not require coding skills. Indeed, the engine is built to develop simple games, like platform games. Its main purpose is to develop HTML5 games, but a C++ core has been developed in order to build native games. The main setback of this engine is the AI development, which focus on hard-coded behaviours.

2 Machine learning in video games

Traditionally, games AIs use scripts, implying that the agent behaviour has to be hard-coded by the developers. To develop challenging agents using this technique needs a deep understanding of the mechanics inside the games. Furthermore, the resulting scripts are

[2]Highest number of stars in March 2017: https://github.com/showcases/game-engines?s=stars

often so complex that it becomes impossible to modify or patch if needed. Those limitations led to the use of simulation-based decision-taking algorithm such as the Min/Max algorithm with Alpha/Beta pruning (detailed in [26]) or the Monte Carlo Tree Search (MCTS) (some of its implementation is detailed in [27] and [28]). But these algorithms cannot be applied to all the games. Indeed, these algorithms need to know all the possible actions of every player, which is not always possible following the game. Furthermore, the simulation-based algorithms are limited in performance because they need thousands of game simulations in order to take a decision.

To solve those limitations, machine learning was applied to video games [1][2]. Machine learning techniques aim to use a large amount of data (and optionally metrics computed from the data) to create and train a data model that suits the decision-taking problem. This model is then usually validated on test data, then on the game itself. In the following sections, we will introduce some examples of the use of machine learning in video games.

2.1 Virtual learning

Virtual learning relies on game data accessible through an API that is sometimes provided by the game itself, and sometimes created for the purpose of developing AIs for that game. We can distinguish two topics in virtual game learning: shallow learning and deep learning.

Shallow learning
The shallow learning methods include all the classical methods of machine learning such as k-nearest neighbours algorithm [29][30], M5 [31] or C4.5 [32]. Shallow learning techniques have been applied to video games learning since Arthur Samuel pioneered the research by applying machine learning to the game of checkers [33][34]. His research led the way to complex techniques, like TD-gammon [35] that learns how to play Backgammon using temporal differences, or even the ones developed in [36] and [37] that aimed on developing real-time strategy-adapting AIs that play respectively Starcraft and Hearthstone.

Deep learning
The particularity of deep learning is that there are several layers of learning units helping the model to learn complex structures of data where shallow learning would have failed. The main data structures used in deep learning are deep neural networks (DNN), which are detailed in section 3. In some cases, DNNs were shown to be more powerful and robust than classical techniques in video games [38]. They can be used to make an AI imitate the ways of humans playing [39], to create AIs that evolve while playing [40] or even to develop AIs that win games that were a priori a decade away from being mastered like the game of Go [41].

2.2 Visual learning

There exists a trend in video games machine learning, especially when using deep neural networks, that aims on learning how to play the game using visual sensors and a visual representation of the game. Some researches made AIs play on a visual representation of

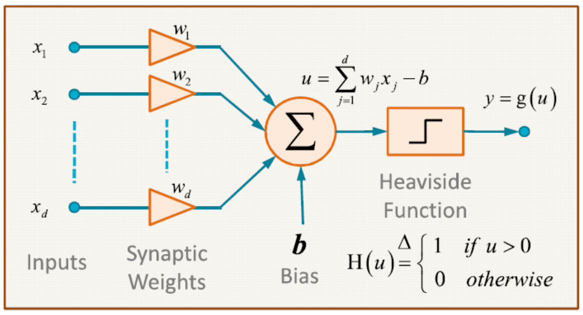
Atari games [42] and Super Mario [43]. Another research studied how learning from game frames can be reused to learn to play another game [44]. But using images to learn how to play a game is completely different from using game data coming straight from the game's API. This is why our thesis will not include these techniques.

3 Artifical neural networks

An artificial neural network is a data processing model of which the idea is inspired by the structure of the human brain. In the 1950's, researchers tried to understand what made the human brain that good to learn new things. Along the years, they came up with the creation of artificial neuron models. In the following sections, we will explain the principles behind artificial neural networks used to classify or to predict data. The inputs of those models will be data vectors, and their outputs will be probabilities to belong to a class (for the classifiers), or data vectors (for data prediction).

3.1 Artificial neuron model

The idea behind the artificial neuron model was to develop a model that handles data in a way relatively similar to the human brain's. It means that in an artificial neuron, an artificial synaptic weight w_i will be assigned to the ith input dimension, and a weighted sum will then be applied to prepare the output of the neuron. Once the weighted sum is performed, its result will be compared with a pre-set bias b. The result of that comparison is called u, and is given to a sign function, called the activation function, which will test if u is positive or negative. The result of this function classify the input data into one class or the other, resulting in a linear classifier usually called a Perceptron [45].



x_1
x_2
x_d
w_1
w_2
w_d
$u = \sum_{j=1}^{d} w_j x_j - b$
$y = g(u)$
Heaviside Function
b
Bias
Inputs
Synaptic Weights
$H(u) \overset{\Delta}{=} \begin{cases} 1 & if\ u > 0 \\ 0 & otherwise \end{cases}$

Figure 1: Scheme of a classical Perceptron model using a single artificial neuron
Source: [46]

To train this binary model, the synaptic weights must be adjusted until the output suits well to the data. To make sure of this, the data set is usually randomly divided into two subsets: the training subset and the validation subset. The training subset is used to adjust the weights, and the validation subset is used to make sure the model is well trained (i.e. the weights are well adjusted for the given data).

A multi-dimensional version of the binary Perceptron exists and consists in a neuron layer, which contains as many neurons as the number of output dimensions. The input is fully connected to the neurons, so that each neuron can benefit of every dimension of the input. In the case of a classifier, the outputs are probabilities for the input vector to belong to each class, and in the case of a predictor, the output is the predicted data vector.

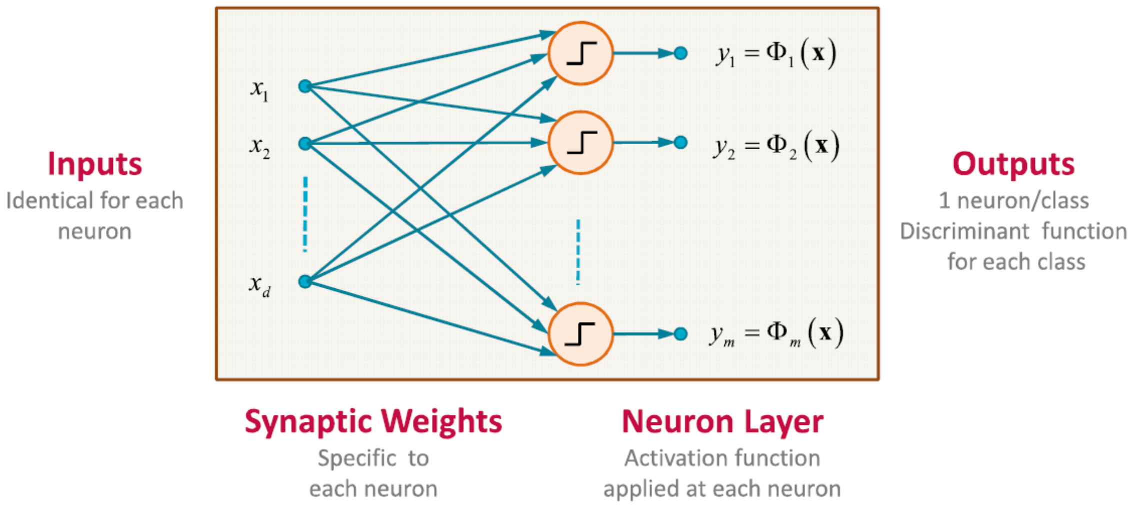

Inputs
Identical for each neuron
x_1
x_2
x_d
$y_1 = \Phi_1(\mathbf{x})$
$y_2 = \Phi_2(\mathbf{x})$
$y_m = \Phi_m(\mathbf{x})$
Outputs
1 neuron/class
Discriminant function for each class
Synaptic Weights
Specific to each neuron
Neuron Layer
Activation function applied at each neuron

Figure 2: Scheme of a multi-class Perceptron
Source: [46]

The training process of this multi-class Perceptron is a bit more complex than the previous one, as we must adjust the synaptic weights to minimize the error between the outputs of the Perceptron and the targets t_i that we set (the labels or the known output coming from the training data set). Any type of error measurement can be used in the training process, some are listed in section 3.3.4.

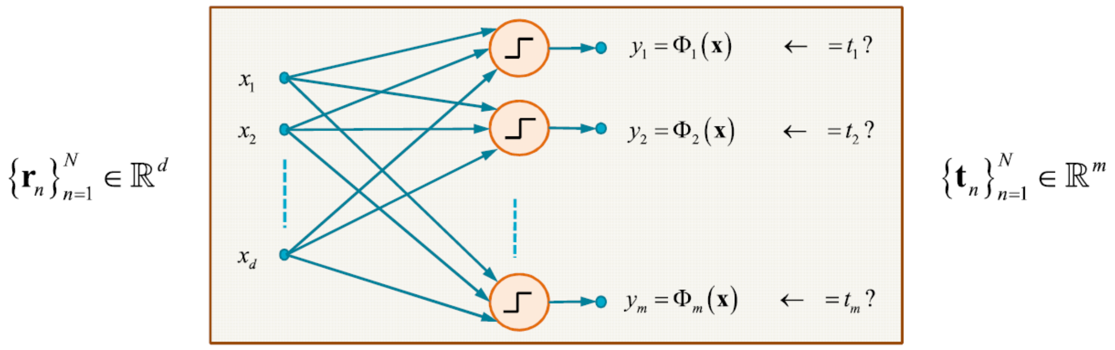

$\{\mathbf{r}_n\}_{n=1}^{N} \in \mathbb{R}^d$
x_1
x_2
x_d
$y_1 = \Phi_1(\mathbf{x}) \leftarrow = t_1 ?$
$y_2 = \Phi_2(\mathbf{x}) \leftarrow = t_2 ?$
$y_m = \Phi_m(\mathbf{x}) \leftarrow = t_m ?$
$\{\mathbf{t}_n\}_{n=1}^{N} \in \mathbb{R}^m$

Figure 3: Illustration of the training of a multi-dimensional Perceptron
using N training vectors, r_n representing a training vector,
$\Phi_i(x)$ being the Perceptron output values of the current target vector t_n.
Source: [46]

It is possible to make a Perceptron non-linear. Indeed, if we set the activation functions to non-linear functions, it results in non linear outputs. In practice, this is usually done using a sigmoid [47], a hyperbolic tangent function [48] or using rectified linear units (ReLU) [49].

3.2 Complex architectures

As the goal of the artificial neuron model was to mimic a biological neuron, artificial neural networks (ANN) will try to reproduce the way the biological neurons receive and share information. Typically, an ANN consists of multiple layers of artificial neurons: an input and an output layer, joined by hidden layers in between. We talk about deep neural networks (DNN) when it has several hidden layers of neurons (from a few dozens to hundreds of layers, each one made of several neurons). In this section, we will introduce some neural network architectures and their particularities.

3.2.1 Multi-layer perceptron

The most common neural network is called the Multiple-Layers Perceptron (MLP) [50]. Its activation functions are non-linear functions. It has been proven that a MLP with only one hidden layer can approximate any function [51], depending on the activation function(s) and the number of artificial neurons it uses. The more layers and neurons, the more there are parameters to adjust, and the more time it will take to train the model. Indeed, the weight of every neuron must be adjusted to suit the training targets. This is called the back-propagation, and the most known algorithm doing this is the gradient descent algorithm [52]. In practice, the model runs fast once it is trained, and there exist multiple training optimizers that focus on speeding up the training process. Some of those are detailed in section 3.3.3.

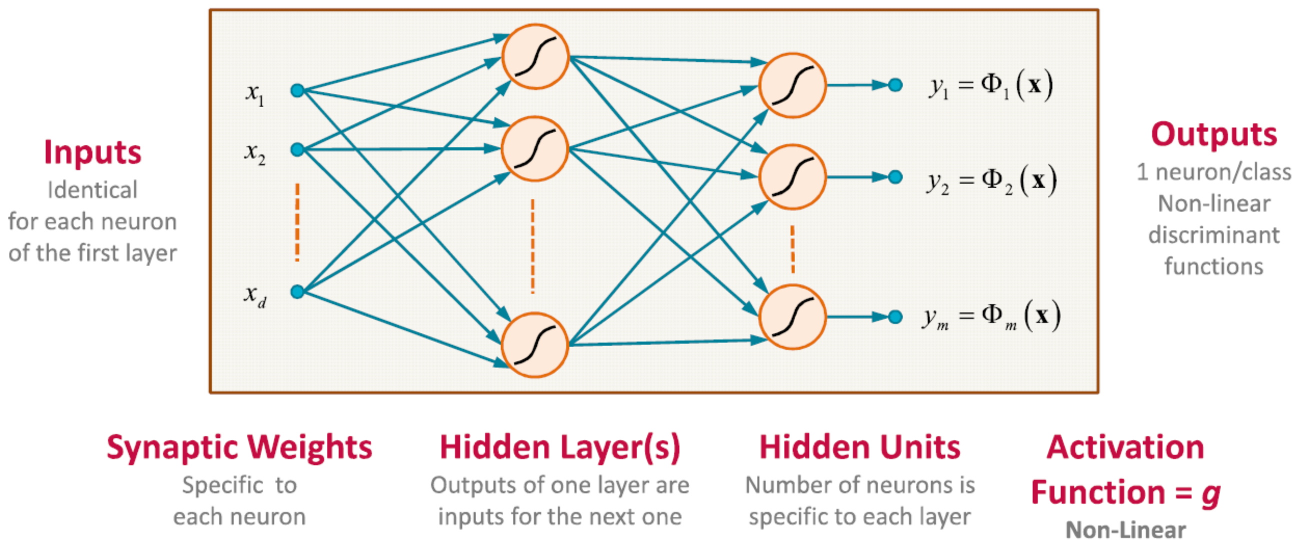

Inputs
Identical for each neuron of the first layer
x_1
x_2
x_d
$y_1 = \Phi_1(\mathbf{x})$
$y_2 = \Phi_2(\mathbf{x})$
$y_m = \Phi_m(\mathbf{x})$
Outputs
1 neuron/class
Non-linear discriminant functions
Synaptic Weights
Specific to each neuron
Hidden Layer(s)
Outputs of one layer are inputs for the next one
Hidden Units
Number of neurons is specific to each layer
Activation Function = g
Non-Linear

Figure 4: Illustration of a Multi-Layer Perceptron
Source: [46]

3.2.2 Recurrent neural networks

Recurrent neural networks [53] (RNN) are neural networks that have a loop in their structure, which allow them to keep a memory of the data that already passed through the network. This memory is usually used to predict or classify time-dependent or order-dependent data. The most used RNN architecture is the Long-Short Term Memory model [54] (LSTM), which has the ability to understand long term dependencies between inputs and outputs, as illustrated in figure 6.

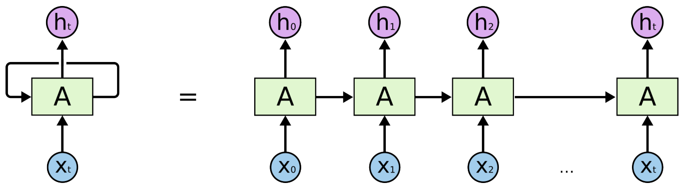


Figure 5: Illustration of a basic recurrent neural network and its "unrolled" equivalent on the right.
Source: Colah[3]

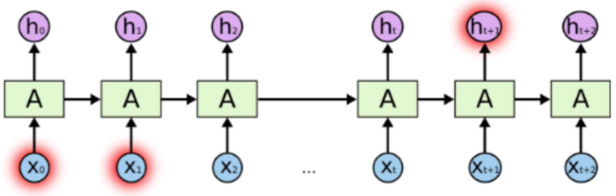


Figure 6: Illustration of the dependencies between inputs and outputs in an unrolled recurrent neural network.
Source: Colah[3]

We can directly see how this structure can be used in our thesis. Indeed, this model can predict data from past data. Applied to video games, it could predict the next action of a player using the previous actions of al the players. The long-term dependencies will be very useful as an action done in an early state of the game may lead to a winning or a losing action long after it.

3.2.3 Convolutional neural networks

Convolutional neural networks [55] is the name given to deep neural networks that have the particularity to force synaptic weights to have the same value in a certain neighbourhood of neurons. The main asset of these DNNs is that they can take raw data as input and extract interesting features automatically. These features are then sent to a more classical MLP to get

[3]http://colah.github.io/posts/2015-08-Understanding-LSTMs/

the wanted outputs. The training process of CNNs is one of the longest amongst all neural networks architectures. Indeed, the back-propagation starts at the end of the MLP and must adjust the weights of every layer through the deep network until the first layer of the CNN.

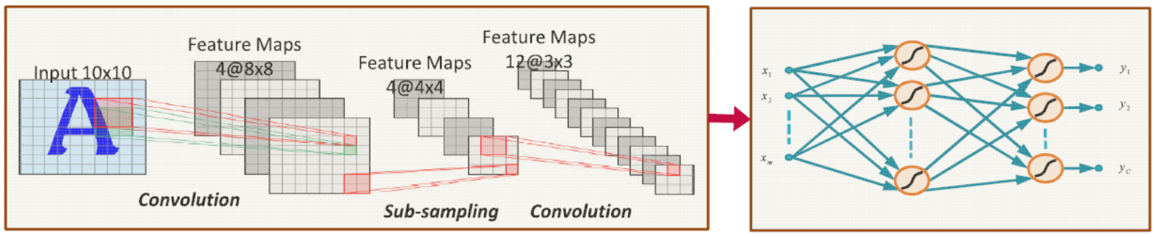


Figure 7: Illustration of a Convolutional Neural Network
Source: [46]

We can easily see how we could take advantage of the automatic features extraction to learn from video games. The idea is to give a raw representation of the game (in our case, its grid) as input of the CNN, and to get the next action to perform as output value.

3.3 Parameters

There are multiple parameters to set to build a neural network. We will enumerate some of them below.

3.3.1 Number of layers/neurons

There is no formal method to choose the number of hidden layers and neurons in a neural network. Nevertheless, these numbers are function of the problem complexity. The more complex the problem is, the more neurons and layers must be placed in the network. If these numbers are way too high for a simple problem, the resulting model will get terrible results, and vice versa. In [56], it was shown by experiments that for a complex enough problem, the more hidden layers and neurons, the more accurate the model. But like most of the machine learning models, making it too complex may lead the training process to over-fit the data.

3.3.2 Activation function

Varying the activation function of the layers allow to fine-tune the model. Any non linear activation function can approximate any output function, but the choice of the activation function seems to affect the accuracy of the approximation [57]. For example, periodic activation functions are likely to get better results than non-periodic activation functions to approximate a periodic function.

3.3.3 Training optimizers

The most common training optimization technique is the batch training. The subsequent parameter of this optimization is the batch size. This parameter affects the training process by updating the weights only when `batch_size` training vectors passed through the network. The higher the batch size, the faster the training will be, as it may cut off several computations. But as the weights are adjusted for a batch of vectors, increasing the batch size may affect negatively the results of the model, as it is shown in [58].

Additional training optimizers are usually used along with batch training, some examples are ADAM [59], NADAM [60], which is a variant of the first one, ADAGRAD [61] or ADADELTA [62]. All of these focus on reducing the amount of computations inside the back propagation algorithm. In practice, ADAM is usually a good choice compared to other optimizers [63]. Other optimization option and parameters can be found in [63].

3.3.4 Loss measure

As explained earlier, the training process focuses on adjusting the weights of the network to minimize the error between its result and the target vector. It means that changing the error measure affects the weights adjustment of each step. There exist multiple common loss metrics[4] One of the most used loss metric is the mean squared error (MSE):

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(p_i - t_i)^2$$

where N is the dimensionality of the prediction/target vector, p is the prediction of the model and t is the target vector that the model should have obtained as prediction

The problem is that using loss measure like the mean square error only does not suit when solving problems with classes. Indeed, let us take as example 3 totally independent classes in which we want to class data: `1`, `2` and `3`. Let us take the j^{th} training vector, with $t_j = 1$ and $p_j = 2$. The obtained error is $(2-1)^2 = 1$. But if we take the k^{th} training vector, with $t_k = 1$ and $p_j = 3$, the obtained error is $(3-1)^2 = 4$. Therefore, we obtain an error that is 4 times the first error, whereas the error should be the same, because in both cases we took the wrong class, and it is not worse to switch from class 1 to class 2 than to switch from class 1 to class 3. This problem is usually solved using other loss measures such as categorical cross entropy[4], a softmax activation function [64] for the output layer, along with a special class encoding called the *one-hot encoding*. This encoding aims to transform class labels into vectors containing only one 1, and multiple $0s$. At the end, each class should have a unique one-hot vector. If we take back our example with the 3 classes, the one-hot encoding of each these classes could be:

$$1 = [1, 0, 0]$$
$$2 = [0, 1, 0]$$
$$3 = [0, 0, 1]$$

The position of the 1 in the vectors does not really matter as long as a unique vector is assigned to each class. Using this encoding, the previous problem does not occur anymore, as each vector sum to 1.

[4] The details of the most common loss metrics can be found at: http://lasagne.readthedocs.io/en/latest/modules/objectives.html#loss-functions

Part III
Game engine

The goal of the framework is to offer an extensible platform to test different types of AIs on grid-based two-dimensional games. The engine is made to run grid-based games in which the players can compete or team-up to reach the goal of the game. Each developed game will provide an API that will be used by the AIs to obtain information on the current state of the game.

Our game engine is built on top of `Pygame` (see section 1 of part II), which is used mainly for its drawing methods and windows management. We did not consider using `GODOT`, `LibGDX`, `Cocos2D-x`, `MonoGame` or `turbulenz` (see section 1 of part II) because they are way too complex and hard to learn at first sight to develop simple grid-based games. Indeed, they include many concepts and generic methods that can work with any type of game, which makes them inherently complex and implies a significant technical overhead when developing very simple games. All this means, therefore, that our engine must provide a very simple way to develop a new grid-based game. Furthermore, we did not used `GDevelop` because its AIs development feature was too shallow and basic, and finally, we did not consider using the RTS engines, as RTS game mechanics were too different from simple grid-based games.

1 Software used

The software used to develop our game engine is described below.

Programming language

We chose `Python 3` as the programming language of our project. This choice has been motivated by the ability of rapid prototyping offered by `Python`. The syntax of `Python` is close to pseudo-code and is easy to read for any developer, which will make our framework easier to use. Additionally, `Python` comes with a wide variety of modules and libraries, such as `PyGame`, introduced in section 1 of part II, as well as `Keras` that provides neural networks related learning techniques.

Libraries

We detailed the `Python 3` libraries we used to develop our framework in table 1.

Library	Description
`Scipy` `0.18.1`	Required for matrix-based processing and geometrical operations. We chose it because it is the main[5] package in scientific computations for Python. It is used to get powerful and fast data structures such as KDTree [65]. It uses `Numpy` as its array computations library.
`Matplotlib` `1.5.3`	We chose `matplotlib` because of its `Path` class, which defines the path of a polygon. It is used to determine whether the clicked coordinates belong to a tile of the game. Furthermore, this library is known for its plotting abilities and is used to generate plots of the generated game data.
`Pathos` `0.2`	Used for its multi-processing abilities to run multiple instances of AIs to play the game. Where the `multiprocessing` built-in module of `Python` uses `pickle` to serialize the data passed to the new processes, `pathos` uses a module named `dill`, which is more robust than `pickle` because it can serialize anything. This is the only mutli-processing tool that uses `dill` and this is why we chose this library.
`Pygame` `1.9.3`	Graphics library used to draw the board and the units of the game. This is a well known library used for game development. It was chosen because it is a small library that has a lot of documentation and a wide range of examples.
`Pandas` `0.19.2`	Library used during the data collection to store the data into data frames and data sets. It was used because it was the library that fitted the best with the previously mentioned scientific libraries: `Numpy` and `Scipy`, as it uses the same data structures.

Table 1: Libraries used to develop our framework

2 Game definition

Below, we define the different concepts developed in the engine.

2.1 Logical loop

A game must run a loop numerous times per second. The default frame rate for our framework is 30 frames per second, because it is fast enough so that the game stays fluid and the inputs can be treated rapidly, with a latency of approximately 0.0333s. At each iteration of this loop, the game will perform the following actions:

- Check if any keyboard or mouse button has been pressed, and eventually dispatch it to the right controller(s).
- Check if any controller wants to perform a new action, and check if it is a correct action for the corresponding unit(s). If it is, add it to the move queue.
- Perform the next step of each move in queue.
- Check if there is a collision on the current game state.
- Draw the game board and its contents.
- Check if the game is finished, paused, or must go on. If the game ends, so does the loop. The game ends when there remains only one team including alive entities.

2.2 Entity

An entity is the smallest object that can be placed on a game board. A unit is an instance of an entity, with the particularity that it contains a list that maintains a reference to its own entities. An entity can be killed, but can have more than one life. An entity can be represented by a sprite, which is an image loaded into the game. It will be drawn at each iteration of the game's logical loop.

The entity's sprite is the only thing that can customize its graphics on the game. Indeed, the main loop will automatically draw the image/sprite of a entity place on the board. Apart from the sprite, two booleans define an entity: `active` and `controlled`. The first indicates whether the unit is used or not when checking if the game is finished. If the entity is not active, it will not be taken into account when the game will count how many entities of each team is alive. It can be used to draw independent entities. The second parameter indicates whether the unit is controlled or not. Hence, it must be set to True if the unit is directly linked to a controller.

When a unit accomplishes an action, the game engine will check if there is a collision on the game board. A collision is detected when two different units are located on the same tile on the board. When a collision happens, the default reaction is to take one life from each colliding unit. When creating a new game, the developer will choose if the game allows collisions within the same team and if a unit can suicide on its own entities. These are the only parameters to set to get the collision handling running.

2.3 Game Board

A board is made of multiple lines of tiles linked to one another. Each tile has four basic properties:

- An identifier that is unique in the board.
- A list that contains the identifier of every neighbour of this tile.
- A boolean indicating if this tile can be walked on
- A boolean indicating if this tile is deadly for any unit walking on it.

Every other effect that a unit can trigger by walking on a tile can be described when instantiating a move between two tiles. This instantiation is done within the game rules, defined when creating a new game.

2.4 Teams

A game developed within the engine is team-based. Indeed, we explained earlier that the game ends when it only remains one team alive in the game. Nevertheless, games that are not essentially team-based can be implemented too. To explain how we can do this, we distinguish three types of games:

1. War games – They are played by a minimum of two players belonging to different teams. The goal is to be amongst the last team alive.
2. Duel games – They are a subset of war games in which there is only one player per team.
3. Puzzle games – They are played by one or more players that collaborate to reach a common goal. These players belong to a common team.

While war games and duel games are handled naturally by the framework, puzzle games require a trick to be defined inside the engine. An example of puzzle game is Sokoban, of which the development is discussed later in section 5.3. The trick is to add a fake unit belonging to another team, that must be killed when the other players won the game. One can see this fake unit as being the game itself preventing the real unit(s) from winning.

3 Controllers

Each player is represented in-game by a controller, which will send actions to and receive information from the game. For a human player, the controller just binds inputs to actions, but for computer players, the controller must choose new moves according to new states of the game.

3.1 Independent processes

Each controller will interact with the game using a *ControllerWrapper*. Each wrapper runs a loop on an independent process, so that the risk that an AI slows down the game or cheats is minimized. To communicate with the game, the wrapper is sending events through inter-process *pipes*.

3.2 Events

To reduce useless computations, many of the game mechanics are waiting for an event to be triggered. This is why controllers will wait for an event coming from the game to choose a new move. Indeed, we considered that it was useless for a controller to choose a new move while nothing has changed on the game board since the last move selection.

To choose a new move, a bot controller must have a valid copy of the game state. To avoid sending useless copies of the game, these controllers will have a local copy of the game state, wrapped into an API, which will be updated by events sent by the main game when an action is effectively performed.

Two controllers belonging to the same team could communicate to reach a similar goal. Each controller can send a message to one of its teammates through an inter-process pipe. Then, the receiving controller can react if needed by selecting a new move. The interaction between controllers and the game engine is illustrated on figures 8 and 9.

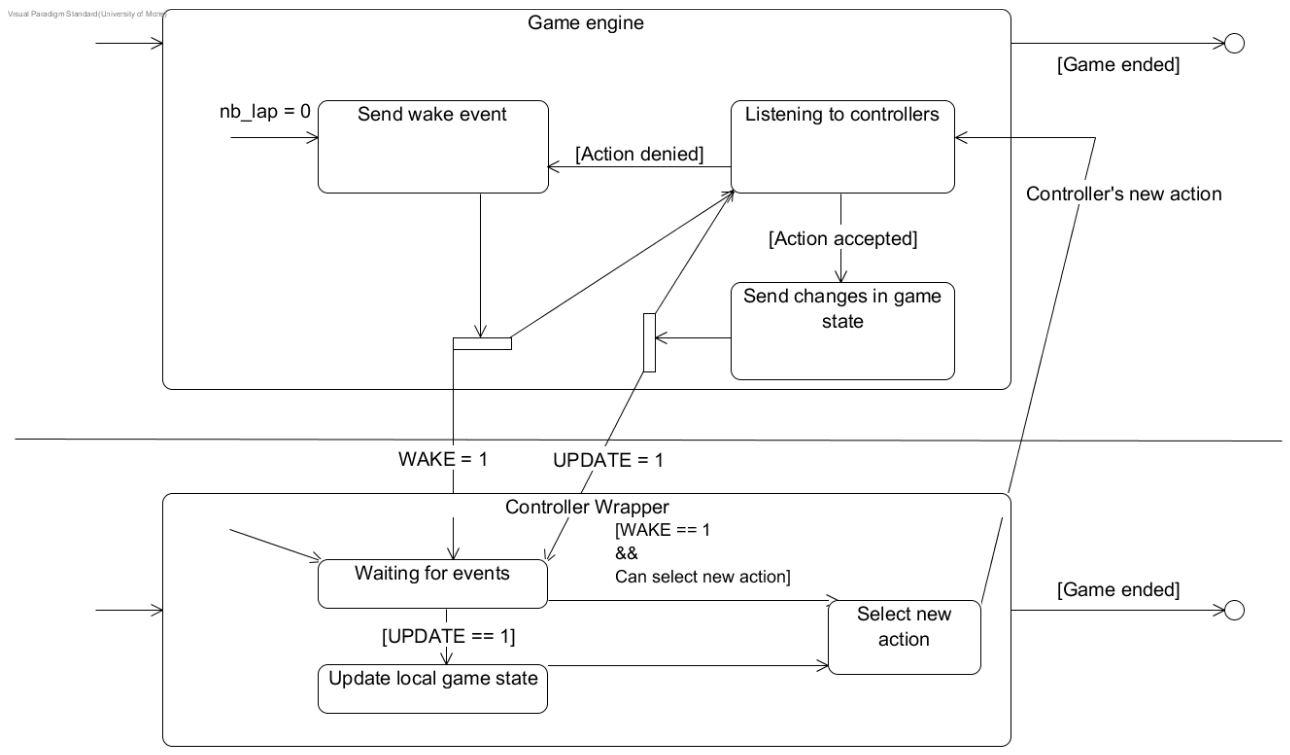


Figure 8: Illustration of the communication between the controllers and the game engine

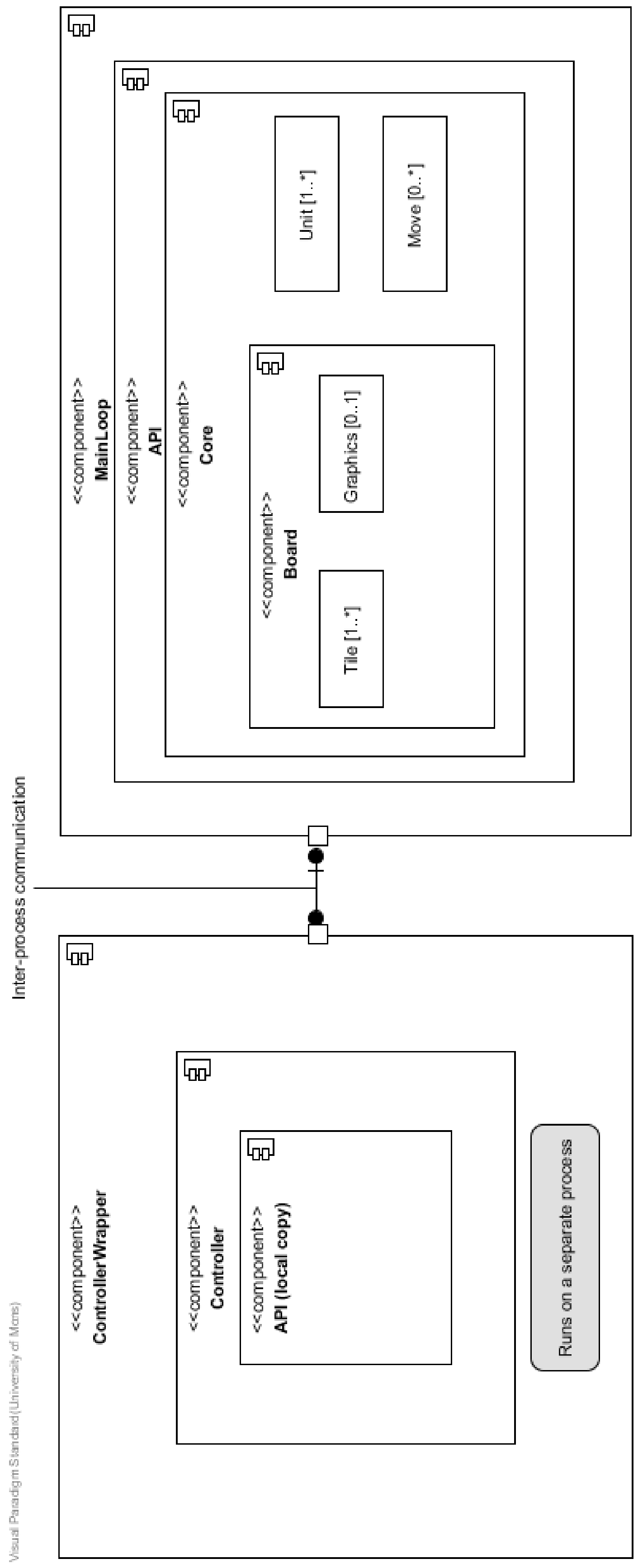


Figure 9: Component diagram representing the interaction between the main loop and one controller

4 Code insight

In this section, we will introduce the main classes that a developer must interact with when creating a new game. Every part of code that is not detailed here is detailed in the code[6] as typed comments.

4.1 Core

The `Core` class represents the game board with units placed on them. It defines methods that control the game execution. The basic `Core` class, of which the class diagram is on figure 10, is abstract and must be implemented when creating a new game. The methods and properties to implement are the following:

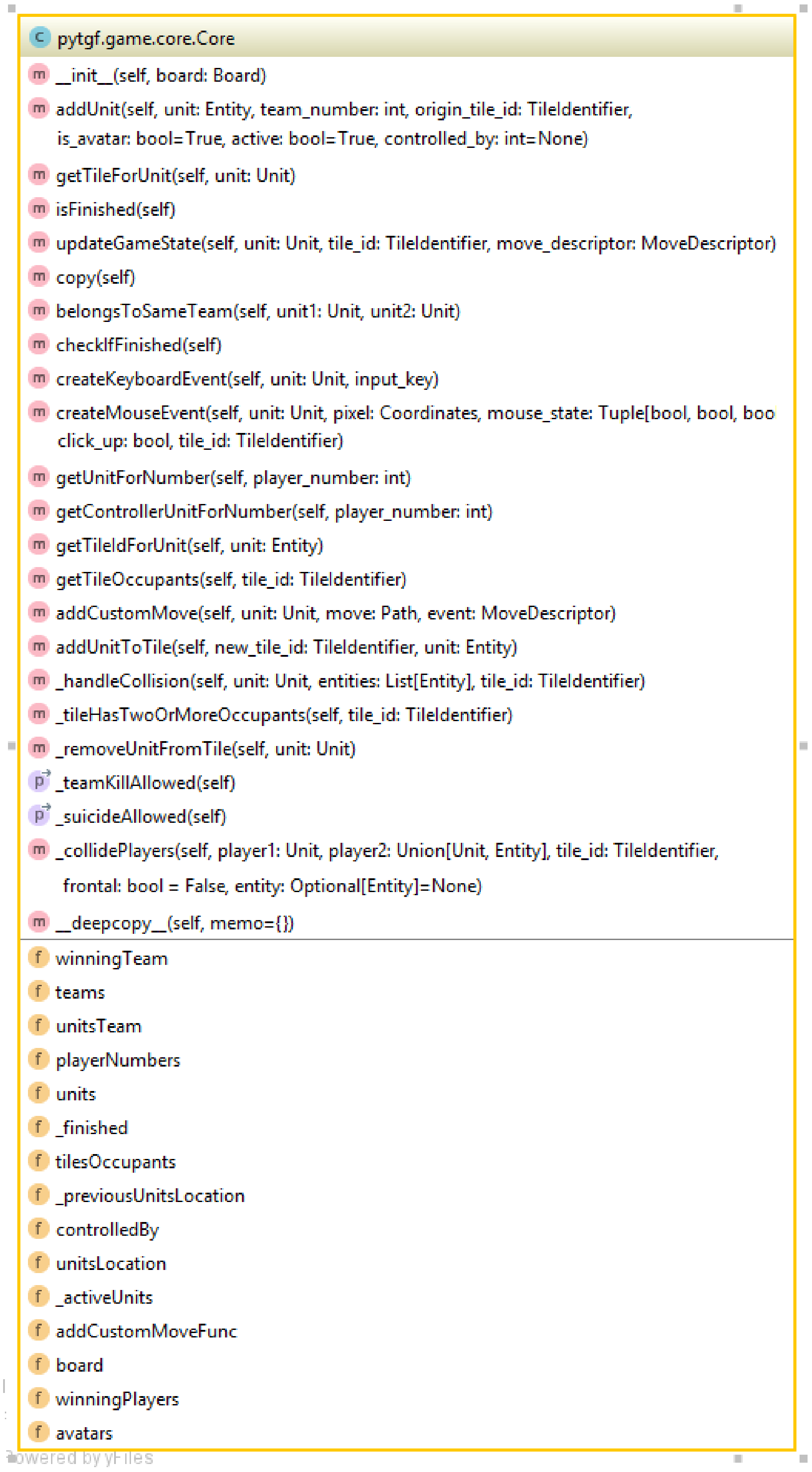
- `_suicideAllowed` – Property that must return a boolean: true if a unit can collide and suicide on its own entities, and false otherwise.
- `_teamKillAllowed` – Property that must return a boolean: true if a unit can collide on units and entities of its own team.
- `_collidePlayers` – Method that handles the collision between a unit and another entity. It has a default body defined in `Core`. By default, each time a unit collides onto another entity, they both lose one life if the collision is considered deadly (i.e. if the other entity belongs to another team or team kill is allowed, or if it belongs to that unit and suicide is allowed). Overriding this method allows to add game specific mechanics inside the game core.

4.2 API

Every new game inherits from a common API, which contains basic methods that suit to most grid-based games. This basic `API`, illustrated on figure 11 class is abstract, it is therefore required to implement it when creating a new game. The abstract methods that need implementation are the following:

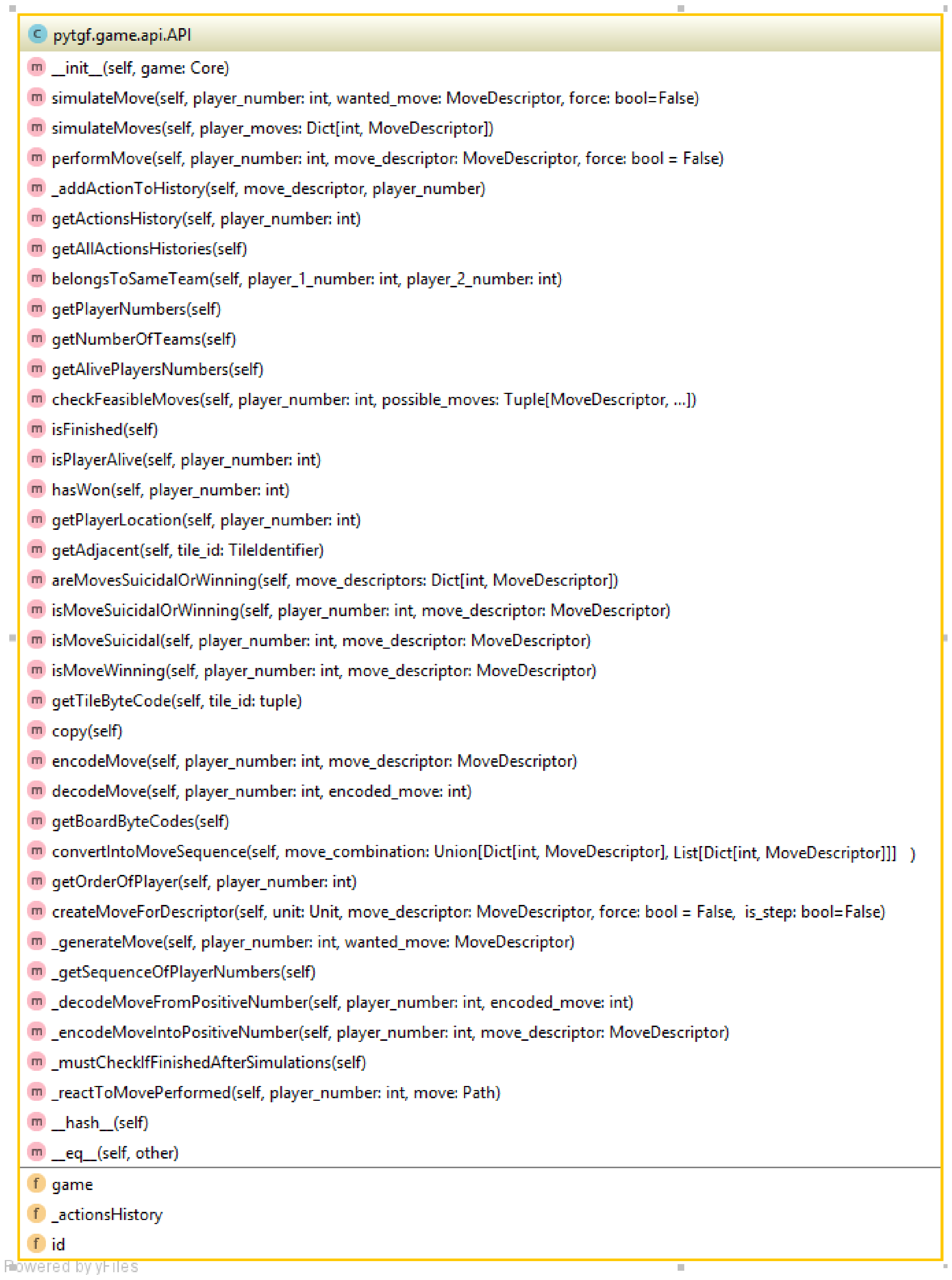
- `createMoveForDescriptor` – It is surely the most important method to implement when creating a new game inside the engine. Indeed, this method will instantiate a new action (also called move) for a given action descriptor. An action descriptor, or move descriptor, is a `Python` object or value that represents a move that a unit wants to do.
- `_encodeMoveIntoPositiveNumber` – This method is used when gathering data from a game state. It takes a move descriptor as input and must return a positive integer that represents the move.
- `_decodeMoveFromPositiveNumber` – This method does exactly the opposite of the previous one. It takes a positive integer as input and must return a move descriptor for the given player.

[6] Available on GitHub: https://github.com/Angeall/pyTGF



```
pytgf.game.core.Core
__init__(self, board: Board)
addUnit(self, unit: Entity, team_number: int, origin_tile_id: TileIdentifier,
 is_avatar: bool=True, active: bool=True, controlled_by: int=None)
getTileForUnit(self, unit: Unit)
isFinished(self)
updateGameState(self, unit: Unit, tile_id: TileIdentifier, move_descriptor: MoveDescriptor)
copy(self)
belongsToSameTeam(self, unit1: Unit, unit2: Unit)
checkIfFinished(self)
createKeyboardEvent(self, unit: Unit, input_key)
createMouseEvent(self, unit: Unit, pixel: Coordinates, mouse_state: Tuple[bool, bool, boo
 click_up: bool, tile_id: TileIdentifier)
getUnitForNumber(self, player_number: int)
getControllerUnitForNumber(self, player_number: int)
getTileIdForUnit(self, unit: Entity)
getTileOccupants(self, tile_id: TileIdentifier)
addCustomMove(self, unit: Unit, move: Path, event: MoveDescriptor)
addUnitToTile(self, new_tile_id: TileIdentifier, unit: Entity)
_handleCollision(self, unit: Unit, entities: List[Entity], tile_id: TileIdentifier)
_tileHasTwoOrMoreOccupants(self, tile_id: TileIdentifier)
_removeUnitFromTile(self, unit: Unit)
_teamKillAllowed(self)
_suicideAllowed(self)
_collidePlayers(self, player1: Unit, player2: Union[Unit, Entity], tile_id: TileIdentifier,
  frontal: bool = False, entity: Optional[Entity]=None)
__deepcopy__(self, memo={})
winningTeam
teams
unitsTeam
playerNumbers
units
_finished
tilesOccupants
_previousUnitsLocation
controlledBy
unitsLocation
_activeUnits
addCustomMoveFunc
board
winningPlayers
avatars
```

Figure 10: Class diagram of the `Core` class



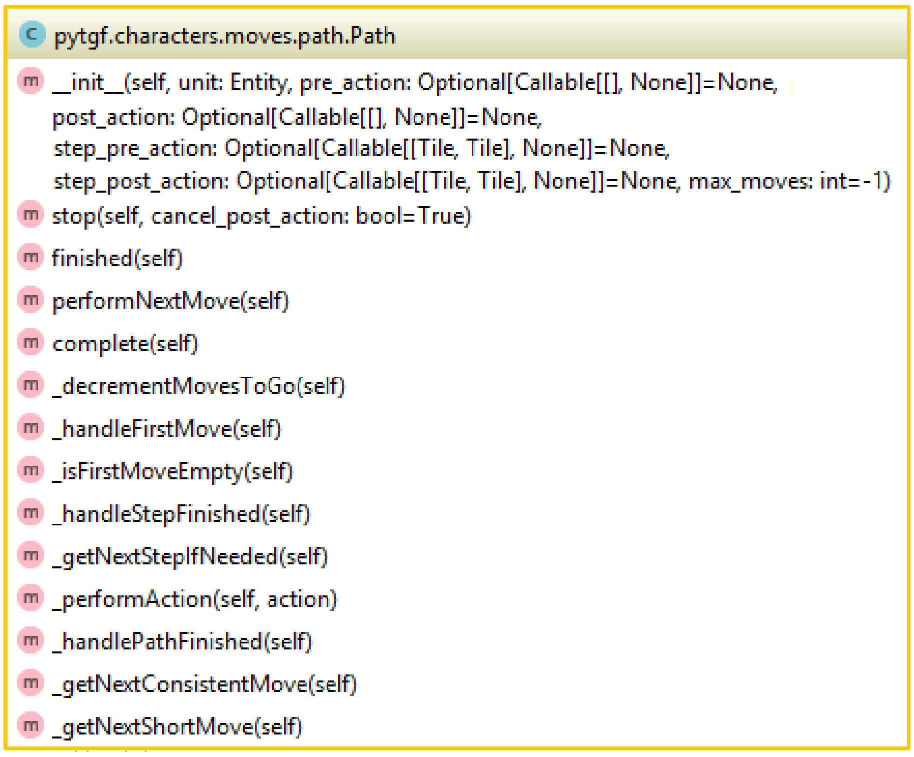
pytgf.game.api.API

```
__init__(self, game: Core)
simulateMove(self, player_number: int, wanted_move: MoveDescriptor, force: bool=False)
simulateMoves(self, player_moves: Dict[int, MoveDescriptor])
performMove(self, player_number: int, move_descriptor: MoveDescriptor, force: bool = False)
_addActionToHistory(self, move_descriptor, player_number)
getActionsHistory(self, player_number: int)
getAllActionsHistories(self)
belongsToSameTeam(self, player_1_number: int, player_2_number: int)
getPlayerNumbers(self)
getNumberOfTeams(self)
getAlivePlayersNumbers(self)
checkFeasibleMoves(self, player_number: int, possible_moves: Tuple[MoveDescriptor, ...])
isFinished(self)
isPlayerAlive(self, player_number: int)
hasWon(self, player_number: int)
getPlayerLocation(self, player_number: int)
getAdjacent(self, tile_id: TileIdentifier)
areMovesSuicidalOrWinning(self, move_descriptors: Dict[int, MoveDescriptor])
isMoveSuicidalOrWinning(self, player_number: int, move_descriptor: MoveDescriptor)
isMoveSuicidal(self, player_number: int, move_descriptor: MoveDescriptor)
isMoveWinning(self, player_number: int, move_descriptor: MoveDescriptor)
getTileByteCode(self, tile_id: tuple)
copy(self)
encodeMove(self, player_number: int, move_descriptor: MoveDescriptor)
decodeMove(self, player_number: int, encoded_move: int)
getBoardByteCodes(self)
convertIntoMoveSequence(self, move_combination: Union[Dict[int, MoveDescriptor], List[Dict[int, MoveDescriptor]]]  )
getOrderOfPlayer(self, player_number: int)
createMoveForDescriptor(self, unit: Unit, move_descriptor: MoveDescriptor, force: bool = False,  is_step: bool=False)
_generateMove(self, player_number: int, wanted_move: MoveDescriptor)
_getSequenceOfPlayerNumbers(self)
_decodeMoveFromPositiveNumber(self, player_number: int, encoded_move: int)
_encodeMoveIntoPositiveNumber(self, player_number: int, move_descriptor: MoveDescriptor)
_mustCheckIfFinishedAfterSimulations(self)
_reactToMovePerformed(self, player_number: int, move: Path)
__hash__(self)
__eq__(self, other)
```

```
game
_actionsHistory
id
```

Powered by yFiles

Figure 11: Class diagram of the `API` class



```
pytgf.characters.moves.path.Path
__init__(self, unit: Entity, pre_action: Optional[Callable[[], None]]=None,
post_action: Optional[Callable[[], None]]=None,
step_pre_action: Optional[Callable[[Tile, Tile], None]]=None,
step_post_action: Optional[Callable[[Tile, Tile], None]]=None, max_moves: int=-1)
stop(self, cancel_post_action: bool=True)
finished(self)
performNextMove(self)
complete(self)
_decrementMovesToGo(self)
_handleFirstMove(self)
_isFirstMoveEmpty(self)
_handleStepFinished(self)
_getNextStepIfNeeded(self)
_performAction(self, action)
_handlePathFinished(self)
_getNextConsistentMove(self)
_getNextShortMove(self)
```

Figure 12: Class diagram of the `Path` class

4.2.1 Moves

A move is represented by two classes in the engine: `ShortMove` and `Path`, illustrated on figure 12. A `Path` instance is actually a sequence of `ShortMove`. A `Path` object can take multiple callbacks actions that will be triggered at different moments during the path:

- `pre_action` – Function that does not take any parameter. It is triggered before the path begins.
- `post_action` – Function that does not take any parameter. It is triggered once the path is finished.
- `pre_step_action` – Function that takes two parameters: the previous tile and the next tile on which the unit will be located. It is triggered each time a new `ShortMove` begins.
- `post_step_action` – Function that takes two parameters: the previous tile and the new tile on which the unit will be located. It is triggered each time a `ShortMove` ends.

All these functions are optional, and are ignored if set to None. Using these callbacks, any interaction with the game and its tiles can be made at any time.

4.3 Controllers

When creating a new game, one must implement one controller class that is unique to its game (e.g. `LazerBikePlayer` for the Lazerbike game of section 5.1). This class basically does

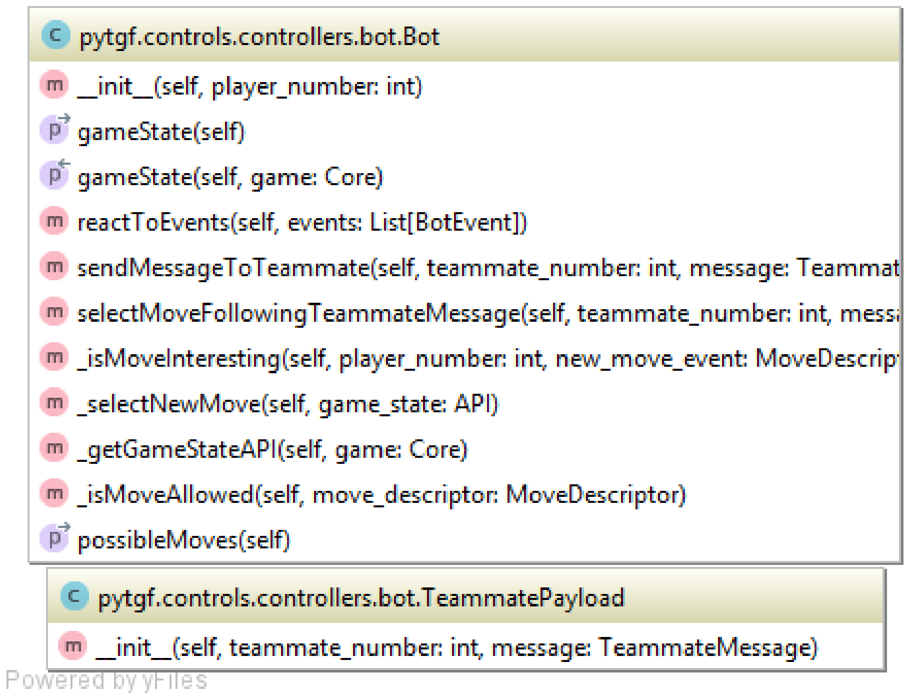

pytgf.controls.controllers.bot.Bot
__init__(self, player_number: int)
gameState(self)
gameState(self, game: Core)
reactToEvents(self, events: List[BotEvent])
sendMessageToTeammate(self, teammate_number: int, message: Teammat
selectMoveFollowingTeammateMessage(self, teammate_number: int, messa
_isMoveInteresting(self, player_number: int, new_move_event: MoveDescrip
_selectNewMove(self, game_state: API)
_getGameStateAPI(self, game: Core)
_isMoveAllowed(self, move_descriptor: MoveDescriptor)
possibleMoves(self)
pytgf.controls.controllers.bot.TeammatePayload
__init__(self, teammate_number: int, message: TeammateMessage)
Powered by yFiles

Figure 13: Class diagram of the `Bot` class

nothing but adding a new type of controller that the AI selector can recognise when building the GUI of the game. Then, two subclasses can be implemented from this base class: the bot controller and the human controller classes.

4.3.1 Bot controller

A new bot controller in the game can be created by extending the `Bot` class, see figure 13, and the basic controller type for the chosen game. The main methods to override are:

- `_isMoveInteresting` – This method must return a boolean that indicates whether the game state update must trigger the `_selectNewMove` method or not. Can be used to limit the useless work. For example, in the Lazerbike game (see section 5.1), a new game state is interesting only if all the alive players have moved. If we received the move of only one of the three players, it is pointless to select a new move, as the game state is inconsistent. This problem does not occur in the Sokoban example (see section 5.3), as the players have no obligations to move at the same time.

- `possibleMoves` – This method can return the moves allowed for the bot controller if they can be listed. In the other case, it can optionally return a hint on the type of moves that the controller can perform. This property is not mandatory by itself, it is just an helper method so that AIs developer can rapidly know what they must return by implementing `_selectNewMove`.

- `_getGameStateAPI` – This method takes a `Core` object and must return an `API` object. This is where the AI developer can instantiate its specific API for its game from the given core object.
- `_isMoveAllowed` – This method returns true if a move descriptor is legal for the unit controlled by this controller. It must check if the move is understandable by the game core, but it is not necessary to check if the move is doable considering the current game state.

Then, to develop an AI from this bot, we must implement two more methods:

- `_selectNewMove` – Given a new game state, the AI must return a new move descriptor to send to the game.
- `selectMoveFollowingTeammateMessage` – This method is triggered when a message coming from a teammate is received. It optionally returns a new move descriptor to send to the game, in reaction to the message.

4.3.2 Human controller

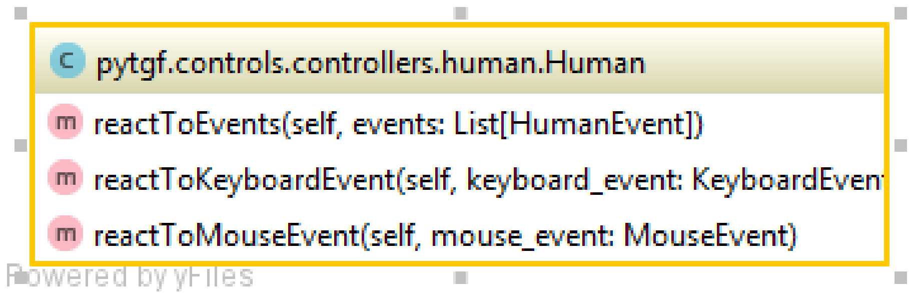

pytgf.controls.controllers.human.Human
reactToEvents(self, events: List[HumanEvent])
reactToKeyboardEvent(self, keyboard_event: KeyboardEven
reactToMouseEvent(self, mouse_event: MouseEvent)
Powered by yFiles

Figure 14: Class diagram of the `Human` class

Human controllers must bind inputs to return move descriptors. The class methods are listed on figure 14. The methods to implement are the following:

- `reactToKeyboardEvent` - method that will handle a keyboard input and transform it if needed in a move descriptor.
- `reactToMouseEvent` - method that will handle a mouse input and transform it if needed in a move descriptor.

The human controller class has been made generic so that any other type of input can be added later. Indeed, The logical loop of the game will send events to the controller, and it is the controller that will dispatch the right information to the right method.

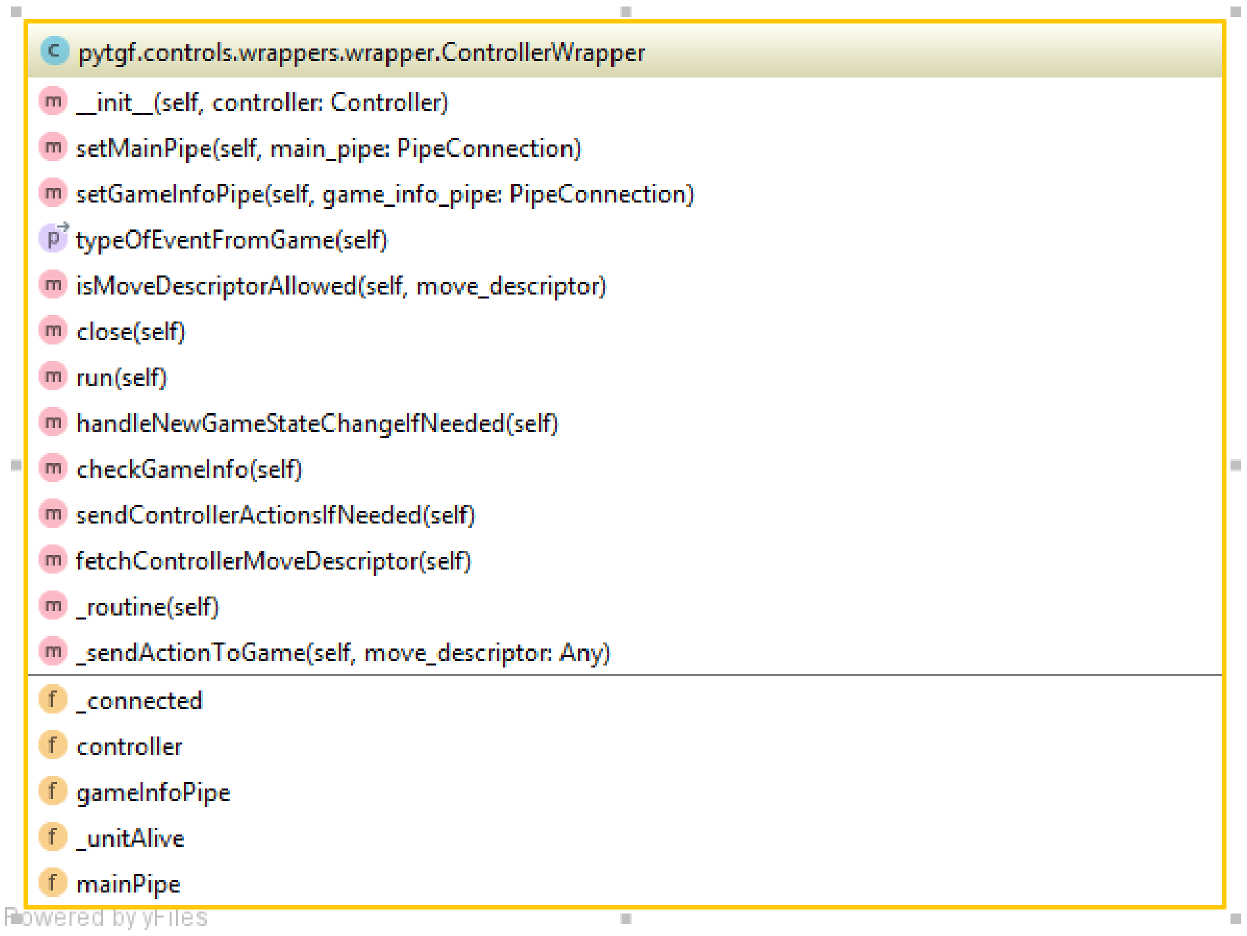

pytgf.controls.wrappers.wrapper.ControllerWrapper
__init__(self, controller: Controller)
setMainPipe(self, main_pipe: PipeConnection)
setGameInfoPipe(self, game_info_pipe: PipeConnection)
typeOfEventFromGame(self)
isMoveDescriptorAllowed(self, move_descriptor)
close(self)
run(self)
handleNewGameStateChangeIfNeeded(self)
checkGameInfo(self)
sendControllerActionsIfNeeded(self)
fetchControllerMoveDescriptor(self)
_routine(self)
_sendActionToGame(self, move_descriptor: Any)
_connected
controller
gameInfoPipe
_unitAlive
mainPipe

Figure 15: Class diagram of the `Wrapper` class

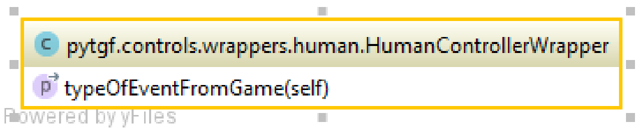
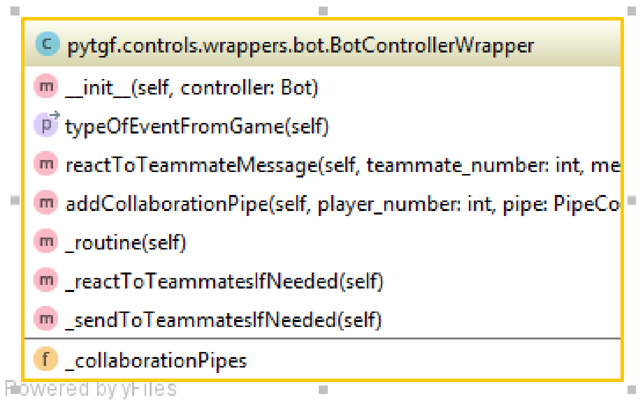
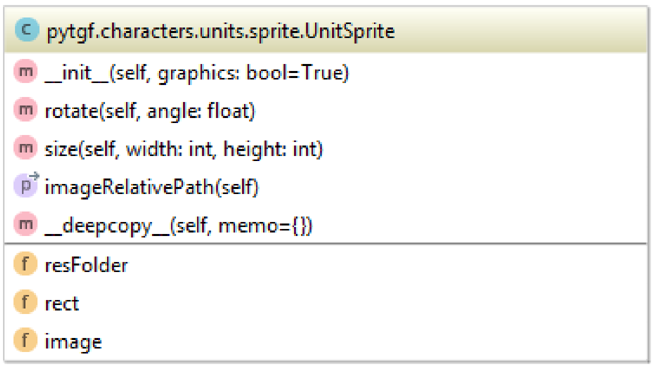

pytgf.controls.wrappers.human.HumanControllerWrapper
typeOfEventFromGame(self)

Figure 16: Class diagram of the `HumanWrapper` class



pytgf.controls.wrappers.bot.BotControllerWrapper
m __init__(self, controller: Bot)
p typeOfEventFromGame(self)
m reactToTeammateMessage(self, teammate_number: int, me
m addCollaborationPipe(self, player_number: int, pipe: PipeCo
m _routine(self)
m _reactToTeammatesIfNeeded(self)
m _sendToTeammatesIfNeeded(self)
f _collaborationPipes

Figure 17: Class diagram of the `BotWrapper` class



pytgf.characters.units.sprite.UnitSprite
m __init__(self, graphics: bool=True)
m rotate(self, angle: float)
m size(self, width: int, height: int)
p imageRelativePath(self)
m __deepcopy__(self, memo={})
f resFolder
f rect
f image

Figure 18: Class diagram of the `UnitSprite` class

4.4 Controller wrapper

A new controller wrapper is needed when creating a new game. It must be created the same way than the controllers. Indeed, one must first create a class that inherits from the base `ControllerWrapper` class, illustrated on figure 15, and then create two classes, one inheriting from the newly created controller wrapper and from the `HumanControllerWrapper` class, and another, again inheriting from the newly created controller wrapper but also from the `BotControllerWrapper`. The only method that needs to be implemented is `isMoveDescriptorAllowed`, that must return true if the move descriptor will be understood by the `createMoveFromDescriptor` method of the game's API, and false otherwise.

4.5 Units and Entities

Finally, it remains to define the units that will be used in the game. The main point of overriding the `UnitSprite` class is to give graphics to the created units. Then, the `Entity` and `Unit` classes can be implemented to link the newly created sprites with the units of the game, and optionally to customize the number of lives, and add custom attributes to the new units.

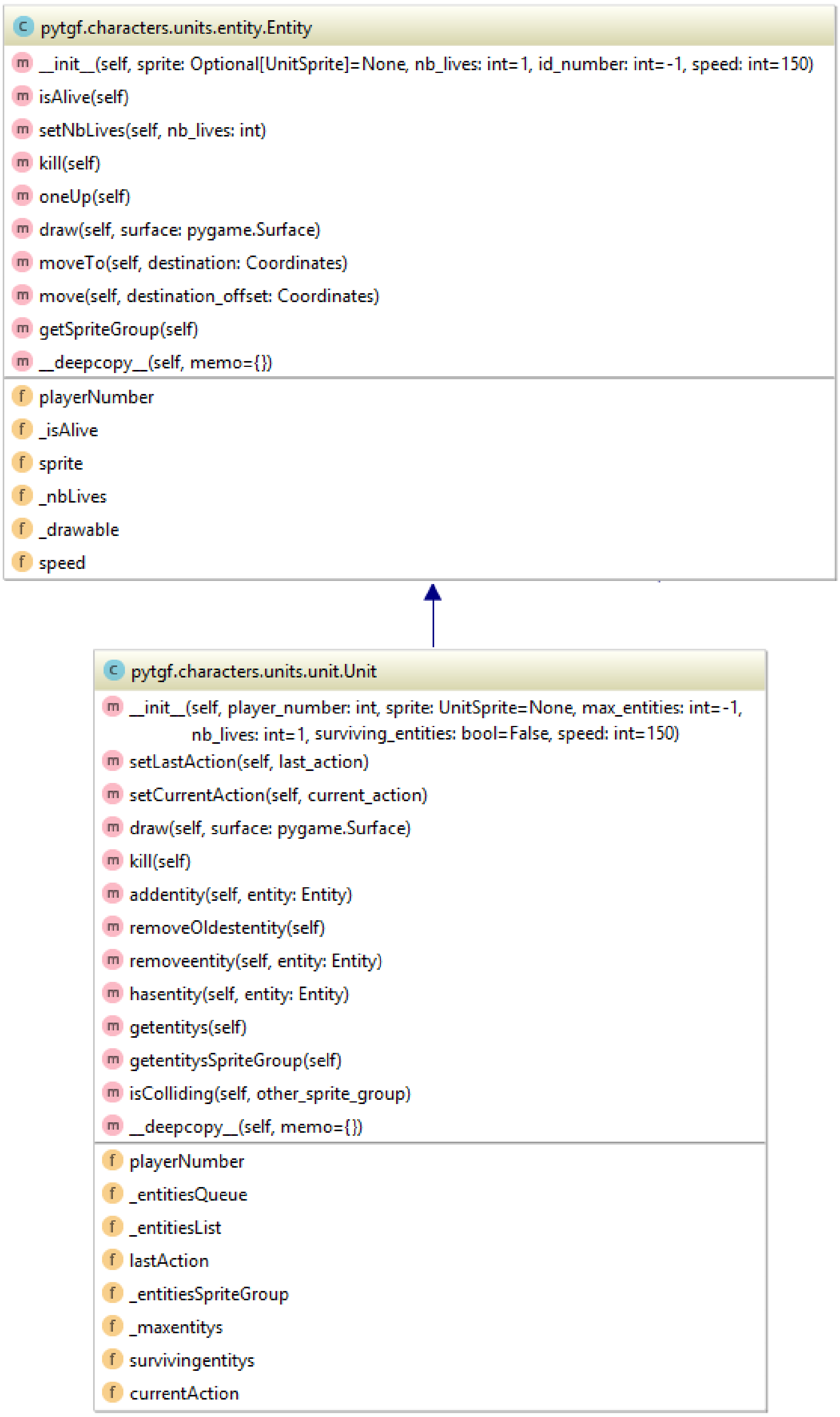

C pytgf.characters.units.entity.Entity
m __init__(self, sprite: Optional[UnitSprite]=None, nb_lives: int=1, id_number: int=-1, speed: int=150)
m isAlive(self)
m setNbLives(self, nb_lives: int)
m kill(self)
m oneUp(self)
m draw(self, surface: pygame.Surface)
m moveTo(self, destination: Coordinates)
m move(self, destination_offset: Coordinates)
m getSpriteGroup(self)
m __deepcopy__(self, memo={})
f playerNumber
f _isAlive
f sprite
f _nbLives
f _drawable
f speed
C pytgf.characters.units.unit.Unit
m __init__(self, player_number: int, sprite: UnitSprite=None, max_entities: int=-1, nb_lives: int=1, surviving_entities: bool=False, speed: int=150)
m setLastAction(self, last_action)
m setCurrentAction(self, current_action)
m draw(self, surface: pygame.Surface)
m kill(self)
m addentity(self, entity: Entity)
m removeOldestentity(self)
m removeentity(self, entity: Entity)
m hasentity(self, entity: Entity)
m getentitys(self)
m getentitysSpriteGroup(self)
m isColliding(self, other_sprite_group)
m __deepcopy__(self, memo={})
f playerNumber
f _entitiesQueue
f _entitiesList
f lastAction
f _entitiesSpriteGroup
f _maxentitys
f survivingentitys
f currentAction

Figure 19: Class diagram of the `Unit` class

5 Implemented games

5.1 Lazerbike

The first game that was developed within the framework is a game where players control motorbikes that leave laser traces on their way. A bike that crash onto a laser trace or into another bike is destroyed. A bike running into a wall is destroyed as well. The goal is to be amongst the last team alive. To do so, a player can set a trap to others using its laser traces.

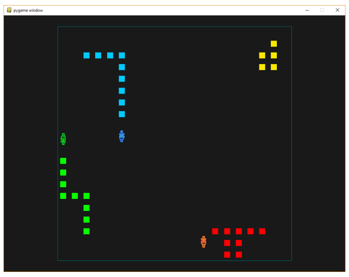


Figure 20: Example of a 4-players lazerbike game,
in which the yellow player is dead

This game was the subject of a contest held by Google in which the contestants had to develop a competitive AI to win the game. The winner of the contest, `A1k0n`[7], used advanced graph theory to develop its AI.

5.2 Connect 4

The well-known board game, Connect 4, consists of a vertically suspended rectangular grid of circles with six lines and seven columns. When the game starts, the players choose a color and then take turns dropping the discs into a non-full column of the board. The disk falls and places itself on the lowest position available in the column. The winner is the player that manages to align four pieces of his own colour in a row (horizontally, vertically or diagonally).

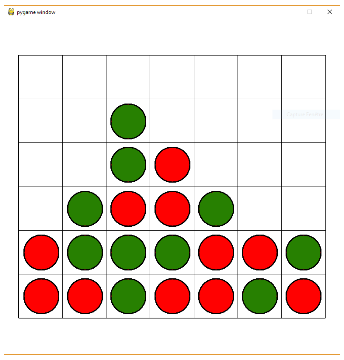


Figure 21: Example of a connect 4 game in progress.

[7]The details of its algorithm can be found on: https://www.a1k0n.net/2010/03/04/google-ai-postmortem.html

To comply with the teams and units base of the framework, invisible non-active units, called `BottomUnits`, are placed on the lowest space available for each column. Once one column is full, its corresponding `BottomUnit` is removed from the game. Indeed, they will be used in the collision detection between a disc unit and these fake units. Each disc is an uncontrolled unit belonging to the player's team. These fake and invisible units that are `BottomUnits` are illustrated on figure 22

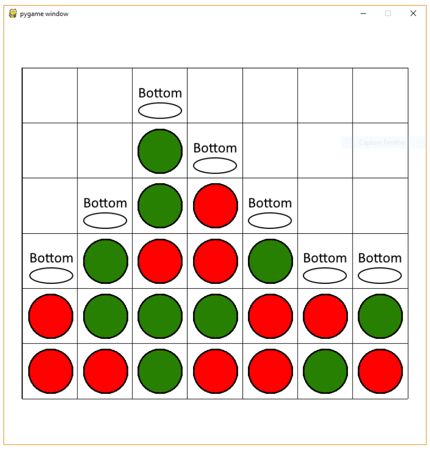


Figure 22: Illustration of the "Bottom" fake units on the Connect 4 board

This game is solved in terms of AI. Indeed, Victor Allis [66] and James D. Allen [67] both found within the same year a winning strategy that works every time for the first player to play. But these strategies are very hard to follow, as there is almost every time only one winning move for each game state.

Even if it seems like a simple game with a small grid of 42 tiles and two players, this game has a huge number of possible game states. it has been shown [68] that there are $4\ 531\ 985\ 219\ 092$ different possible states, and among these, there are $1\ 905\ 333\ 170\ 621$ terminal states. It means that generating the entire decision tree is nearly impossible for this game, even if it seems simple.

5.3 Sokoban

The third game that was developed is a puzzle game, where players, that can not walk through walls (brown tiles), must push boxes into holes (black tiles) to fill them (grey tiles), in order to get to the final tile (green tile). The game can be played by a single player, but multiple players can cooperate to achieve the same goal: reaching the final tile. Typically, the latter is protected by boxes that can be pushed into holes. By pushing the boxes in front of him, the player can eventually reach the final tile by walking on filled holes or block the access to the final tile, leading to a defeat. The difficulty of the game comes from the board itself, which can be designed in a text file. As an example, the following text gives the board shown in figure 23:

```
wwwwwwwwwwwwwwwwwwww
p                  w
wwwwwwwwwwwwwwwww b ww
wwwwwwwwwwwwwwwww   ww
wwwwwwwwwwwwwwwww   ww
wwwwwwwwwwwwwwwww h ww
wwwwwwwwwwwwwwwww e ww
wwwwwwwwwwwwwwwww h ww
wwwwwwwwwwwwwwwww   ww
wwwwwwwwwwwwwwwww b ww
p                   w
wwwwwwwwwwwwwwwwwww w
```

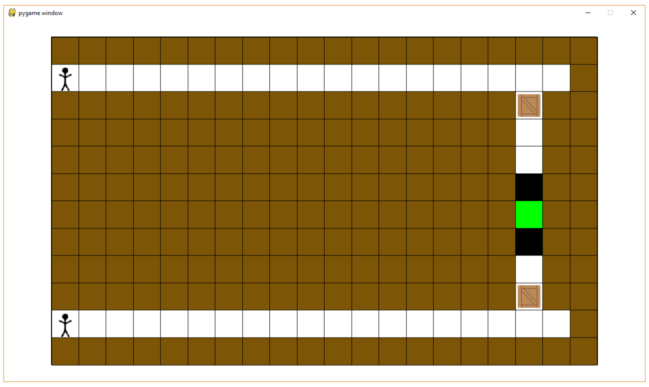


Figure 23: Example of a 2-player Sokoban game

As said earlier, the framework relies on a team-based definition of games. Which means that a game is meant to end when there is only a single team remaining. While Sokoban is a game where multiple players can collaborate, those players are in the same team. To solve this, we used a simple trick, which is to create another team in which the boxes will be, and another one in which we will add a special invisible unit. This unit will be virtually located on each final tile, and, when a real player comes on a final tile, the special unit dies, if and only if all the real players are located with him on final tiles. Otherwise, this unit stays alive, keeping the game running.

The Sokoban game is not solved in terms of AI, as the game rules does not define a fixed board configuration. There exist multiple algorithms and heuristics that solve numerous game configuration [69]. This game adds a new difficulty to AI development as these AIs must be able to adapt themselves to any game configuration.

6 Implementing a new game

Our engine has been designed to ease the development of new grid-based games. To achieve this, we developed a helper script that focuses on creating the necessary files to implement a new game. It creates a file structure and commented code that a developer can use to start implementing a new game. In these generated files, TODOs and FIXMEs are left in the code to indicate to the developer where to implement the rules of the game. Indeed, the generated files are implementing the most generic board game possible, although the user can select the type of game he wants to create in the options of the script. He can choose between a war game, a duel game, and a puzzle game. These types were defined in section 2.4 and have an impact on how the game rules must be implemented in the engine.

Generated files

The files generated by the script are structured the same way as the examples introduced in 5. The generated files and classes are named following the name of the game that was given by the user. Typically the script creates the following directories and files, placed inside a directory named after the game (in practice, the term *game*, in the name of the files below, is replaced by the name of the game):

- controllers/
  - __init__.py – Defines a python package
  - player.py – Defines the basic player classes for the game
  - wrapper.py – Defines the basic controller wrapper classes for the game
- rules/
  - __init__.py – Defines a python package
  - *game*api.py – Defines the methods that AIs will use to choose a new move and get information on the current state of the game. The user must also define how an action can be generated when a controller selects a new move.
  - *game*core.py – Defines methods that affect the execution of the game, such as the collision handler method.
- units/
  - __init__.py – Defines a python package
  - This directory contains the units that the user wants to create for its game (their number and their names depend on the options used when the script was executed).
- res/
  - AIs/ – Folder that will contain the AIs developed for the game
- builder.py – Defines methods to create new instances of the game.
- main.py – Defines methods to launch a basic GUI and to launch the game.

Implement rules

There are four main things to do to implement the rules of the new game when the files are generated. First, one must define how to create game actions in the API of the game. Path classes must be used to define the actions performed when an entity is moving from tile to tile. Then, if needed, the collision handling method must be adapted to the game in the core of the game. Thirdly, the units classes must be completed with an image file if needed, and optionally with custom attributes that can be used in the game. Finally, the methods that create the game must be adapted if needed to customize the new game instances. The only remaining files to create to launch a new game are the AIs that will play the game.

7 Game engine quality

At this point we will make sure that our engine fulfils the objectives that we set in section 1.1 of part I. We will first check if the engine is as extensive as we wanted it to be, and then if it is as robust as we wanted it to be.

7.1 Extensiveness

Below is a reminder of the qualities we listed for our engine to be extensive, so that it can be used to implement a large number of grid-based computer games.

1. Handle turn-based and real-time games.
2. Handle different numbers of players.
3. Handle team-based games.
4. Handle competitive and cooperative games.
5. Handle different game rules.

7.1.1 Real-time and turn-based games

To allow the implementation of both real-time and turn-based games, we chose to vary the moments at which the events are sent from the engine to the controllers and inversely. Indeed, the controllers choose a new move when they receive an event from the game. This is why we installed a "lap" system. A lap is defined as a certain amount of time during which the game engine waits for the players to choose their next move. Additionally, the state of the game is updated at different times.

Real-time games In real-time games, all players are sent events simultaneously when the game is updated, and their new actions are taken into account at the same time. Indeed, we consider that all controllers have to choose their next action during the same lap. If a controller did not choose any action during the lap, a random action is selected for its unit, so that the engine does not wait indefinitely if a controller crashed. Figure 24 illustrates the process of sending and waiting events in a real-time game. Lazerbike is an example of real-time game. The duration of a lap is defined by the speed of the bikes. A lap ends when the bikes reach the next tile, and a new one starts right after the game state was updated.

Turn-based games In turn-based games, an event is sent only to the next player that must play when the game state is updated, and the game engine only listens to the next player's future actions, so that any other player's action is put in wait until its turn comes. It means that every player has its own lap in which it must select a new action. Once again, if a controller runs out of time, a random action is selected without its knowledge. Figure 25 illustrates the process of sending and waiting events in a turn-based game. The Connect 4 game is an example of a turn-based game. The duration of a lap is defined arbitrarily beforehand and establishes a time-constraint for the AIs.

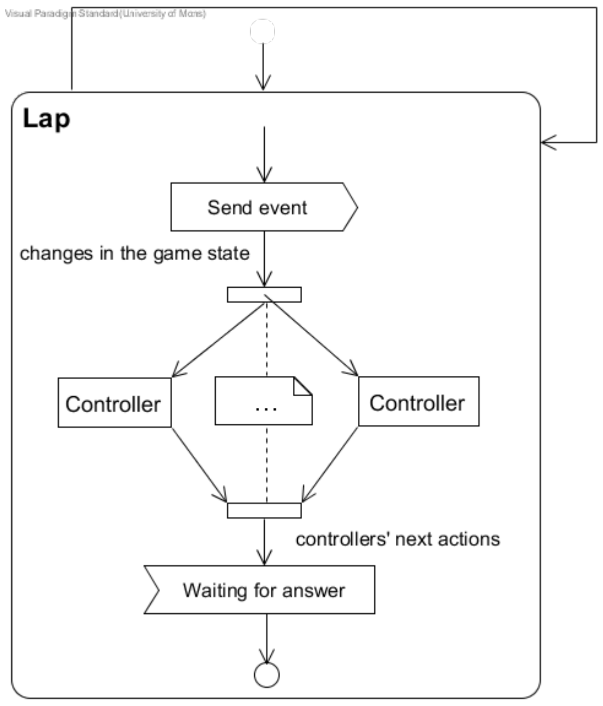

Visual Paradigm Standard(University of Mons)
Lap
Send event
changes in the game state
Controller
...
Controller
controllers' next actions
Waiting for answer

Figure 24: Illustration of the sending and waiting events process in a real-time game

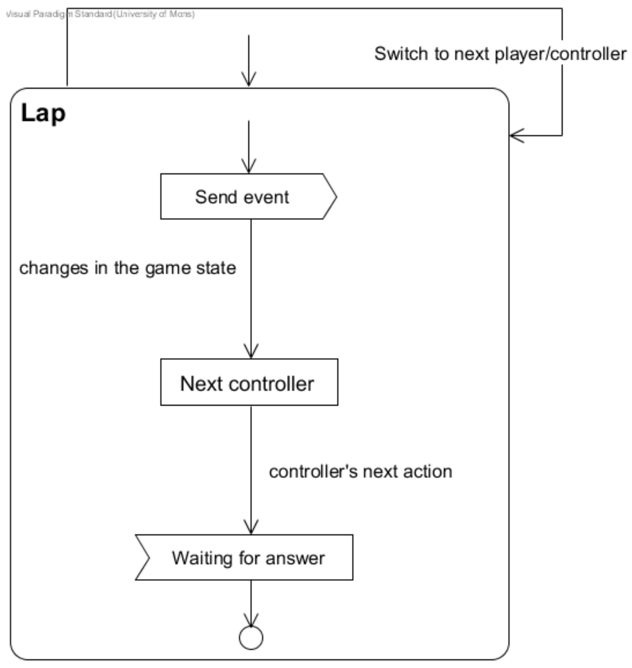

Visual Paradigm Standard(University of Mons)
Switch to next player/controller
Lap
Send event
changes in the game state
Next controller
controller's next action
Waiting for answer

Figure 25: Illustration of the sending and waiting events process in a turn-based game

7.1.2 Number of players

Varying the number of players between two games can be done quite naturally. Indeed, the units and their controller are added to the game using a method when instantiating the game. When building the GUI for a game, two parameters are available to set the minimum and the maximum number of players.

Thanks to this, any number of players can be added to the game, even if the players don't have any avatar on the grid. Indeed, we explained earlier that units could be declared as active or non-active and controlled or not-controlled. For example, in the Connect 4 game players are not visible on the board, but they still must be added to the game, and killed if the player lost.

7.1.3 Teams

The engine can run both team-based non-team-based games. Indeed, we distinguished three types of games that the engine can implement in section 2.4: war games, duel games and puzzle games. We explained earlier that the framework is team-based. Hence, for duel games, we must create as much teams as there are players, and assign each player to a different team. The teams in a war game are well-defined in general and can be directly put into the game using the GUI's team selector. For a puzzle game, there must be a fake unit belonging to a fake team, which must be killed if the player(s) won. Indeed, the engine's method that checks if the game is finished counts the number of teams that include at least one alive player. Hence, a game that does not have at least two teams would be considered as finished as soon as it would start. This choice was made to abstract as much as possible the game's execution and life cycle.

7.1.4 Cooperative and competitive games

To make the framework compliant with both competitive and cooperative games, we had to address two problems: introducing teams in the game engine and make the communication between the bot controllers easy to manipulate. This is made possible by the inter-process pipes shared by team mates that were introduced in section 3.2. This way, even bot controllers can cooperate to reach a common goal.

7.1.5 Game rules

It was difficult to adapt the framework to any grid-based game rules. The easiest way to do this was to give to the user the "new game creation tool" introduced in section 6. Indeed, the rules of the game have to be defined in the action generation, in the unit definition and optionally by overriding existing methods in the engine core for the pickiest rules. With the three game examples that we implemented and introduced in section 5, in which the code is minimalist (under 1000 lines of code per game, and with a mean of 663 lines of code per game), we provided a proof-of-concept that the engine can implement multiple types of game, in which the rules differ consequently. Indeed, between Lazerbike, a real-time game in which units are using continuous moves and are leaving laser traces behind them, Connect 4, a turn-based game in which the players are not represented on the grid in itself, and Sokoban, which is a real-time cooperative game, the only point in common is that they all use a grid of tiles in their rules.

7.2 Robustness

We want our engine to be robust and efficient, and we listed some points to consider for our framework. Below is a reminder of those points:

1. Provide a secured way for the AIs to interact with the game and prevent cheating.
2. Make the game independent of the AIs' performance (i.e. make sure a slow AI does not slow down the game).
3. Provide a complete API for the AIs to communicate with.

7.2.1 Preventing cheating

To prevent cheating we used different processes to run the game core and the units' controllers. Doing this, we let the operating system handle the data protection between processes, and if a controller tries to send an invalid action to the game engine, the move generation method will reject it, leaving the game state untouched.

7.2.2 Assure performances

The multi-process technology used to prevent cheating also makes the engine more efficient in performance. Indeed, by default, `Python` runs on a single process. Running multiple processes allows to take advantage of today's multi-core architectures. Additionally, it can prevent the engine to be disturbed by an over-thinking AI that would take all the computing power otherwise.

7.2.3 Providing an API

Each controller has access to an API as well as to a copy of the raw game core encapsulated inside the API. The API class defines methods that are common to every grid-based games, but custom APIs can easily be developed when implementing a new game, as explained in section 6.

7.2.4 Stress test

To conclude this list, we provide the results of a stress test, in which four different AIs tried to send numerous invalid or unauthorized actions to the game core. To do so, we developed an AI that sends `X` actions per seconds in the `lazerbike` game. To evaluate the engine, we performed runs of 10 seconds on the game and counted the number of frames that were displayed during this laps of time. We also compared these results with the result of a game in which four players did nothing (passive bots). As the loop ran at 30 frames per second, the results we were expecting were about 300 frames. The test on the passive game returned 299 frames in 10 seconds.

Value of X	# Frames (spam)
100	298
500	299
10000	298

As we can see on the frames count above, the bot can send as many message to the game core as it wants, the game is not affected, or at most lose 1 frame.

The only risk the engine could face is that one developed AI causes the computer to slow down by overloading the computer with multi-threaded computations. As the entire computer will slow down, the game engine would too.

8 Code quality

The framework was written with a focus on code quality and robustness. `Codacy`[8] was used to control the quality of each update of the framework. `Codacy` is an online tool that statically checks the quality of the code at each commit, following multiple standards of the programming language such as `PEP8` [70]. `TravisCI`[9] was used to build and test the framework in a neutral environment. Indeed, `TravisCI` is an online tool that executes the unit tests of a software at each commit and sends an alert when a unit test fails.
The type hinting of `Python 3`[10] was used throughout the code in order to keep the code as clear and understandable as possible. Indeed, `Python` is a dynamically typed programming language, and using an API in which the user has no clue on the type of the parameters is really difficult. Furthermore, the comment coverage on the methods of the framework exceeds 90%, detailing the parameters and the return value of the methods.

Additionally, we tried to write as much *Pythonic* code as possible. A code is said "Pythonic" when it respects the guidelines and the coding spirit of `Python`. Some examples[11] of guidelines are the following:

- Use "_" and "__" to inform on the visibility of a method or property: a single underscore to indicate a protected member and two for a private one.
- Keep libraries as lightweight as possible.
- Prefer *foreach* loops rather than while loop when it is possible.
- When it does not affect the performance, prefer returning a value rather than modifying parameters in-place.
- ...

To summarize all of this, Tim Peters, one of the most known Python developers, established the "Zen of Python". It is a list of advices on writing Pythonic code:

[8] https://www.codacy.com
[9] https://travic-ci.org
[10] https://docs.python.org/3/library/typing.html
[11] These come from a useful article: https://blog.startifact.com/posts/older/what-is-pythonic.html

The Zen of Python, by Tim Peters
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one– and preferably only one –obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
*Although never is often better than *right* now.*
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea – let's do more of those!

9 Implemented algorithms

9.1 Path finder

It is obviously helpful that a game API can provide a path finding method to find the optimal path between a given point and another. To do so, two simple implementations of the Dijkstra's algorithm [71] have been developed. These implementations allow the users to configure the path finding by giving a function that takes a tile object as input and returns true if the path can use that tile, and false otherwise. This method is used to filter the tiles that can be walked on from those that are forbidden. For example, a certain type of tile could be non-deadly by itself but could induce a penalty if it is walked on. If possible, we will first try to find a path that does not include such tile. If it is not possible, the user can try to use less severe criteria to reach the wanted tile.

The first implementation finds the optimal path from the given tile to every other tile located within a given maximum distance. The second implementation restrains the searches and focuses on finding the optimal path between two given tiles. Obviously, the second method is faster than the first.

9.2 Simultaneous Alpha-Beta

One of the best known decision-taking algorithms is the Min/Max algorithm, with its Alpha/Beta improvement [26]. The essence of the algorithm is to simulate a large amount of moves for the players, given an evaluation function, to select which move is the best for a certain player, given the current game state. The classical implementations commonly consider only two players that play one after the other. As our framework runs real-time multi-player games, our implementation must take into account *(a)* that there can be more than two players and *(b)* that the players can move simultaneously.

(a) More than two players

Two players in the classical implementation are used to distinguish a *MAX* step, in which the player that looks for a move (which we will call the *main player*) will try to maximize the outcome of its possible moves, and a *MIN* step, in which the other player will attempt to minimize the score of the main player with the outcome of its possible move. The *MIN* step is useful so that the *MAX* step does not take into account bad moves of the opponent as a good move for him, but instead, the algorithm supposes that the opposite player plays the best way he can to minimize the score of the main player. For our implementation, we can retain that the *MAX* step is used to choose between the actions of the main player, and that the *MIN* step is used to choose between the actions of the other player.

As the number of players can be bigger than two, we must generalize the *MAX* and *MIN* steps so that we can use it. Keeping the logic behind the two steps, we will use the *MAX* step to choose the action of the main player, and the *MIN* step to choose the action of every other player. But does it remain logical that the outcome of the actions of every other players looks to be minimized ?
We can distinguish two types of other player: *(1)* a player that is in the team of the main player, and *(2)* a player that is in any other team. It seems obvious that a player of type *(2)* will eventually seek to minimize the score of the main player, or at least of its team, while a player of type *(1)* cannot be controlled by the main player, and will be assumed to be a bad player (i.e. for which the action will minimize the score of the main player, or of its team), so that the actions of the main player do not depend on the actions of that other player.

(b) Simultaneous moves

As the moves are made simultaneously[12], we must simulate the moves of all players all at once. This is a problem if we keep the classical implementation, which simulates the move at each step, whether it is a *MIN* step of a *MAX* step. To solve this problem, we will simulate the moves only in the *MIN* step, in which the moves of the other players are evaluated. This way, alpha can still be updated during *MAX* steps, and its pruning can be done during *MIN* steps, while beta can still be updated during *MIN* steps, and its pruning can be done during *MAX* steps (the alpha and beta pruning are detailed in [26]), similarly to the classical implementation.

In figure 26, an example of tree coming from our simultaneous alpha beta is shown. This tree simulates two moves for each players, the blue horizontal lines representing the moment where simulations can be done. The letters next to the arrows of the tree represent

[12]We can note that a turn-based game can also be simulated by setting a parameter

the actions taken by the players, in order. The arrows coming from a *MAX* vertex are actions selected for the main players, while arrows coming from a *MIN* vertex are actions coming from other players (e.g. `AB` means the first other player does action `A`, and the second does action `B`).

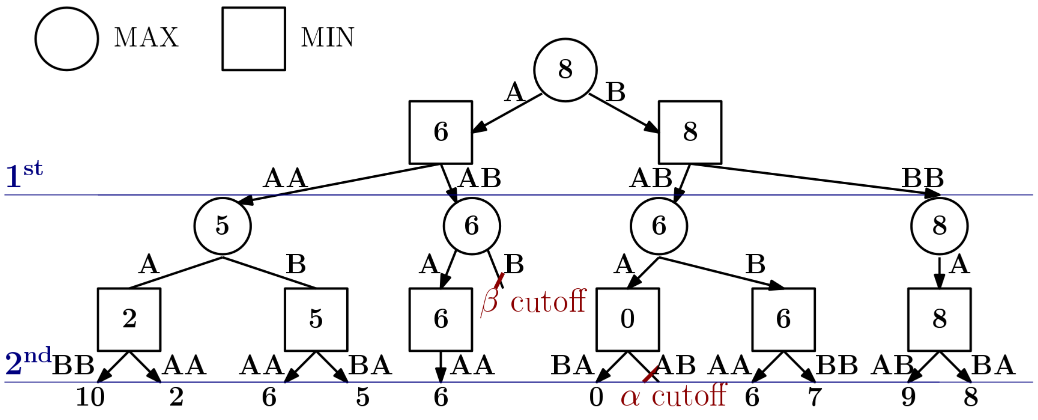


Figure 26: Example of a 3-players simultaneous alpha beta

10 Data generation principles

Multiple data generation techniques have been developed in order to gather training data to train AIs based on machine learning. We developed four different techniques to do so. They are described below. All of these work following a common procedure:

1. Generate and simulate moves. The method used depends on the technique.
2. Collect data. The data to collect and the moment to collect it depends on the type of data, cf. the following paragraphs on data prediction and classification.
3. Save the data into files. This is done when a certain number of data, set using a parameter, has been collected.

These techniques all generate three types of data: raw data, sequences of actions and game board information. In fact, three types of machine learning techniques were considered: data prediction, data classification and deep learning. In the first type, we need sequences from which we will predict the next data. In the second type, we need information on the current state and information to compute probabilities to belong to a certain class and in the last, we need raw data from which we can extract features. We will detail and explain in the following paragraphs why those techniques require different learning data.

10.1 Data prediction

Data prediction in video games needs actions sequences that lead to a final state of the game from which it can learn. As it needs sequences to predict the next action, we can use any part of the generated sequence to make the model learn from it. For example, let us take the following sequence and consider it as an ordered list of dependent actions in a video game:

$$1,\ 3,\ 2,\ 8$$

From this little sequence, a predicting model can learn from three situations:

$$(1) \rightarrow 3$$

$$(1,3) \rightarrow 2$$

$$(1,3,2) \rightarrow 8$$

Yet, for many games, we need multiple vectors to predict the next action. Indeed, there are often multiple players to play the game, and the next action of a player does not depend on that player's actions only, but on the actions of all the players. Let us take the following sequence as example, knowing that we want to predict the next move of the first player and that there are two players that are playing the game:

$$\begin{array}{rcccc} Player\ 1: & 1, & 3, & 2, & 8 \\ Player\ 2: & 6, & 1, & 4, & 1 \end{array}$$

The prediction steps will then look like this:

$$\begin{pmatrix}1\\6\end{pmatrix} \rightarrow 3$$

$$\begin{pmatrix}1\\6\end{pmatrix}, \begin{pmatrix}3\\1\end{pmatrix} \rightarrow 2$$

$$\begin{pmatrix}1\\6\end{pmatrix}, \begin{pmatrix}3\\1\end{pmatrix}, \begin{pmatrix}2\\4\end{pmatrix} \rightarrow 8$$

To capture such data, we will generate moves using the technique's specific generation method until reaching a final game state. Then, it remains to save the actions history of all the players. As we want our player to win the game, we will only use sequences that led our player to win the game for the learning process. Indeed, as we want to predict the next action to perform, it would be pointless to predict a move that will lead to a defeat.

As this method uses sequences of actions, this technique is highly dependent on the initial state of the game. Indeed, executing a sequence of actions on another initial state than the one used to collect the sequence is likely to fail. We will take the Lazerbike game, explained in section 5.1, as an example. Let us take the following sequence of actions for the red player:

F, R, R, F, F, L
where F = go forward, R = turn right, L = turn left

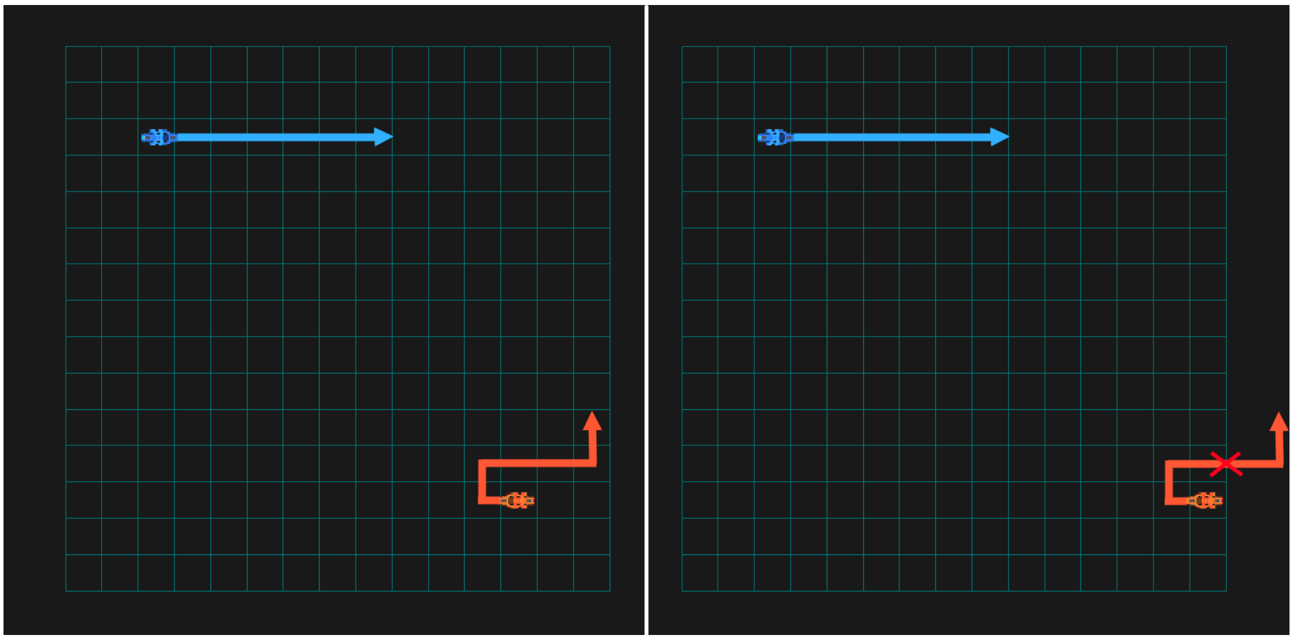


Figure 27: Execution of a sequence of actions on two different initial states

We can see on figure 27 that if the red player changes its initial position, as in the second state, the sequence of actions makes him crash into a wall, which was not the case on the first initial state. This is a drawback, and it can make this technique totally useless for some games. For example, in the Sokoban game, detailed in section 5.3, changing the board, and, hence, its initial state is part of the game. Indeed, the boxes, walls and players' positions are changing between two levels of difficulty. Using this technique, we would be able to train a model to solve one level, but as it is, we could not train a model that can solve different level configurations.

10.2 Data classification

This second type of machine learning techniques focuses on understanding how to separate the data into multiple groups or classes. Typically, to perform the classification we need high level features extracted from the raw data, and this is what we called "game board information". We divided this information into two kinds: a priori data and a posteriori data. A priori data are collected as soon as the game state is reached and must summarize it, while a posteriori data needs information about a final state that was reached from the first state as it will be used to evaluate the game state for which we collected a priori data. In classification, a priori data, as we defined it, is used as data vectors, while a posteriori data is used as target vectors. We will have one target vector per game state and per action feasible from that state. To summarize, the learning process must find the relations between the data vectors and their label, which is selected using their target vectors, while the classification process must choose the right class in which the given data vector must be placed in.

We explained that to collect a posteriori data, we must reach a final state to evaluate the state for which the data was collected. The question then arises of how to choose the actions that will lead to the final state ? This choice is made differently depending on the technique that is used and will be discussed below. We did not talk yet about classes. Indeed, the goal is to classify data vectors. As the context is a video game, we will need to select the right action at a given moment, for a given game state, the class in which to put a data vector will be the action that suits the best to the data vector, and hence to the game state. This is why the target vectors linked with each game state will be evaluated and transformed into a score, or a probability to assign to each class, and hence, to each action.

As an example, we can take the Connect 4 game, which is detailed in section 5.2. As a priori data, to summarize the game board, we will encode the entire board. The Connect 4 board is made of 42 tiles, and for each tile, we will have a feature in the data vector encoded as the following: **0** if the tile is empty, **1** if the tile is filled with a disc belonging to player 1 and **2** if it is filled with a disc belonging to player 2. As a posteriori data, we can for example take two metrics: 1) 1000 if the player has won the game, 0 otherwise and 2) The number of winning possibilities that were available just before the game ended.

It is not always possible to take the entire board as training data. Indeed, in the Lazerbike game for example, detailed in 5.1, the board is made of 225 tiles, which would drastically make rise the dimensionality of a data vector. Hence, for such games, the need for data summarization arises, and more complex data must be collected in order to reduce the number of features per data vector.

10.3 Deep learning

As we saw in section 3.2.3 of part II, deep neural networks such as convolutional neural networks have the ability to automatically extract features from raw data. This is why our data collection techniques extract raw data as well. What we call raw data is information on a game state that is enough to reconstruct that game state. Basically, it represents a saved game. From this, any metric can be computed and features can be extracted from a deep learning model.

A game state can be saved using an encoding method that is implemented when creating a new game. That method returns a code for a given tile. Each value for that code means something. For example, -1 can represent a deadly tile; 1 can represent the tile on which player 1 is currently located; 2 can represents the tile on which the player 2 is currently located, and so on...

11 Data generation algorithms

In the following section, we will discuss the algorithms that we developed inside the game engine that focus on generating data.

11.1 Thorough data generation

As our goal was to give a simple way to develop simple games, we can assume that generating the whole decision tree of such a game is possible. This is why we developed this technique, whose goal is to try all the possible moves and generate the whole decision tree. Obviously, this is possible only on very simple games.

Technical overview

The technique is based on the developed simultaneous alpha beta algorithm detailed in 9.2 in which the cut-off abilities have been disabled so that no situation is forgotten during the exploration; it is hence a recursive deepening depth-first exploration of the decision tree. This is the most complete technique because it generates all the possible states from a given state. The default value is the initial state of the game, so that the technique goes through every state.

Action generation

There is no real action generation step, as we consider every possible move at each step. Indeed, the states will be enumerated recursively by simulating all the possible moves, one after another.

A posteriori state selection

To evaluate a game state at a certain moment, we said earlier that we needed a final game state to be reached from the first one. As there are many final states that can be reached from a given state, we will use the alpha beta properties (see section 9.2) and an evaluation function. This will for example avoid to select a final game state in which the opponent player(s) committed suicide. Hence, if we want a good evaluation of what a certain game state can give, we must set a good evaluation function, so that all the players play correctly thorough the simulations to obtain the selected final states.

11.2 Randomized data generation

As we said in the last section, generating the whole decision tree for a game is impossible for most games. We could just say that we were going to let the complete technique run until we are out of time, but the problem is that the selected actions would not be diverse enough to be used in a machine learning technique. Indeed, as the game states are explored using a depth-first exploration, the first selected actions would be present in every data. As an example, let us assume that for a given game, each player get 3 possible actions (0, 1, 2) at each step, and that the number of steps to reach a final state is 20. The number of possible game states is:

$$3^{20} = 3\ 486\ 784\ 401$$

As the complete technique will explore every possible action, it will start with the first action: 0. The five first selected actions will hence be:

$$[0, 0, 0, 0, 0]$$

The problem here is that for the algorithm to change the 5^{th} action, it must reach $3^{15} = 14\ 348\ 907$ final states, which is quite high. To generalize, once i action have been reached, the technique must reach 3^{20-i} final states.

Technical overview

This difficulty to obtain diverse data is why we decided to develop this second technique that will use randomly generated game states to collect data. Indeed, the idea is to perform a random number of random actions on an initial game state until reaching a non-terminal game state from which we will simulate a certain number of actions and collect data on the resulting game states. To do so, two variables are left as parameters: `X` and `Y`. `X` will be the number of random states to generate during the data collection, and `Y` will be the number of final game states we want to reach from each random state. These parameters are illustrated on figure 28

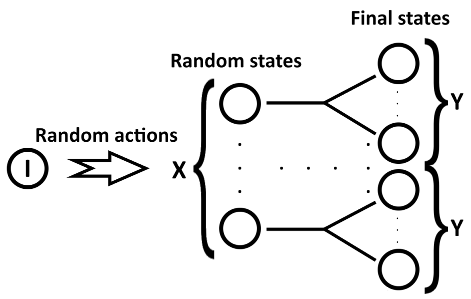


Figure 28: Illustration of the randomized data generation

Actions generation

As explained in the last paragraph, there are two phases of action generation, the first being the random state generation phase, and the second being the terminal states generation phase. During the first, a recursive process will randomly select and simulate a random number of actions from an initial game state. During the first phase, if those actions lead to a terminal state the recursive calls will get back to the last non-terminal state and simulate another move. Indeed, we do not want to reach a terminal state yet as we want to reach multiple (`Y` if possible) final states from the random state we are generating. Once we reached a stable non-terminal random game state, we will use the thorough technique, in which the actions are selected randomly at each step, from this state to reach `Y` final states if it is possible. We repeat the entire process until `X` random states have been generated.

A posteriori state selection

As for the thorough technique, the terminal states from which the a posteriori data will be collected will be selected using the alpha beta algorithm and a given evaluation function left as parameter.

11.3 Battle-based data generation

The previous techniques used randomly selected actions to reach final states, without any game-logic. It also means that any machine learning technique that would learn from such data will learn from those random movements performed by the opponent(s). Hence, it will be difficult for the machine learning models to understand and counter advanced strategies of opponents that could not be generated randomly.

Technical overview

This is why we developed this technique of data collection that will use an AI developed beforehand for the game. But there is a risk of over-fitting when learning from a single AI. Indeed, the model could adapt itself to the opponent's strategy, but it could fail to win against a simple AI as it would not have learnt to win against such AI. A solution would be to make it learn from data coming from multiple AIs of multiple levels.

As for the randomized technique, we must set two parameters, `X` and `Y`, which have a similar goal as in the latter technique. Indeed, `X` will determine the number of games to simulate, and `Y` will be the number of terminal states to reach for each initial state. Optionally, a parameter `Z` can be set and will determine how much winning terminal states the player must reach before starting a new game. The number of terminal states to reach for each state will hence be either `Y` if we reached at least `Z` winning states, or more until we reach this number of winning states. This parameter is optional because it may take a very long time to reach these winning states, it can even be impossible. In that case, we will recursively try every possible action for the learning player, and we saw earlier that it may take a large amount of recursions. The data collection ends when enough (`X`) games have been simulated.

Actions generation

The actions of the learning player are still randomly selected. At each step, the player will try multiple actions recursively, while the opponents will choose their actions using their selected AIs. This technique can be inefficient if the opponent is too good. Indeed, it may take a long time to find winning actions sequences considering the opponent select intelligent moves.

A posteriori state selection

The final states are computed following the best final state for the learning player. Indeed, the previous techniques used alpha beta to simulate the moves of the opponent so that the final state was not reached because of suicidal moves from the opponents. In this technique, the actions chosen by the opponents depend directly on the selected AIs, and we do not need anymore to simulate their moves.

11.4 Imitation data generation

The difficulty for the last technique to reach winning final states made us develop another technique, which will be a bit more intelligent in the moves that the learning players select. For this one, we will need an AI for each unit in the game, even the learning player. Indeed, the goal is to gather data that will be used to mimic the strategy of one of the selected AIs.

11.4.1 Technical overview

The functioning of this technique is similar to the battle-based one. Indeed, we will simulate a certain number (`X`) of games using each player's AI until we reach enough (`Y`) final states, or, optionally, enough (`Z`) winning states for the learning player.

11.4.2 Action generation

A new parameter can be set for this technique: `W` $\in [0, 1]$ which will be the probability for an action to be randomly selected for one of the player. Indeed, if one of the selected AIs is based on machine learning, there is a risk of over-fitting and we could not be able to reinforce it. Furthermore, if one of the AI that is weaker than its opponents', it may never win, resulting in an inefficient learning session. This is why we allow to test multiple moves at a certain point. At each step, we will let one of the AI choose an action, and with a probability `W` we will add another move randomly. If multiple actions are selected this way, they are simulated recursively one after the other.

One way to avoid generating random moves is to try multiple combinations of AIs to collect data, so that models that will learn from it can take the best of each AI, and adapt to multiple types of opponents.

11.4.3 A posteriori state selection

The final states are selected the same way than the battle-based technique.

12 Known limitations

At the moment, the lap system has not been totally implemented and would need some testing. Currently, the syncing of the units' actions is handled by starting simultaneously actions that take the same amount of time to be completed.

A second limitation impacts the performances of the AlphaBeta-based AIs. Indeed, to copy the current state of the game takes approximately 1ms. When the decision tree requires millions of game simulations, the time to wait for an answer is very long. To overcome this problem, the simulation system should stop focusing on copying the game state and should use temporary actions to apply to the game board. This way, actions could be performed, and then cancelled to fall back on the initial game state.

Part IV
AIs generation

We generate and test AIs on two different games that serve as case studies. The first game is Connect 4 that was introduced in section 5.2 of part III and the second game is Lazerbike, presented in section 5.1 of part III. In both cases, we start by explaining how we generated learning data for the game (**P0**). Then, we introduce neural networks we developed for the case study (**P1**), then we show that neural networks can get better results than classical machine learning techniques (**P2**). This is achieved by making different AIs by learning from the same data set, and then by making them fight against each other. Finally, we make the different neural-network-based AIs fight against different AIs in order to evaluate their efficiency (**P3**).

1 Learning techniques

In this section we will introduce the different techniques that we considered to learn to play both games. The biggest difficulty when training models to play games is that choosing an action at a given moment is subjective until the final state is reached. Indeed, there are multiple ways to win the game, and multiple actions could be good to take for a given game state. Hence, to punish a model because it chose a certain action is not always justified. We will call this problem the *subjectivity problem*.

1.1 Sequence learning

In sequence learning, we will apply what was explained in section 10.1 of part III. Therefore, we will use action sequences, and for each, we will make our models learn how to predict the action that follows that sequence. Actions sequences are interesting data because we do not need to extract features from it to make our models learn. We can use the unmodified data coming from played games as training data. One difficulty about using sequences as learning data is that it is not easy to visualize.

1.1.1 Classical machine learning

There are three main sequence learning techniques that do not rely on neural networks [72]: sliding windows, conditional random fields (CRFs) and hidden Markov models (HMMs). However, sliding windows only take advantage of very short term memory [72]. As a result, they would be very bad at our games, as every action depends on all the actions that have been played before. Hence, we will focus on CRFs and HMMs.

HMMs are models that are widely used in time-series prediction, such as speech recognition [73]. These models are made of states, their functioning is detailed in [73]. It can be difficult to provide a HMM complex enough to understand long-term relationships between data. However, their training process if very fast and they usually do not take much space to store (from 1 to 100 kilo bytes).

CRFs have been built to overcome the issues of HMMs, and they are known to get better results than HMMs on most problems [72]. Their functioning is detailed in [74]. Their training process takes approximately the same time as HMMs', and they take the same storage size as HMMs.

1.1.2 Neural networks learning

Learning how to play games using action sequences implies that the models must use long-term as frequently as short-term memory to remember which actions were made in order to select the next action. This is exactly what LSTMs were made for. However, there are many ways to make a LSTM-based neural network. Indeed, in most neural network tools, one has to build a neural network by defining its layers, and as a result, one could mix multiple types of layers such as LSTM layers, convolutional layers, dense layers[13], ...

In any case, LSTM layers can learn from an entire sequence and return another sequence. This is usually called sequence to sequence learning [3]. This is useful to make the model learn each step of the sequence without having to decompose it. Indeed, as we saw in section 10.1 of part III, from one sequence of size n, the model can learn $n - 1$ consequent actions. In practice, these can be learnt all at once by shifting the original sequence from one. For example, the sequence

$$[4, 2, 1, 4, 5, 6]$$

can be learnt using sequence to sequence learning as following:

$$[4, 2, 1, 4, 5] \rightarrow [2, 1, 4, 5, 6]$$

Indeed, the sequences must have equal lengths, and the first actions of the sequence cannot be learnt from nothing.

We tried multiple architectures for each case study, and the architectures proposed in [3] and [75] obtained terrible results in both cases; these models attempted impossible moves and/or committed suicide way too often to deserve to be mentioned in the tests below. We can explain this by the fact that the input sequence is first summarized into a single cell "C", which is supposed to give the context, as illustrated on figure 29. It seems difficult to summarize all the complex mechanics of a game with all the previous actions into a single context cell. As a result, we built models that did not summarize the context at any point, leaving all information available until the output layer. Our model architecture will hence be made of a certain number of LSTM layers, which we will vary, with sometimes a dense layer on top of it in order to gather the information of all the LSTM layers before the output layer. The general form the chosen architecture can be seen on figure 30, with $X > 0$ and $Y \in [0, 1]$. We will call this architecture the recurrent architecture in the following.

[13]Layer made of classical neural networks cells, each one entirely (densely) connected to the previous layer.

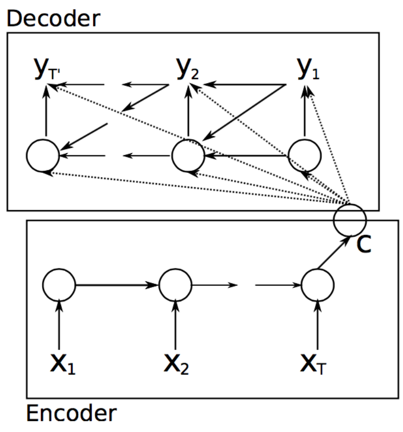


Figure 29: Illustration of a "peeky" sequence to sequence model.
Source: [75]

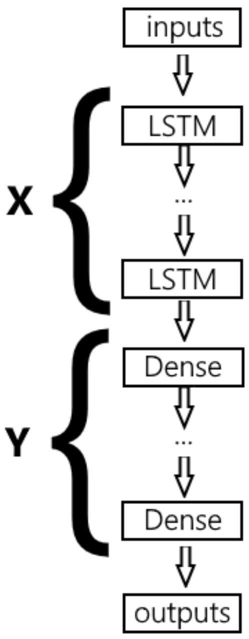


Figure 30: Generalized illustration of our architectures.

In addition to parameters X and Y, there are other parameters to set in order to make the recurrent model learn:

- Number of cells per layer.
- Loss metric (see section 3.3.4 of part II).
- Activation function.
- Dropout probability.

The dropout probability is a parameter specifically interesting for LSTMs. It represents the probability for a LSTM cell to drop inputs, outputs or connections with the previous or the next layers [76]. Having this dropout probability seems to avoid over-fitting in the model [77] and seems to lead to better results for some applications [78].

We can note that LSTMs need all sequences to have the same size. Indeed, the first layer of the network has a fixed size, and so must the sequences. As two different rounds may end with a different number of played actions, we had to pad the sequences with a neutral value, as recommended in the Keras documentation[14]. A neutral value is a value that is not already used in the dataset.

1.2 Board analysis

One way to decide which action one AI must select is to analyse the board in itself. But it implies that one must extract useful features from the board in order to make a decision. Once we gathered the features that represents the current state of the board it remains to classify those features amongst the possible actions. The result of the classification will hence be the action that best suits the current situation. The data to use for such learning method is the one detailed in section 10.2 of part III. The greatest difficulty about gathering such data being that one must define a good evaluation function and define good features and metrics to classify the current state of the board.

As said earlier, making neural networks learn takes a long time. We did not have the time or the resources to explore this method of learning for this masters thesis.

1.2.1 Classical machine learning

There exist multiple classification algorithms, and some are listed in section 2.1 of part II. Each algorithm has its advantages and setbacks. One should try multiple algorithms and compare the results with the neural network-based technique.

1.2.2 Neural networks learning

The most known neural network architecture that classifies data is MLP, introduced in section 3.2.1 of part II. As for all the neural network techniques, the main difficulty to obtain a good model is to try several sets of parameters and compare the results of each.

1.3 Feature extraction

This last method aims to pass the raw game board to a model that automatically extracts interesting features and then gives the next action that suits best for a given player. For the same reasons as the previous technique, we could not explore this method due to a lack of time and resource to train our neural networks.

[14] https://keras.io/preprocessing/sequence/#pad_sequences

1.3.1 Classical machine learning

Two steps are required to perform classical machine learning using raw data: automatic feature extraction, followed by a classification. Features can be extracted automatically using multiple techniques [79]. The classification methods are the same as those discussed earlier in section 1.2.1. One should try to use multiple combinations of automatic feature extraction techniques and classification algorithms to compare them with the neural network-based solution.

1.3.2 Neural networks learning

CNNs provide an all-in-one solution, with automatic feature extraction at the beginning of the network, and a classifier at its end. Nevertheless, the main drawbacks of these networks are firstly the difficulty to find an optimal set of parameters, and secondly the time it takes to make such networks learn. Indeed, among all the neural network techniques, this one is the slowest to train. This is explained by the fact that automatic feature extraction requires a large amount of hidden layers in the network, inducing a huge number of parameters to train (hundreds of millions to hundreds of billions).

2 Software used

2.1 Neural networks

We used `Keras` [80] to handle neural networks. Essentially, `Keras` is an API that provides methods to generate and use neural networks. The API is currently implemented by two well-known libraries: `Theano` [81] and `Tensorflow` [82] used as back-ends by `Keras`.

`Keras` offers a wide range of neural network architectures, metrics and optimizers. It allows to switch between the two back-ends at runtime, in order to make the code as portable as possible. `Tensorflow` and `Theano` are both using *tensors*, making it easier for `Keras` to handle both with the same API. Tensors are geometrical objects describing linear relations between vectors, scalar or other tensors. They are defined in details in [83]. Tensors have shown to be very efficient for heavy computations [81], [82] and in particular to handle neural networks.

The use of one back-end or the other is transparent when using `Keras`. This can be useful because some will find it easier to install `Theano`, while others could find it easier to install `Tensorflow`. Indeed, on Windows configurations, the installation of `Tensorflow` is easier than `Tensorflow`'s, but `Tensorflow` (r1.0) does not support the latest version (8.0) of `CUDA` while `Theano` supports it. `CUDA` is the NVIDIA library used for heavy GPU computations [84]. Indeed, neural networks' learning process can benefit from a speed boost when using GPU combined with a classical CPU [85].

2.2 HMM

The main Python library that provides an API to learn from HMMs is hmmlearn, which is a scikit-learn module. scikit-learn is a scipy toolkit that focuses on machine learning. Unfortunately, hmmlearn's API only provides unsupervised learning methods, and we need supervised methods because we have targets in our dataset. The documentation of hmmlearn[15] redirects to the repository of seqlearn[16] for supervised learning using HMMs. This is why we used seqlearn to train HMM models.

2.3 CRF

To train CRF models, we used pycrfsuite[17], a Python library based on CRFSuite, which is a library that provides state-of-the-art methods [86] to handle CRFs.

3 Case study – Connect 4

In this first case study, we will test multiple neural-network based AIs that play the Connect 4 game. We can note that in the further sections, some parameters may seem, and sometimes are, taken arbitrarily. This is explained by the fact that it takes a very long time (12 hours to 5 days) to train a neural network correctly. As a result, we were not able to vary all the parameters in order to find the optimal neural network.

3.1 P0 – Learning data generation

We used three multi-threaded computers during two weeks to generate data for this game. Finally we generated 11 663 347 complete Connect 4 games using the random data generation introduced in section 11.2 of part III. For each game, we stored the complete board, some information and metrics, namely the number of actions that directly lead to a win, and finally the action sequences that led to the final state of the game. These sequences are made of integers $\in -1, 0, 1, 2, 3, 4, 5, 6$. If one player gets -1 as action in the sequence, it means that the game was finished when it was that player's turn to play. We also used -1 as neutral value when we padded the sequences for the purpose of making the networks learn. An action $\in [0, 6]$ identifies the game column in which the playing disc was put.

3.2 P1 – Models

The first models we created were trained using mean squared error to measure the loss between two iterations on the data **(1)**. Then, we realised that the mean squared error was not the best choice as our targets are qualitative values rather than quantitative ones. We pointed this out in section section 3.3.4 of part II. Consequently, we trained other models, using categorical cross entropy as loss metric, and using one-hot encoding for the networks

[15] https://github.com/hmmlearn/hmmlearn
[16] https://github.com/larsmans/seqlearn
[17] https://github.com/scrapinghub/python-crfsuite

targets **(2)**. Finally, as the results were a little bit disappointing, we transformed the input, that was the columns played by both players. Indeed, these inputs were also taken as numerical values, while they are qualitative values. To fix it, we encoded the two actions into a vector of 16 binary values. The first eight values encode the action of player 1 and the final eight encode the action of player 2. For example, if the actions of both player were encoded as $[5, -1]$ in the previous models, they are now encoded as $[0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0]$ **(3)**. We used sequences of 20 actions to make the models learn, because there can be at most 21 actions.

To evaluate the quality of a model, the first thing to do is to compare it with a random action chooser. If our model is weaker than or equivalent to a random AI, we can say that it is a bad model.
During the tests, we noticed that the models often struggled to "finish their strategies". In fact, they tend to build a winning strategy, but when the time comes to end the game by putting a fourth disc in a row, they use another column. This can be explained by the sequence learning principle. Indeed, in the generated data, multiple sequences have the same beginning. For example, let us take two actions sequences leading to a win for the same player:

$$[0, 0, 1, 2, 6, 4, 6]$$

$$[0, 0, 1, 2, 6, 5, 2]$$

The model will tend to reinforce its learning for the five first numbers because it is the second time that it encounter this 5-number sequence, but it will struggle to choose between $[4, 6]$ and $[5, 2]$, leading sometimes to $[4, 2]$ or $[5, 6]$, which may not lead to a win. To solve this, we will force some defending and winning moves for our models. In practice, when it comes to the model's player to play, (a) if there is a directly winning move, take it instead of the model prediction, and (b) if there is a directly losing move the other player could do when it is its turn, play this move before the other does it. Guided by a concern of equality, we will provide (a) and (b) to the classical machine learning models, but also to the random AI.

Consequently, to win a game, a model must reach a point where there are at least two possible winning moves (fatal trap for the opponent), so that even if the opponent defends itself, we can win by playing the other action. As our random AI uses the same principle, we must not consider it only like a dummy random AI, but like an AI that does not build a strategy to win, but wins if it can and defends itself if it can.

We can note that most of the models below have learnt from approximately 10% of the whole dataset, because we did not have the time to make each one learn 100%. Yet, this was not the case for classical models, which learnt from the whole dataset, because their learning process is way faster (2 to 6 hours to learn everything) than the one for neural networks.

Mean squared error based

In table 2, we listed all the models we created for this case study that use mean squared error as their loss metric. In table 3, we listed the results of 10,000 games in which these models fought the random AI, and we plotted these results in figure 31.

ID	X – cells	Y – cells	Dropout (%)	Activ. func.	Nb param.	Batch size
1	1 – 42	0	0	`tanh`	7503	20
2	3 – 42-42-2	0	0	`tanh`	22,203	20
3	3 – 96-96-42	0	0	`tanh`	135,523	20
4	3 – 96/layer	1 – 42	0	`tanh`	190,357	20
5	3 – 96/layer	1 – 42	20	`tanh`	190,357	20
6	3 – 96/layer	1 – 42	40	`tanh`	190,357	20
7	3 – 96/layer	1 – 42	40	`tanh`	190,357	1
8	5 – 96/layer	1 – 42	20	`tanh`	338,581	20
9	5 – 96/layer	1 – 42	40	`tanh`	338,581	20
10	7 – 96/layer	1 – 42	40	`tanh`	486,805	20
11	9 – 200/layer	1 – 42	20	`tanh`	2,737,285	20

Table 2: Table containing the recurrent architectures that were trained using mean squared error as loss metric

ID	Wins	Draws	Defeats	Success rate (%)
1	4670	373	4957	46.70
2	5035	379	4586	50.35
3	5395	355	4250	53.95
4	5870	310	3820	58.70
5	6581	309	3110	65.81
6	6848	260	2892	68.48
7	7089	252	2659	70.89
8	6728	303	2969	67.28
9	6757	276	2967	67.57
10	6299	320	3381	62.99
11	6305	291	3404	63.05

Table 3: Table containing the results of the models for 10,000 games against the random AI

Notes on the listed models

1 – The first model we tried was a recurrent architecture with $X = 1$ (42 cells) and $Y = 0$. We took 42 cells for the LSTM layer because the board has 42 tiles, and we thought that it might be a good idea to have one cell computing the state of each tile. It only obtained 46.70% of success, but as a reminder, the random AI against which it fought was provided (a) and (b). Nevertheless, we will try to improve the model.

2 – For this second model, we tried to add layers in order to make it more complex. Indeed, we chose $X = 3$ (42-42-2 cells), $Y = 0$. We added the final LSTM layer of 2 cells because we wanted to try a LSTM layer that predicted the next move instead of the final

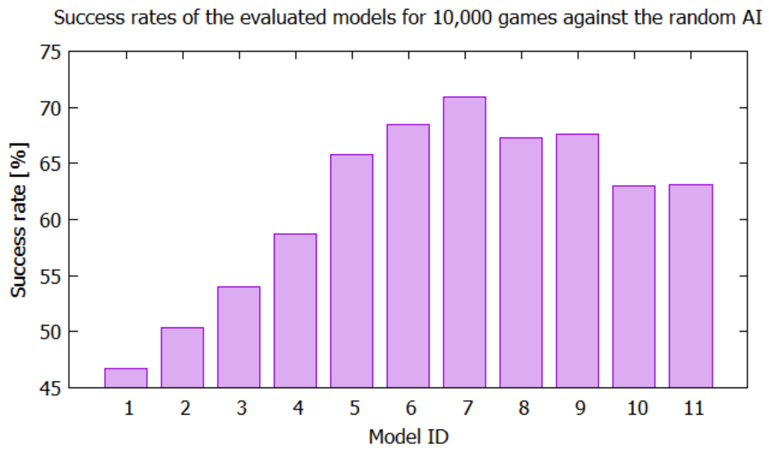


Figure 31: Bar chart of the success rate that our models obtained for 10,000 games against the random AI

output layer. Its success rate improved a little bit compared to the last one: 50.35%, but it does not represent a significant improvement. This is why we continued to make our model more complex.

3 – For the third model, we added neurons instead of layers. It obtained 53.95% of success rate. Adding complexity to the network seems to improve the results. This is why we will make the next model even more complex.

4 – We added a dense layer at the end of the network in order to process and gather the results of the LSTM layers. It obtained a success rate of 58.70% which is the best result so far. This is why we will keep this Dense layer at the end of our architecture to build the next models.

5 – For this model, we added a dropout rate, as advised in [78] and [77]. It obtained the highest success rate so far: 65.81%. As its results were good, we decided to refine this model.

6 – We chose to refine the fourth model by raising the dropout probability to 40%. Its success rate raised to 68.48%. Until now, we used a batch size of 20. Theoretically, we can improve the success rate of our models by using a batch size of 1 during training.

7 – Taking 1 as the training batch size of our model, we obtained 70.89% of success, but this choice slows down drastically (approximately 10 times slower) the training process, and we could not use a batch size this small for all our models.

8 – We tried to make the model more complex by adding new layers to check if we gained in success rate. After 10,000 games against the random AI, it obtained 67.28% of success. which is an improvement compared to model 4, which, apart from X, has the same parameters than this model.

9 – As we obtained better results by raising the dropout rate from 20 to 40 previously, we decided to raise it to 40 for the last model too. It obtained 67.57% of success, which is a very small improvement compared to the last model.

10 – We tried to raise the complexity of the model again by adding two layers to the previous network. It obtained 62.99% of success. We might have reached the point where the model becomes too complex for the problem. In that case, its bad scores can be a result of over-fitting in the model, which means that it could not be able to handle correctly unknown data.

11 – To verify our hypothesis, we tried to raise the complexity of the previous model to have 9 LSTM layers of 200 cells. It obtained a success rate of 63.05%, which is a result similar to the previous one. It showed that we did not gain in success rate any more by making our models more complex.

Categorical cross entropy based

As we wanted our models to obtain better results, we decided to use one-hot encoding for the targets and we used categorical cross entropy as loss metric along with a `softmax` activation function on the output layer. In table 4, we listed all the models of this category that we created for this case study and we indicated the results obtained by these networks in table 5. We plotted these results in figure 32.

ID	X – cells	Y – cells	Dropout (%)	Activ. func.	Nb param.	Batch size
12	3 – 96/layer	1 – 42 cells	40	`tanh`	190,658	20
13	3 – 96/layer	1 – 42 cells	40	`relu`	190,658	20
14	3 – 96/layer	1 – 42 cells	40	`tanh`	190,658	1
15	5 – 96/layer	1 – 42	40	`tanh`	338,882	20
16	5 – 200/layer	1 – 42	20	`tanh`	1,454,386	20
17	6 – 200/layer	1 – 100	20	`tanh`	1,787,308	1
18	7 – 200/layer	1 – 100	20	`tanh`	2,108,108	20
19	9 – 200/layer	1 – 42	20	`tanh`	2,737,586	20

Table 4: Table containing the recurrent architectures that were trained using categorical cross entropy as loss metric

Notes on the listed models

12 – For the first model of this category, we chose the settings to mimic model 6, which obtained good results. This new model obtained 54.44% of success, which is disappointing compared to the results of model 6. To understand these bad results, we analysed the output values of this network. For each input, the network returns a list of probabilities to belong to a certain class. We have 8 classes for this game: the first one representing the neutral value, and the seven others representing the 7 columns of the game board. The only probability

ID	Wins	Draws	Defeats	Success rate (%)
12	5444	361	4195	54.44
13	4901	364	4735	49.01
14	5115	379	4506	51.15
15	4692	397	4911	46.92
16	5137	369	4494	51.37
17	5259	368	4373	52.59
18	4968	369	4663	49.68
19	5215	346	4439	52.15

Table 5: Table containing the results of the models for 10,000 games against the random AI

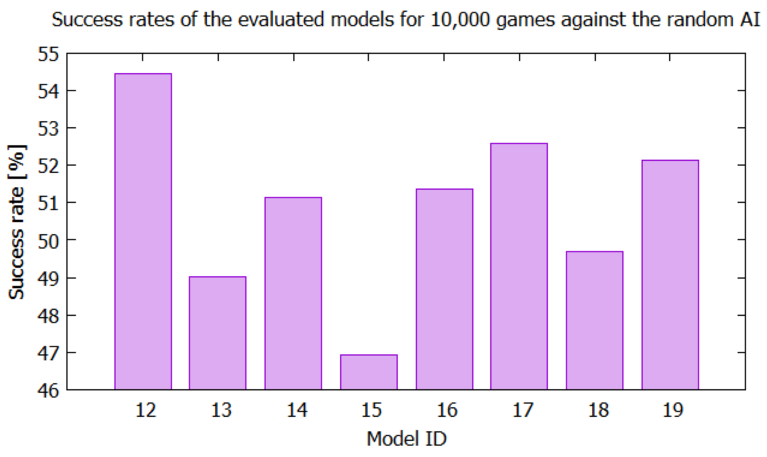


Figure 32: Bar chart of the success rate that our models obtained for 10,000 games against the random AI

that seemed correct in the outputs was the neutral value probability: either it was close to 0, or close to 1. The seven last probabilities were nearly equal every time. This means that the model did not know for sure in which column put the disc at any moment. As these values are the results of the network's activation function, we tried to change it in the next model.

13 – This model uses the same parameters as model 12 apart from the activation function, that we set in this case to `relu` for the hidden layers. This network obtained 49.01% of wins, hence it seemed that it was not because of the activation function that we obtained bad results. To train the previous model, we used 20 as the batch size. In the previous category, model 7 obtained better results than model 6 because we reduced the batch size during training. Hence, for the next model, we will reduce the batch size during the training using the previous model too to check if it obtains better results.

14 – Using a training batch size of one in model 12 gave 51.15% of success rate. The results are still below the models of the first category. Hence, we tried to make the next models more complex to check if we needed a model that was more complex to evaluate the actions correctly.

15-19 – We evaluated the results of five models of different complexity, we even trained model 17 with a batch size of 1, but we never obtained results that exceeded 53% of success. The problem was always the same, the probabilities were very close to each other, which means that the network was never really sure of the action to select. To solve this problem, we tried to encode the inputs differently. It led to the next category of networks.

Qualitatively encoded inputs

To improve the results of our networks, we encoded our inputs qualitatively, as explained earlier. In table 6, we listed all the models of this category that we created for this case study and we indicated the results obtained by these networks in table 7. We also plotted these results in figure 33.

ID	X – cells	Y – cells	Dropout (%)	Activ. func.	Nb param.	Batch size
20	3 – 200/layer	1 – 42 cells	20	`tanh`	823,986	20
21	5 – 200/layer	1 – 42 cells	20	`tanh`	1,465,586	20
22	7 – 200/layer	1 – 42 cells	20	`tanh`	2,107,186	20
23	15 – 200/layer	1 – 42 cells	20	`tanh`	4,673,586	20

Table 6: Table containing the recurrent architectures that were trained using categorical cross entropy as loss metric and using inputs encoded qualitatively

ID	Wins	Draws	Defeats	Success rate (%)
20	3543	327	6130	35.43
21	3467	364	6169	34.67
22	3462	319	6219	34.62
23	3433	360	6207	34.33

Table 7: Table containing the results of the models for 10,000 games against the random AI

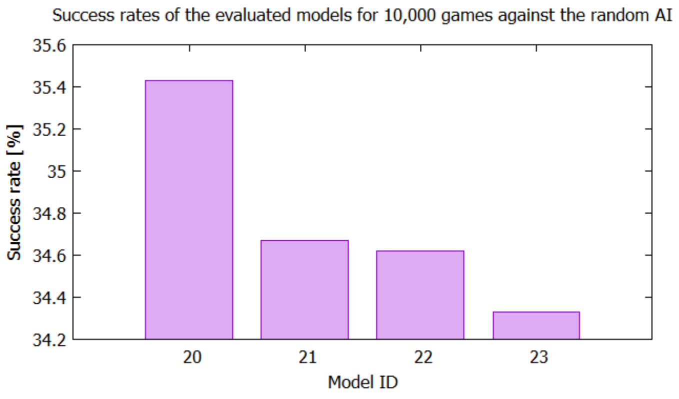


Figure 33: Bar chart of the success rate that our models obtained for 10,000 games against the random AI

Notes on the listed models

20-23 – We tried multiple complexities for the models of this category, but they all failed to dominate the random AI. They were worse than the random AI because they tended to fill each column one after the other, while the random AI played more diversely, conducting to winning strategies. The last two categories should have obtained better results than the first one. In practice, it was not the case. This can be explained by the fact that the data was probably too diverse for a categorical approach. Indeed, from a given non terminal action sequence, there can be multiple good actions to choose. The problem of these categories was that their models never knew what action was better than any other, as all the probabilities were nearly equal. This shows that these networks could not learn winning strategies with a categorical approach. This could be due to the subjectivity problem mentioned previously.

3.3 P2 – Comparison with ML techniques

In this part we will compare the results of the classical machine learning techniques with the two best neural networks of the previous step. We trained the HMM model using one-hot encoding for the inputs, as its functioning requires. On the other hand, we could choose the input of the CRFs to be either a sequence of actions encoded as integers, or qualitative vectors as we did in the third category of P1 for the neural networks. We chose to do both, so that we have two models to train and to compare. We will call CRF_1 the model taking classical sequences of player actions and CRF_2 the model taking sequences of qualitatively encoded actions.

We will begin by comparing the results of the HMM model and the CRF models by making them fight 10,000 games against the random AI. Their results after 10,000 games against the random AI are displayed in table 8. The three models dominate the random AI, obtaining approximately 60% of wins.

Model	**Wins**	**Draws**	**Defeats**	**Success rate (%)**
HMM	5702	1445	2853	57.02
CRF_1	5944	332	3724	59.44
CRF_2	5925	289	3786	59.25

Table 8: Results of 10,000 games of classical machine learning techniques against the random AI

The best networks we obtained in P1 were models 6 and 7. We made them fight 10,000 games against the three classical models. We listed their results in table 9, and we plotted them in figure 34.

PL1	*PL2*	**PL1 wins**	**PL1 draws**	**PL1 defeats**	**Success rate (%)**
Model 6	HMM	5073	253	4674	50.73
Model 6	CRF_1	5505	284	4211	55.05
Model 6	CRF_2	5413	317	4270	54.13
Model 7	HMM	6195	415	3390	61.95
Model 7	CRF_1	5755	449	3796	57.55
Model 7	CRF_2	5732	490	3778	57.32

Table 9: Results of 10,000 games of classical machine learning techniques against neural networks-based AIs

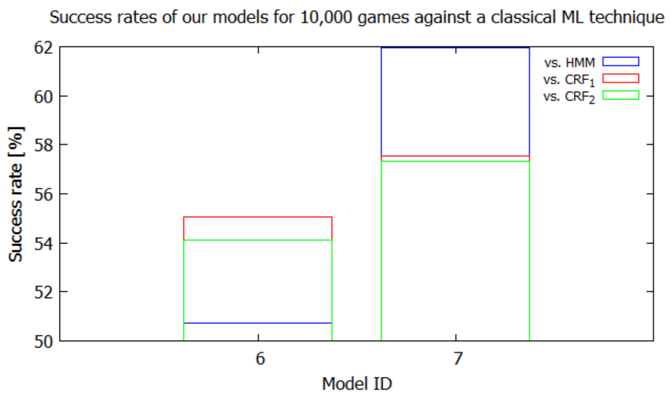


Figure 34: Bar chart of the success rate that our models obtained for 10,000 games against classical ML techniques

Our neural networks tend to dominate the classical machine learning techniques when playing the game, even though the latter learned using the entire dataset and neural networks learned using a fraction (∼10%) of the dataset. It shows that neural networks can get better results than classical machine learning techniques to predict complex time series.

3.4 P3 – Validation

To evaluate the strength of our neural networks, we made two of them (models 6 and 7) fight against an AI based on the Alpha Beta algorithm. The results of 10,000 games are shown on table 10.

Model	**Wins**	**Draws**	**Defeats**	**Success rate (%)**
Model 6	2767	962	6271	27.67
Model 7	3030	949	6021	30.30

Table 10: Results of 10,000 games of neural networks-based AIs against an Alpha Beta based AI

These results are disappointing. Nevertheless, the data with which our neural networks were trained were generated using randomly generated game states, as explained in section 11.2 of part III. Hence, winning approximately 30% of the games and obtaining almost 10% of draws against an AI that analyses the game state in real time is not so bad. If the models were trained using data of higher quality, generated using real games that were won by good AIs against other good AIs, the neural networks would have outperformed the Alpha Beta. Unfortunately, we lacked time and resources to generate such data...

3.5 Summary

For this first case study, we trained three categories of neural networks designed to play the Connect 4 game. Only the first category gathered networks that obtained more than 60% of wins when fighting against the random AI. The two categories that failed to obtain good results mainly differed from the other by the encoding of their inputs and outputs. Indeed, with these categories, we tried a qualitative approach of the data, which did not succeed at all, probably because of the subjectivity problem.

After we selected the two best models using their results against the random AI, we made them fight against classical machine learning techniques. The results were good as our models outperformed the three different classical techniques, proving that neural networks are amongst the best models to predict time series such as action sequences. Unfortunately, our models did not dominate an alpha beta based AI. This can be explained by the the poor quality of the data used to train the models. Indeed, the data was generated on a random basis, and training the models using data of better quality would have greatly helped to beat the alpha beta AI, especially if the data was generated using real games played by good AIs. Even though the results are disappointing, our models represent a good proof of concept regarding action sequences learning. Indeed, they obtained approximately 30% of wins against the alpha beta AI, despite their poor training.

4 Case study – Lazerbike

In the first case study, we used a turn-based game to perform the tests. We will now use a real-time game to evaluate the performances of our neural networks-based AIs. To do so, we chose the Lazerbike game, which is a real-time game that can be played by two players. As stated for the first case study, some choices taken during this case study may be arbitrary because of the huge amount of time and resources taken by training neural networks.

4.1 P0 – Learning data generation

Generating data for this game was harder than for the Connect 4 game. Indeed, there exist more moves that can end the game. It is hence easier to reach a final game state during the randomized data generation, in which case the algorithm must roll back to reach a non-final game state. Because of this problem, it may take a long time before reaching a non-terminal state on which collect data. We had three multi-threaded computers running for two weeks to collect the data. In the same way as for the previous game, we also gathered game board information during the data generation. Eventually, $1\ 537\ 371$ action sequences were generated. For this game, an action is an integer $\in -2, -1, 0, 1$. -2 is the neutral value, it means that the player linked with this action is already dead. We also used this value to pad our sequences to prepare the neural networks training. The other actions represent the direction chosen by the player: -1 means that the bike turned left, 0 that it continued forward and 1 that it turned right. Indeed, in this game, these are the only directions allowed at any moment for the players.

4.2 P1 – Models

We used the Connect 4 case study to conduct the training of our models, following the set of parameters that worked well. Based on those results, we only considered the first category of models in this case study, in which the networks are trained using mean square error as loss metric. We used sequences of 112 actions to train the model. Indeed, the game is played on a game board containing 225 tiles, which means that there are at most 113 moves that can be performed.

In the same way as for the first case study, we noticed that the machine learning models committed suicide too often. We fixed that problem by providing these models a non-suicidal move selection. If the move they selected would be suicidal, we force them to select another move if possible. The random AI that will compete against our models has been provided this feature as well. Consequently, there are only two ways to win a game: either the opponent sets a trap for itself from which it cannot escape, as illustrated in figure 35, or the player must set a trap for its opponent, as illustrated on figure 36.

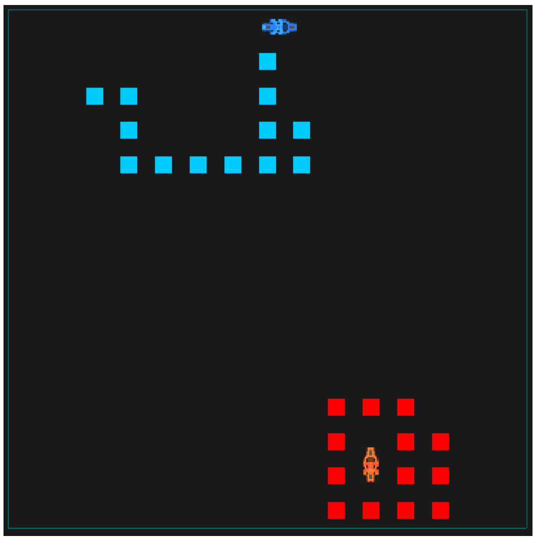


Figure 35: The red player has set a trap for himself, he has no way out

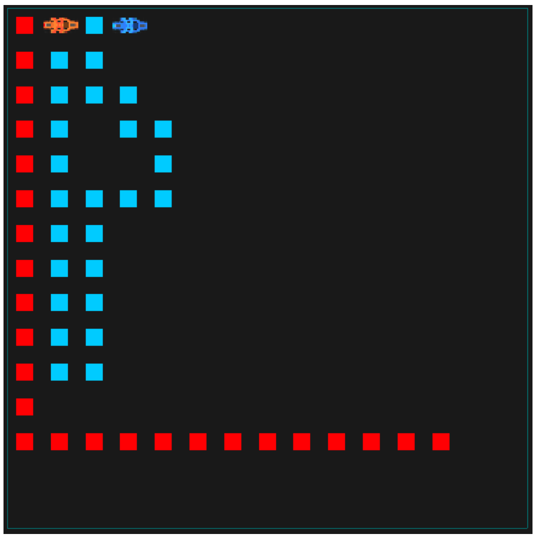


Figure 36: The blue player has set a trap for the red one, the latter has no way out

The models we used for this case study are listed on table 11, and their results for 10,000 games against the random AI are available on table 12 and on figure 37.

ID	X – cells	Y – cells	Dropout (%)	Activ. func.	Nb param.	Batch size
1	3 – 96/layer	1 – 42 cells	0	`tanh`	190,357	20
2	3 – 96/layer	1 – 42 cells	40	`tanh`	190,357	20
3	5 – 96/layer	1 – 42 cells	40	`tanh`	338,581	20
4	3 – 96/layer	1 – 42 cells	40	`tanh`	190,357	1
5	3 – 450/layer	1 – 225 cells	40	`tanh`	4,160,701	20
6	3 – 96/layer	1 – 42 cells	40	`tanh`	190,486	20

Table 11: Table containing the recurrent architectures that were trained using categorical cross entropy as loss metric and using inputs encoded qualitatively

ID	Wins	Draws	Defeats	Success rate (%)
1	5022	791	4187	50.22
2	5865	1075	3060	58.65
3	5649	1038	3313	56.49
4	4160	627	5213	41.60
5	4642	646	4712	46.42
6	3478	463	6059	34.78

Table 12: Table containing the results of the models for 10,000 games against the random AI

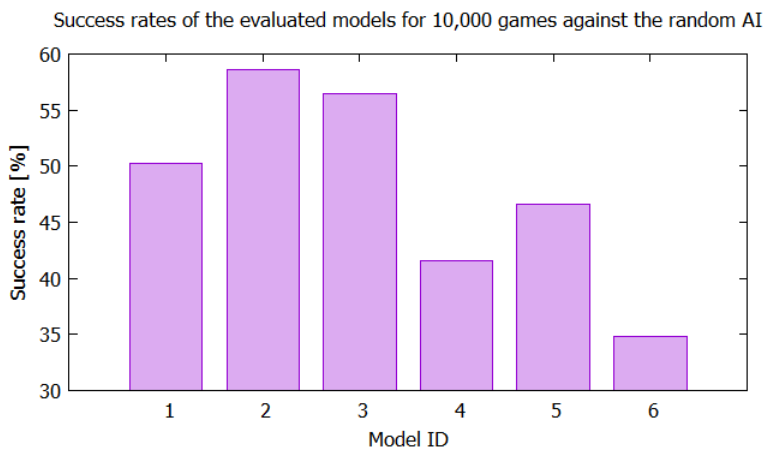


Figure 37: Bar chart of the success rate that our models obtained for 10,000 games against the random AI

Notes on the listed models

1 – For this first model, we tried to adapt the architecture that obtained the best results in the first case study to the lazerbike game. We first tried with a dropout probability of 0% to check if increasing this probability improved the results in this case study as well. This model obtained 50.22% of success rate against the random AI, which is not great.

2 – To improve the results, we tried to raise the dropout probability to 40% in the second model. As expected, it obtained better results than the first model, with a success rate of 58.22%.

3 – We tried to increase the complexity of the model to check if the resulting model increased its success rate. This is why we added 2 LSTM layers to the second model to obtain this third model. As it did not obtain better results than the second model, we can suppose that increasing complexity does not produce significantly better results at this point.

4 – In the first case study, training the models with a batch size of 1 improved the results of the models. This is why we tried to train the second model using 1 as batch size. Surprisingly, this fourth model obtained only 41.60% of success rate against the random AI. We can explain this bad rate by an over-fitting of the training data. Indeed, the action sequences of this game are longer than the ones used in the previous case study, and it may affect the training process, which may be more prone to over-fitting.

5 – In the first case study, we chose the number of neurons by taking two times the number of tiles in the game board. This is why we tried to do the same with this fifth model, increasing the number of neurons per LSTM layer to 450. This model obtained only 46.42% of success.

6 – Finally, we tried the categorical approach in this second case study to check if the bad results obtained in the first one where related to the game complexity. We adapted the second model for it to return categorical outputs. After 10,000 fights against the random AI, it obtained only 3478 wins. As the results were bad in both case studies, we can conclude that the categorical approach does not work well to predict action sequences.

4.3 P2 – Comparison with ML techniques

We built three classical machine learning models to compete against our neural networks. We designed them the same way as for the previous case study: one HMM-based model, and two CRF-based models. In a similar way as previously, the second CRF model has the particularity to take qualitative vectors as input, while the first one takes quantitative values for the actions.

We will start by evaluating these classical models by making them fight 10,000 games against the random AI. Their results are displayed in table 13. The HMM model obtained very good results, while the two others obtained mixed results.

Model	**Wins**	**Draws**	**Defeats**	**Success rate (%)**
HMM	7020	851	2129	70.20
CRF_1	4494	679	4827	44.94
CRF_2	5039	854	4107	50.39

Table 13: Results of 10,000 games of classical machine learning techniques against the random AI

The best networks we obtained in the last section were models 2 and 3. We evaluated them against the classical models during 10,000 games. We listed their results in table 14 and we plotted them in figure 38.

PL1	*PL2*	**PL1 wins**	**PL1 draws**	**PL1 defeats**	**Success rate (%)**
Model 2	HMM	3681	3201	3118	36.81
Model 2	CRF_1	5183	1559	3118	51.83
Model 2	CRF_2	4902	1320	3778	49.02
Model 3	HMM	3536	5341	1123	35.36
Model 3	CRF_1	5934	2001	2065	59.34
Model 3	CRF_2	6160	1677	2163	61.60

Table 14: Results of 10,000 games of classical machine learning techniques against neural networks-based AIs

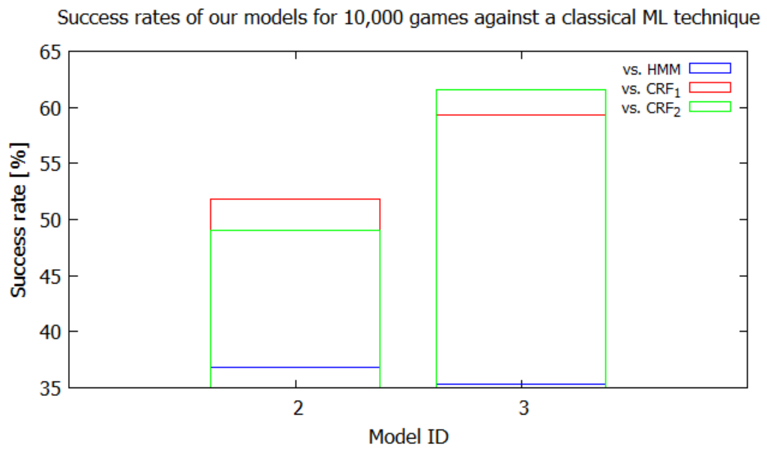


Figure 38: Plot of the success rate that our models obtained for 10,000 games against classical ML techniques

Our models neutralized the HMM model, but it obtained several draws. Model 3 obtained better results than Model 2 against the CRF models. As our models did not clearly beat the HMM model, we will keep this for the validation step.

4.4 P3 – Validation

To evaluate the quality of our neural networks, we took models 2 and 3 to make them fight against an AI based on the Alpha Beta algorithm. The results of their 10,000 games are shown on table 15 and plotted on figure 39. We also made the HMM model fight against the same Alpha Beta based AI so that we could compare their respective results.

Model	**Wins**	**Draws**	**Defeats**	**Success rate (%)**
Model 2	4878	623	4499	48.78
Model 3	4848	650	4502	48.48
HMM	3781	533	5686	37.81

Table 15: Results of 10,000 games of neural networks-based AIs against an Alpha Beta based AI

The results show that our models can compete against the AlphaBeta-based AI, obtaining approximately 48% of success. Nevertheless, the evaluation function used in the AlphaBeta algorithm is a naive function which rewards in the case of a victory and punish in the case of a defeat. However, the results are positive as the training data was gathered on a random basis. The HMM-based AI tends to lose against the AlphaBeta AI, proving that our neural network-based AI are better than classical models-based AIs.

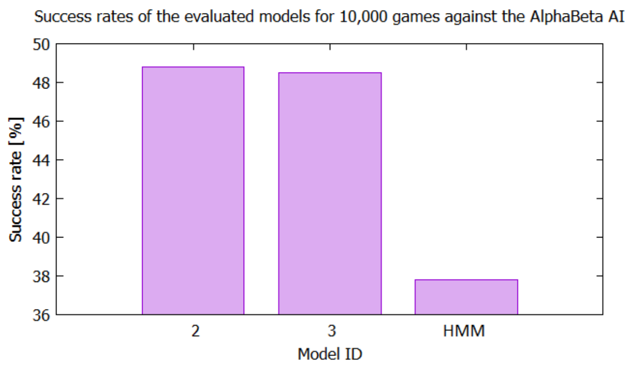


Figure 39: Bar chart of the success rate that our models obtained for 10,000 games against the AlphaBeta-based AI

4.5 Summary

During this case study, we confirmed that the categorical approach does not work well in practice, and we experienced that taking a small batch size or too many neurons per layer could easily lead to an over-fitting model in complex games. Then, we compared our models with classical machine learning models. The results were not totally convincing, this is why we included the HMM model in the validation step in order to compare its results with our models. Finally, our two best models obtained good results against a naive AlphaBeta-based AI, while the HMM model tended to lose against it.

We did not evaluate our models against a smarter AlphaBeta AI because our models did not clearly beat the naive one. Nevertheless, we provided a good proof of concept regarding this action sequences learning technique, and we should be able to improve the results of the models by training them using data of better quality.

5 Further work and encountered difficulties

To perform these case studies, we faced some difficulties. We listed some of them in this section.

5.1 Resources

Making neural networks learn and gathering huge amount of data are very time-consuming tasks, especially when they are executed on personal computers. Such computations are usually performed on powerful servers using high-end GPUs and CPUs. We struggled to have access to such computers, so we had to transform each PC we had access to into low-end computations servers, which were not optimized for heavy learning processes, nor heavy data collection. As a result, we had to limit the number and the diversity of the models we trained, and we sometimes needed to make arbitrary choices when setting the parameters of our neural networks.

5.2 Nature of data

The data that we used to train our neural networks in the case studies were action sequences. This data is not easy to manipulate in machine learning. Indeed, during the learning process of our neural networks, we could not use validation data to evaluate the quality of the training because choosing the right action at a given moment is subjective until the game ends. This is why the only validation we could do was to make our AIs fight against other AIs and observe their results. Furthermore, time sequences are hard to illustrate, which did not ease the pre-analysis of the data.

5.3 Future work

To finalise the validation of our research over neural networks learning using action sequences, we would need data of better quality. Indeed, we should perform further experiments using data coming from real games played either by good human players or by good AIs, so that neural networks can try to mimic their strategies. We developed methods, introduced in section 11 of part III, in the game engine that can generate data from AIs. To collect data coming from human players, we could develop a website in which human players could play, so that our AIs could learn from their strategies. It could be interesting to continue the research on action sequences learning because it does not require to extract features and metrics from the game board. All we had to do is to take the raw action sequences from played games.

Furthermore, two of the learning techniques proposed in section 1 were not explored in this masters thesis and deserve to be experimented and compared between each others. Indeed, the only technique that was explored involved action sequences, while the other techniques we presented do not need time dependencies in their learning data, and use different neural network architectures such as MLP or CNN. The disadvantage of these two techniques is that they require metrics coming from the game board so that they can evaluate the "score" of each action for the current state of the game board. We did not have this constraint with action sequences because the sequences we used to train our network led to victory.

Part V Conclusion

For this masters thesis, we developed a new game engine that was designed to facilitate the prototyping of simple grid-based games and their AIs. The developed solution meets the expectation and provides a new and easy way to develop grid-based games. Indeed, our framework offers a simple technique to automatically generate the main files and methods needed to create the new game.

In this engine, we designed four techniques to automatically generate game data designed to be used to make machine learning models learn how to play the game. These techniques have been developed generically so that they could be used with any game developed within our framework. We used one of these techniques, the randomized one, to generate data for two grid-based games: Connect 4 and Lazerbike.

We proposed three techniques to make neural networks learn how to play grid-based video games. These three techniques each rely on a different neural network architecture. We performed two case studies to validate the technique using recurrent neural networks and action sequences. We chose this technique because it is the one that requires the least game knowledge to be effective. Indeed, generating action sequences only requires game simulations.

The first case study aimed to train recurrent neural networks to play the Connect 4 game. We tried multiple combinations of parameters to obtain competitive models and we realised that the quantitative approach worked better than the qualitative approach, even though our data is qualitative. We showed that our neural networks are better than classical machine learning models by making them fight against each other. Finally, we made our two best models fight against an AI based on the AlphaBeta algorithm to evaluate their strength. Our models obtained disappointing results as they were beaten most of the time by the AlphaBeta AI. Nevertheless, they obtained approximately 30% of victory, which is encouraging because our models were trained with randomly generated data.

The second case study focused on the Lazerbike game. The same way as for the first case study, we trained neural networks using multiple set of parameters. We confirmed that the qualitative approach does not work well for action sequences learning. The comparison between our neural networks and the classical machine learning models obtained mixed results. Nevertheless, our models obtained better results against an AlphaBeta-based AI than the classical models. Their results were more convincing than during the first case study. Indeed, our models obtained nearly 50% of success against the AlphaBeta AI.

The results of the case studies show that our learning technique can provide good results, even if the data used is of poor quality. Hence, we provided good proofs of concept that learning action sequences can give good results to make recurrent neural networks learn. Moreover, this type of data is easy to generate from real or generated games. The next step to validate our technique would be to train our neural networks using data coming from real competitive games in order to evaluate the capacity of our models to mimic good strategies.

Acknowledgement

We would like to thank the director of this masters thesis: Tom Mens, who supported us all along the way, as well as Alexandre Decan, who directed our way of thinking about using neural networks to play the games.

References

[1] Michael Bowling et al. "Machine learning and games". In: *Machine learning* 63.3 (2006), pp. 211–215.

[2] David W Aha, Héctor Muñoz-Avila and Michael van Lent. "Reasoning, Representation, and Learning in Computer Games". In: *Workshop Committee*. 2005.

[3] Ilya Sutskever, Oriol Vinyals and Quoc V. Le. "Sequence to Sequence Learning with Neural Networks". In: *CoRR* abs/1409.3215 (2014). URL: http://arxiv.org/abs/1409.3215.

[4] Forrest N Iandola et al. "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size". In: *arXiv preprint arXiv:1602.07360* (2016).

[5] Mario Zechner and LibGDX community. *LibGDX*. 2014. URL: https://libgdx.badlogicgames.com/.

[6] Juan Linietsky and Ariel Manzur. *GODOT. Game engine*. 2008. URL: https://godotengine.org/.

[7] Kenneth D Forbus and John Laird. "AI and the entertainment industry". In: *IEEE Intelligent Systems* 17.4 (2002), pp. 15–16.

[8] M. I. Jordan and T. M. Mitchell. "Machine learning: Trends, perspectives, and prospects". In: *Science* 349 (6245 2015), pp. 255–260.

[9] Eduard Sackinger et al. "Application of the ANNA neural network chip to high-speed character recognition". In: *IEEE Transactions on Neural Networks* 3.3 (1992), pp. 498–505.

[10] Raghuraj Singh et al. "Optical character recognition (OCR) for printed devnagari script using artificial neural network". In: *International Journal of Computer Science & Communication* 1.1 (2010), pp. 91–95.

[11] Steve Lawrence et al. "Face recognition: A convolutional neural-network approach". In: *IEEE transactions on neural networks* 8.1 (1997), pp. 98–113.

[12] Geoffrey Hinton et al. "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups". In: *IEEE Signal Processing Magazine* 29.6 (2012), pp. 82–97.

[13] Heng Guo and Saul B Gelfand. "Classification trees with neural network feature extraction". In: *IEEE Transactions on Neural Networks* 3.6 (1992), pp. 923–933.

[14] Jonathan Masci et al. "Stacked convolutional auto-encoders for hierarchical feature extraction". In: *Artificial Neural Networks and Machine Learning–ICANN 2011* (2011), pp. 52–59.

[15] Varun Kumar Ojha, Ajith Abraham and Václav Snášel. "Metaheuristic design of feedforward neural networks: A review of two decades of research". In: *Engineering Applications of Artificial Intelligence* 60 (2017), pp. 97–116.

[16] Nikos Paragios. *Computer Vision Research: The deep "depression"*. 2016. URL: https://www.linkedin.com/pulse/computer-vision-research-my-deep-depression-nikos-paragios.

[17] Pete Shinners. *PyGame*. http://pygame.org/. 2011.

[18] Sam Lantinga and SDL community. *SDL*. 1998. URL: https://www.libsdl.org/.

[19] JG In. *Cocos2d-x 3 Mobile game programming*. 2014.

[20] SFT Tech. *openage*. http://www.openage.sft.mx. 2014.

[21] The OpenRA developers. *OpenRA*. http://www.openra.net/. 2007.

[22] OpenRTS developers. *OpenRTS*. https://github.com/methusalah/OpenRTS. 2014.

[23] Spring developers. *Spring*. http://springrts.com. 2005.

[24] Turbulenz Labs. *Turbulenz*. http://biz.turbulenz.com/. 2012.

[25] Sparklin Labs. *superpowers*. http://superpowers-html5.com/. 2016.

[26] Donald E Knuth and Ronald W Moore. "An analysis of alpha-beta pruning". In: *Artificial intelligence* 6.4 (1975), pp. 293–326.

[27] Cameron B Browne et al. "A survey of monte carlo tree search methods". In: *IEEE Transactions on Computational Intelligence and AI in games* 4.1 (2012), pp. 1–43.

[28] Frederik Frydenberg et al. "Investigating MCTS modifications in general video game playing". In: *Computational Intelligence and Games (CIG), 2015 IEEE Conference on*. IEEE. 2015, pp. 107–113.

[29] Daniel T Larose. *Discovering knowledge in data: an introduction to data mining*. John Wiley & Sons, 2014. Chap. 7.

[30] Min-Ling Zhang and Zhi-Hua Zhou. "ML-KNN: A lazy learning approach to multi-label learning". In: *Pattern recognition* 40.7 (2007), pp. 2038–2048.

[31] John R Quinlan et al. "Learning with continuous classes". In: *5th Australian joint conference on artificial intelligence*. Vol. 92. Singapore. 1992, pp. 343–348.

[32] J Ross Quinlan. "C4. 5: Programming for machine learning". In: *Morgan Kauffmann* 38 (1993).

[33] Arthur L Samuel. "Some studies in machine learning using the game of checkers". In: *IBM Journal of research and development* 3.3 (1959), pp. 210–229.

[34] Arthur L Samuel. "Some studies in machine learning using the game of checkers. II—recent progress". In: *IBM Journal of research and development* 11.6 (1967), pp. 601–617.

[35] Gerald Tesauro. "Temporal difference learning and TD-Gammon". In: *Communications of the ACM* 38.3 (1995), pp. 58–68.

[36] Ben G Weber and Michael Mateas. "A data mining approach to strategy prediction". In: *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*. IEEE. 2009, pp. 140–147.

[37] David Taralla et al. "Decision making from confidence measurement on the reward growth using supervised learning: A study intended for large-scale video games". In: *Proceedings of the 8th International Conference on Agents and Artificial Intelligence (ICAART 2016)-Volume 2*. 2016, pp. 264–271.

[38] Georgios N Yannakakis et al. "Performance, robustness and effort cost comparison of machine learning mechanisms in FlatLand". In: *Proceedings of the 11th Mediterranean Conference on Control and Automation MED'03*. 2003.

[39] Christian Thurau, Christian Bauckhage and Gerhard Sagerer. "Imitation learning at all levels of game-AI". In: *Proceedings of the international conference on computer games, artificial intelligence, design and education*. Vol. 5. 2004.

[40] Kenneth O Stanley, Bobby D Bryant and Risto Miikkulainen. "Evolving neural network agents in the NERO video game". In: *Proceedings of the IEEE* (2005), pp. 182–189.

[41] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529.7587 (2016), pp. 484–489.

[42] Volodymyr Mnih et al. "Playing atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602* (2013).

[43] Sean Klein. "CS229 Final Report Deep Q-Learning to Play Mario". In: ().

[44] Alexander Braylan et al. "Reuse of neural modules for general video game playing". In: *arXiv preprint arXiv:1512.01537* (2015).

[45] Frank Rosenblatt. "The perceptron: A probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6 (1958), p. 386.

[46] Bernard Gosselin. *Course of Statistical Pattern Recognition*. UMONS, 2017.

[47] Eric W. Weisstein. *Sigmoid function*. From MathWorld–A Wolfram Web Resource. URL: http://mathworld.wolfram.com/SigmoidFunction.html.

[48] Eric W. Weisstein. *Hyperbolic Tangent*. From MathWorld–A Wolfram Web Resource. URL: http://mathworld.wolfram.com/HyperbolicTangent.html.

[49] Vinod Nair and Geoffrey E Hinton. "Rectified linear units improve restricted boltzmann machines". In: *Proceedings of the 27th international conference on machine learning (ICML-10)*. 2010, pp. 807–814.

[50] Marvin Minsky and Seymour Papert. *Perceptrons: an introduction to computational geometry (expanded edition)*. 1988.

[51] Kurt Hornik, Maxwell Stinchcombe and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural networks* 2.5 (1989), pp. 359–366.

[52] Léon Bottou. "Stochastic Gradient Learning in Neural Networks". In: *Proceedings of Neuro-Nîmes 91*. Nimes, France: EC2, 1991. URL: http://leon.bottou.org/papers/bottou-91c.

[53] Hopfield JJ. "Neurons with graded response have collective computational properties like those of two-state neurons". In: (Proceedings of the National Academy of Sciences of the United States of America). Vol. 81(10). 1986, pp. 3088–3092.

[54] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[55] Springer, ed. *Hierarchical Neural Networks for Image Interpretation*. (Lecture Notes in Computer Science). 2766 vols. 2003.

[56] Gaurang Panchal et al. “Behaviour analysis of multilayer perceptrons with multiple hidden neurons and hidden layers”. In: *International Journal of Computer Theory and Engineering* 3.2 (2011), p. 332.

[57] Hrushikesh Narhar Mhaskar and Charles A Micchelli. “How to choose an activation function”. In: *Advances in Neural Information Processing Systems* (1994), pp. 319–319.

[58] D Randall Wilson and Tony R Martinez. “The general inefficiency of batch training for gradient descent learning”. In: *Neural Networks* 16.10 (2003), pp. 1429–1451.

[59] Diederik Kingma and Jimmy Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).

[60] Timothy Dozat. *Incorporating Nesterov momentum into Adam*. Tech. rep. Stanford University, Tech. Rep., 2015.[Online]. Available: http://cs229. stanford. edu/proj2015/054 report. pdf, 2015.

[61] John Duchi, Elad Hazan and Yoram Singer. “Adaptive subgradient methods for online learning and stochastic optimization”. In: *Journal of Machine Learning Research* 12.Jul (2011), pp. 2121–2159.

[62] Matthew D Zeiler. “ADADELTA: an adaptive learning rate method”. In: *arXiv preprint arXiv:1212.5701* (2012).

[63] Sebastian Ruder. “An overview of gradient descent optimization algorithms”. In: *CoRR* abs/1609.04747 (2016). URL: http://arxiv.org/abs/1609.04747.

[64] Peter Sadowski. *Notes on backpropagation*. 2016.

[65] Beng Chin Ooi, Ken J McDonell and Ron Sacks-Davis. “Spatial kd-tree: An indexing mechanism for spatial databases”. In: *IEEE COMPSAC*. Vol. 87. 1987, p. 85.

[66] Louis Victor Allis. *A knowledge-based approach of connect-four*. Citeseer, 1988.

[67] James D Allen. “A note on the computer solution of Connect-Four”. In: *Heuristic Programming in Artificial Intelligence* 1.7 (1989), pp. 134–135.

[68] John Tromp. 2012. URL: https://oeis.org/A212693.

[69] Timo Virkkala. “Solving Sokoban”. In: (2011). URL: http://weetu.net/Timo-Virkkala-Solving-Sokoban-Masters-Thesis.pdf.

[70] Guido van Rossum, Barry Warsaw and Nick Coghlan. *PEP 8 – Style Guide for Python Code*. 2001. URL: https://www.python.org/dev/peps/pep-0008/.

[71] Kairanbay Magzhan and Hajar Mat Jani. “A review and evaluations of shortest path algorithms”. In: ().

[72] Thomas G Dietterich. “Machine learning for sequential data: A review”. In: *Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR)*. Springer. 2002, pp. 15–30.

[73] Lawrence R Rabiner. “A tutorial on hidden Markov models and selected applications in speech recognition”. In: *Proceedings of the IEEE* 77.2 (1989), pp. 257–286.

[74] John Lafferty, Andrew McCallum, Fernando Pereira et al. “Conditional random fields: Probabilistic models for segmenting and labeling sequence data”. In: *Proceedings of the eighteenth international conference on machine learning, ICML*. Vol. 1. 2001, pp. 282–289.

[75] Kyunghyun Cho et al. “Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation”. In: *CoRR* abs/1406.1078 (2014). URL: http://arxiv.org/abs/1406.1078.

[76] Yarin Gal and Zoubin Ghahramani. “A theoretically grounded application of dropout in recurrent neural networks”. In: *Advances in Neural Information Processing Systems*. 2016, pp. 1019–1027.

[77] Nitish Srivastava et al. “Dropout: a simple way to prevent neural networks from overfitting.” In: *Journal of Machine Learning Research* 15.1 (2014), pp. 1929–1958.

[78] Vu Pham et al. “Dropout improves recurrent neural networks for handwriting recognition”. In: *Frontiers in Handwriting Recognition (ICFHR), 2014 14th International Conference on*. IEEE. 2014, pp. 285–290.

[79] Mykola Pechenizkiy, Seppo Puuronen and Alexey Tsymbal. “Feature extraction for classification in the data mining process”. In: (2003).

[80] François Chollet. *Keras*. https://github.com/fchollet/keras. 2015.

[81] Theano Development Team. “Theano: A Python framework for fast computation of mathematical expressions”. In: *arXiv e-prints* abs/1605.02688 (May 2016). URL: http://arxiv.org/abs/1605.02688.

[82] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: http://tensorflow.org/.

[83] Rutherford Aris. *Vectors, tensors and the basic equations of fluid mechanics – Tensors*. Courier Corporation, 2012, pp. 134–176.

[84] David Kirk et al. “NVIDIA CUDA software and GPU parallel computing architecture”. In: *ISMM*. Vol. 7. 2007, pp. 103–104.

[85] Soheil Bahrampour et al. “Comparative study of deep learning software frameworks”. In: *arXiv preprint arXiv:1511.06435* (2015).

[86] Naoaki Okazaki. *CRFSuite – A fast implementation of Conditional Random Fields (CRFs)*. http://www.chokkan.org/software/crfsuite/. 2002.