UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC
DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València

Integrating reinforcement learning and automated planning for playing video-games

TRABAJO FIN DE MASTER

Master Universitario en Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital

Author: Daniel Diosdado López
Tutor: Eva Onaindia de la Rivaherrera
Tutor: Sergio Jiménez Celorrio

Course 2018-2019

Resum

Aquest treball descriu un nou enfocament de crear agents capaços de jugar a múltiples videojocs que es basa en un mecanisme que intercala planificació i aprenentatge. La planificació s'utilitza per a l'exploració de l'espai de cerca i l'aprenentatge per reforç s'utilitza per a obtindre informació de recompenses anteriors. Més concretament, les accions de estats visitades pel planificador durant la cerca es basen en un algoritme d'aprenentatge que calcula les estimacions de polítiques en forma de xarxa neuronal que s'utilitzen al seu torn per guiar el pas de la planificació. Així, la planificació s'utilitza per realitzar la cerca del millor moviment en l'espai d'acció i l'aprenentatge s'utilitza per extreure funcions de la pantalla i aprendre una política per millorar la cerca del pas de planificació.

La nostra proposta es basa en un algoritme de planificació basat en Iterated Width juntament amb una xarxa neuronal convolucional per implementar el mòdul de aprenentatge per reforç. Es presenten dues millores sobre el mètode de planificació i aprenentatge base (P&A). La primera millora utilitza la puntuació del joc per a disminuir la poda en el pas de la planificació, i la segona afegeix a la primera un ajustament dels hiperparametres i la modificació de l'arquitectura de la xarxa neuronal per millorar les característiques extretes i augmentar el seu nombre.

Les nostres millores es proven en els jocs de la Atari 2600 del Arcade Learning Environment mitjançant el kit d'eines OpenAI Gym. S'analitzen els resultats i es discuteixen els punts forts i els punts febles d'aquest enfocament.

Paraules clau: Agents, Videojocs, Arcade learning environment, Intel·ligència artificial, Iterated Width

Resumen

Este trabajo describe un novedoso enfoque de crear agentes capaces de jugar a múltiples videojuegos que se basa en un mecanismo que intercala planificación y aprendizaje. La planificación se utiliza para explorar el espacio de búsqueda y el aprendizaje por refuerzo se utiliza para aprovechar la información de recompensas anteriores. Más específicamente, las acciones de estados visitadas por el planificador durante la búsqueda alimentan al algoritmo de aprendizaje que calcula las estimaciones de las políticas en forma de una red neuronal, que a su vez se utilizan para guiar el paso de planificación. Por lo tanto, la planificación se utiliza para llevar a cabo una búsqueda del mejor movimiento en el espacio de acciones y el aprendizaje se utiliza para extraer características de la pantalla y aprender una política para mejorar la búsqueda del paso de planificación.

Nuestra propuesta se basa en un algoritmo de planificación basado en Iterated Width junto con una red neuronal convolucional para implementar el módulo de aprendizaje por refuerzo. Creamos dos mejoras sobre el método básico de planificación y aprendizaje (P&A). La primera mejora usa la puntuación del juego para disminuir la poda en el paso de planificación, y la segunda agrega a la primera un ajuste de los hiperparámetros y la modificación de la arquitectura de la red neuronal para mejorar las características extraídas y aumentar su número.

Nuestras mejoras se prueban en juegos de la Atari 2600 del *Arcade Learning Environment*, utilizando el kit de herramientas *OpenAI Gym*. Se analizan los resultados y se discuten las fortalezas y debilidades de este enfoque.

Palabras clave: Agentes, Video Juego, Arcade learning environment, Inteligencia artificial, Iterated Width

Abstract

This paper describes a novel approach of creating multi-game agents for playing videogames that draws upon a mechanism that interleaves planning and learning. Planning is used for exploration of the search space and reinforcement learning is used to leverage past reward information. More specifically, the state-actions visited by the planner during search are fed to a learning algorithm that calculates policy estimates in the form of a Neural Network which are in turn used to guide the planning step. Thus, planning is used to carry out a search for the best move in the action space and learning is used to extract features from the screen and learn a policy in order to improve the search of the planning step.

Our proposal relies on an Iterated Width-based planning algorithm along with a Convolutional Neural Network for implementing the Reinforcement Learning module. We come up with two enhancements over a base planning and learning (P&L) method. The first improvement uses the score of the game to lessen the pruning in the planning step, and the second one adds to the first one a fine-tuning of the hyperparameters and the modification of the neural network architecture to enhance the features extracted and increase the their number.

Our enhancements are tested on Atari 2600 games from the *Arcade Learning Environment* using the *OpenAI Gym* toolkit. The results are analyzed and the strengths and weaknesses of this approach are discussed.

Key words: Agents, Video Game, Arcade learning environment, Artificial intelligence, Iterated Width

Contents

List of Figures

List of Tables

CHAPTER 1
Objectives

One of the most interesting challenges is to make an Artificial Intelligence (AI) agent capable of doing tasks which require an intelligent behaviour. One of this tasks is playing games. Games can be seen as a simplified situation from the real world, so creating agents capable of playing with a certain level can lead to real world applications.

Games, and specifically videogames, make good environments for AI developments since they feature some or even all of the following characteristics:

- Complex strategies: With the exception of the most basic games, in videogames it is necessary to employ advanced strategies to play and beat the games.
- Specific goals: Games usually have a distinct definition of success and defeat.
- Measurable results: Games usually provide a measure of success, which can be used to see how well an agent performs and to compare the performance of different agents.

Among the manifold works in videogame playing, we can differentiate two large groups: researchers who aim to make progress in a single game, such as AlplhaGo or AlphaStar, and those who put the focus on the development of a game agent capable of playing several games such as AlphaZero.

For single games, specific knowledge about the environment can be used. For example, AlphaGo exploits the fact that *Go* is symmetric under certain reflections and rotations, or *Stockfish* uses a database of pre-calculated exhaustive analysis of endgame states.

In the case of multi-game agents, the focus is on the development of techniques and agents capable of adapting to new environments on the fly and learning the underlying rules of different games in order to play them on human or even super-human level.

There are different frameworks already developed to test agents in different environments, such as the Arcade Learning Environment (ALE), the General Video Game AI competition (GVGAI) or OpenAI Gym.

Our goal in this project is to develop an agent capable of playing several games of the Arcade Learning Environment such as *Alien, Battle Zone, Pong* or *Space Invaders* among others. This agent must be able to play different games without previous domain knowledge and learn the games' mechanics by applying state-of-the-art search algorithms, with the objective of achieving the goals of each game. To do so, we will employ a combined technique of planning and learning: planning to carry out a search for the best move in the action space and learning to extract features from the screen and learn a policy to improve the search of the planning step.

CHAPTER 2
Related Work

The videogame industry is one of the fastest growing entertainment sectors in recent years, and it's estimated to generate 152.1 billion of dollars in revenue this year. [51]

Games offer their players a challenge that must be overcome in order to win the game, be it solving a puzzle (*Sokoban*), defeating Non-Player Characters (*Space Invaders*), winning a race (*Mario Kart*) or defeating another player (*Chess, Unreal Tournament, Starcraft*). Creating an Artificial Intelligence (AI onwards) capable of playing these games becomes a challenge, as they must find solutions to problems quickly. Due to this, games have become a popular benchmark for AI.

Many researchers have studied algorithms and techniques that try to approximate optimal play in computer games. Research on these games has enabled some interesting advances in algorithmic AI such as the use of parallelized Alpha-Beta pruning (in *Chess*), or the progress seen in one of the most popular algorithms in Game AI, Monte Carlo Tree Search (MCTS), in the game of *Go* [39] [43].

Early research put the focus on board games like *Chess*, *Scrabble* or *Checkers*. Thanks to the advances in the development of tree-search methods, games like *Scrabble* or *Checkers* are now resolved [40].

Nowadays researchers are more interested in video games where the complexity is harder and there is room for improvements. The main games addressed by researchers are those in which an agent can easily learn how to play but are difficult to master. In order to create and evaluate agents developed by researchers, platforms such as The Arcade Learning Environment [8] and the General Video Game - Artificial Intelligence (GVG-AI) [3] have emerged.

2.1 Game definition

Following we will define some relevant concepts of game theory that will allow us to describe and to contextualize the kind of games addressed in this research.

- According to the number of players, games can be divided into:
  - **One-player games.** There is only one character in the game, who must solve the problem to win or obtain the best score possible. This kind of games are sometimes named *puzzle games* [23].
  - **Two-player games.** These games are characterized by having two players with opposing interests trying to achieve the game objectives. Board games like

Chess or *Go* are examples of two-player games with a large tradition in AI for games researches [12].

– **Multi-player games.** Games that feature more than two players belong to this group. The complexity of these games increases because players can form alliances or compete among themselves. In these games, there are common and opposed interests with the rest of players, and in some cases it is necessary to carry out some type of negotiation in the best interest of a subset of players.

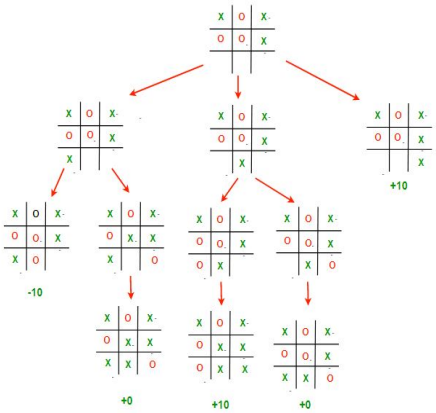
Games are characterized by different features that determine the type of game. Following we enumerate a list of features that are used to describe games. The concepts shown below are not incompatible, so a game may present several of the following features.

- **Simultaneous games vs sequential games.** In game theory, a simultaneous game is a game where each player chooses his action without knowledge of the actions chosen by other players. In other words, more than one player can issue actions at the same time. Games like *Rock-paper-scissors* or *Starcraft* are some examples of simultaneous games [37]. In contrast, sequential games are those in which one player chooses his action before the others choose theirs. In this case, the latter players have information available of the first players' choice, which allows them to create more informed strategies. *Chess*, *Shogi*, *Go* or *Tic-Tac-Toe* are some examples of sequential games.

  In game theory, the representation of the games varies according to the type of game. Simultaneous games are often represented by a matrix that encompasses all possible combinations of players' moves. Each cell specifies the reward and penalty for each player if that combination of moves is feasible. The game *Rock Paper Scissor Lizard Spock* is represented in Figure 2.1a. Sequential games are commonly represented by decision trees in which the time axis is represented by the depth of a node in the tree. In Figure 2.1b we see an example of the game *Tic-Tac-Toe*.

	Rock	Paper	Scissors	Lizard	Spock
Rock	0, 0	-1, 1	1, -1	1, -1	-1, 1
Paper	1, -1	0, 0	-1, 1	-1, 1	1, -1
Scissors	-1, 1	1, -1	0, 0	1, -1	-1, 1
Lizard	-1, 1	1, -1	-1, 1	0, 0	1, -1
Spock	1, -1	-1, 1	1, -1	-1, 1	0, 0

(a) *Rock Paper Scissor Lizard Spock* matrix.



(b) Decision tree in *Tic-Tac-Toe*

Figure 2.1: Game theory representation of simultaneous and sequential games.

- **Zero-sum vs non-zero-sum games.** Zero-sum games are games in which each participant's gain or loss of utility is exactly balanced by the losses or gains of the utility of the other participants. If the total gains of the participants are added up and the total losses are subtracted, they will sum to zero. In this type of games, there does not exist a win-win solution. *Poker* [9] and *Gambling* are popular examples of zero-sum games. Games like *Chess* and *Tennis*, where there is one winner and one loser, are also zero-sum games. In non-zero-sum games, a gain by one player does not

necessarily correspond with a loss by another, because the outcome has net results greater or less than zero. One example of non-zero-sum game is the prisoner's dilemma.

- **Stochastic vs deterministic games.** Stochastic games are dynamic games with probabilistic transitions. In each step, a new random state is created whose distribution depends on the previous state and the actions chosen by the players. In contrast, a game is deterministic if the result of the actions taken by the players leads to completely predictable outcomes [48].

- **Perfect, imperfect and incomplete information games.** Perfect information refers to the fact that each player has the same information that would be available at the end of the game. This is, each player knows or can see other player's moves. A good example would be *Chess*, where each player sees the other player's pieces on the board.

  Imperfect information games are those where players know perfectly the types of other players and their possible strategies, but are unaware of the actions taken by them.

  In incomplete information games, other details of the game are unknown to one or more players. This may be the player's type, their strategies, their payoffs, their preferences or a combination of these. Imperfect information games are therefore incomplete information games but not vice versa.

  Games with simultaneous moves are generally not considered games of perfect information. This is because players hold information that is hidden to the others and so every player must make a move without knowing the opponent's hidden information [25].

- **Combinatorial games.** In this type of games, the difficulty of finding an optimal strategy stems from the combinatorial explosion of possible moves. Combinatorial game theory typically studies sequential games with perfect information. Thanks to the advances in mathematical techniques over the last years, the complexity of combinatorial games has been largely reduced. This improvement has led to the creation of different variants of games with a high combinatorial component. One example is the *Infinite Chess*, where the game is played on an unbounded chessboard [28].

2.2 Overview of control strategies for game agents

In this section we present some of the most relevant control strategies for developing game agents.

- **Hand-coded strategies**

  This type of strategy consists of the design of manually coded heuristic to guide the search process.

  Rule-based controllers use hand-coded strategies that return the action that complies with a set of relatively simple conditions. An example of rule-based controllers can be found in the *Super Mario Bros* game, which defines a particular behaviour where the character Mario constantly runs right and jumps whenever possible. In this case, the game agent would contain a single rule that determines when to jump.

Creating a rule-based agent entails having a detailed knowledge on the dynamics of the game to be able to extract all possible situations and actions. In addition, an unforeseen situation could lead to unexpected behaviours by the agent.

Another limitation of rule-based agents is that rules are derived from the mechanics of the game. As a result, we will get an agent specialized in that game, but unable to adapt its actions to new environments. For example a world-champion *StarCraft* agent will not be able to play *Chess* [11].

- **Combinatorial Search**

  This strategy consists in turning the goal of finding the best move into a search problem. The game is simulated at each player's turn producing all possible move combinations. The result of the simulations is then used to build a search tree that helps find the best possible move [42].

  In most cases the search space is too large and we cannot guarantee the optimal solution, making necessary the implementation of heuristics to guide the search. In dynamic environments, it is also recommended to start the search at every step of the game, because the new information enables to narrow down the search space and discard the exploration of useless parts of the search tree.

  Sequential two-player games typically use this type of control strategy. The complete information of sequential games makes the simulation unique without possible variations as it happens in stochastic games. In addition, the fact that players move by turns provides players with a natural computational time to think about the next move.

  Algorithms like *minimax* or *alpha-beta* [27] are designed to explore sequential games where a player tries to find the movement that maximizes an evaluation function while minimum values of this function favor the opponent. Alpha-beta incorporates a pruning condition that improves the search speed with the same result of minimax.

  Unlike sequential games, in stochastic games there is no guarantee that the game evolution will follow the search exploration carried out by the player. One possible solution is to apply statistical approximations along several iterations of the game. One example of combinatorial search algorithms is *Monte Carlo Tree Search* (MTCS) [13], which analyzes the most promising moves, expanding the search tree based on random sampling of the search space from a selected leaf node of the tree. In the selection process, better nodes are more likely to be chosen.

- **Learning-based controllers**

  This type of control strategy consists in improving the behaviour of a game agent by using previously recorded examples of the game, or interacting directly with the game. The agent is trained by playing multiple games and/or observing professional matches, and iteratively penalizing or rewarding the agent actions in order to obtain better outcomes [44].

  The state of the art in learning-based controllers is full of variations of neural networks (NN onwards), specially deep convolutional neural networks. In games like *Chess* or *Go*, NN are used to evaluate a concrete intermediate state of the game and predict the final outcome. In addition, NN are used to predict the best action to apply in the current state of the game. The combination of search strategies like MTCS with NN is very helpful to conduct search simulation.

  One more advantage of learning-based controllers over combinatorial search controllers is that its not necessary to run a simulation process to find the best action.

Instead, as a result of the training process, we get a model embedded in the controller that acts as a function, returning the action to execute in every state.

A different approach that generally obtains good results are Genetic algorithms. They are stochastic, parallel search algorithms based on the mechanics of natural selection and evolution. Genetic algorithms were designed to efficiently search large, non-linear, poorly-understood search spaces where expert knowledge is scarce or difficult to encode and where traditional optimization techniques fail [14]. The development of a game agent based on genetic algorithms requires first to decide the game parameters to be optimized and then encode these parameters into a chromosome layout. Subsequently, an initial population with sufficiently different chromosomes to guarantee genetic diversity is created. Iteratively, the behaviour of each individual of the population in the game is simulated and a new generation is created, crossing some individuals which are chosen following a selection strategy. The idea is to select the fittest individuals and let them pass their genes to the next generation. In order to avoid the tendency towards local optimization, a mutation component is introduced, where there is a probability of mutation, and some genes are modified according to the mutation strategy.

- **Hybrid Approaches:**

  Hybrid approaches emerge as a combination of some of the aforementioned described techniques. The *Dynamic scripting* [38] game is an example of the utilization of an hybrid approach, where hand-coded strategies are combined with learning-based algorithms. Under this hybrid approach, the agent is guided by a set of rules but the decision of which rule to choose is made by means of a reinforcement learning process. *Dynamic scripting* executes the agent multiple times, adding a reward or penalty proportional to how well the agent performed in a particular level of the game.

2.3 Planning and learning for game playing

Research in videogames aims at developing two types of game agents:

- Agents oriented to play a single game; e.g., AlphaGo [4] which was the first AI program that won versus a professional championship of *Go*; or Stockfish [2], which won the 14^{th} Top Chess Engine Championship of *Chess*; or competitions like the Mario AI championship [1] shown in Figure 2.2.

- Agents able to play several games and to adapt themselves to new environments. The implementation of agents for playing different games is encouraged by platforms such as the Arcade Learning Environment or the General Video Game AI competition (GVG-AI).

The goal of agents that play a single game is to find a behaviour that allows them to master such game, trying to maximize the score or minimize the time it takes to solve the game. To this end, researchers implement techniques like the ones explained in section 2.2 and build specialized agents for a specific games. The result is an agent that excels in the game in question, but is unable to adapt to the conditions of a different game and would most likely get poor results [50].

However, some researchers are interested in developing general game agents capable of playing different games, even games for which the agent has never been trained for.

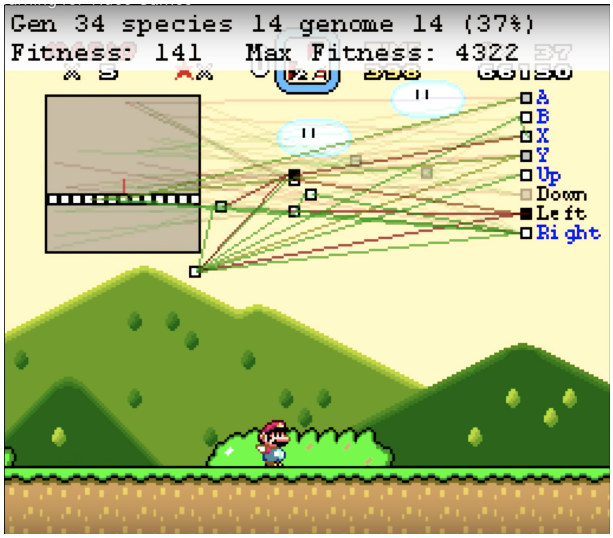


Figure 2.2: Genetic-based controller in Mario AI competition

In this case, the final objective is to test agents in general game competitions and to develop AI techniques that can possibly be later applied to some real-life tasks. Ultimately, researchers aim to develop general AI techniques applicable to different games instead of a specialized tool for only one game.

Following, we are going to discuss briefly three of the most important agents in AI for games: AlphaGo, AlphaZero and AlphaStar.

2.3.1. AlphaGo

AlphaGo is a computer program developed by Google DeepMind that plays the board game *Go* based on a combination of deep neural networks and Monte Carlo tree search and is capable of playing at the level of the strongest human players. AlphaGo uses Monte Carlo Tree Search algorithm to find its moves based on knowledge previously learned by a combination of supervised and reinforcement learning by an artificial neural network. A neural network is trained to predict AlphaGo's own move selections and also the winner's games. This neural network improves the strength of tree search, resulting in higher quality of move selection and stronger self-play in the next iteration. [43]

In October 2015, AlphaGo became the first computer *Go* program to beat a human professional *Go* player without handicaps on a full-sized 19x19 board. In March 2016, it beat Lee Sedol in a five-game match, the first time a computer Go program has beaten a 9-dan professional without handicaps. The next version, AlphaGo Zero, was trained only by self-play, and surpassed all the previous versions.

2.3.2. AlphaZero

AlphaZero is a generalized variant capable of accommodating a broader class of game rules than the previous AlphaGo Zero, a version of AlphaGo created without using data from human games, and stronger than any previous version of AlphaGo. The parameters of the deep neural network in AlphaZero are trained by reinforcement learning entirely from self-play games, unlike it predecessor AlphaGo . Each game is played by running an MCTS search from the current position, and then selecting a move, either proportionally (for exploration) or greedily (for exploitation) with respect to the visit counts at the root state. At the end of the game, the terminal position is scored according to the rules of

the game to compute the game outcome z: -1 for a loss, 0 for a draw, and +1 for a win. The neural network parameters are updated to minimize the error between the predicted outcome and the game outcome, and to maximize the similarity of the policy vector to the search probabilities.[45]

The main differences between AlphaZero and AlphaGo Zero are:

- The neural network of AlphaZero is updated continually, instead of waiting for an iteration to be completed like AlphaGo Zero.
- *Go* (unlike *Chess*) is symmetric under certain reflections and rotations; AlphaGo Zero was programmed to take advantage of these symmetries, while AlphaZero is not, so it can accommodate a broader class of games.
- AlphaGo Zero estimated and optimized the probability of winning, exploiting the fact that *Go* games have a binary win or loss outcome. However, both *Chess* and *Shogi* may end in drawn outcomes. AlphaZero instead estimates and optimizes the expected outcome.

Within 24 hours of training, AlphaZero achieved superhuman level of play in the games of Chess, Shogi and Go by defeating world-champion programs, Stockfish, Elmo, and the 3-day version of AlphaGo Zero, as shown in Figure 2.3.

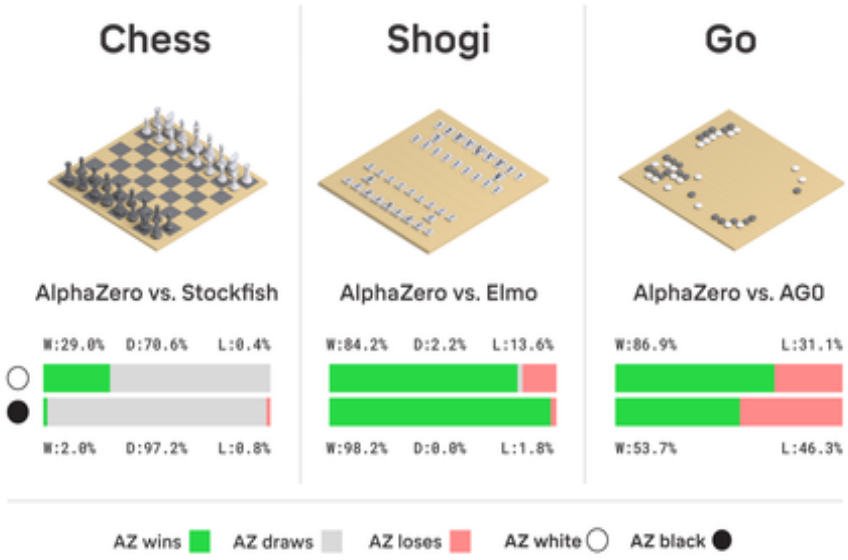


Figure 2.3: Results of AlphaZero in *Chess*, *Go* and *Shogi*

2.3.3. AlphaStar

AlphaStar is the first artificial intelligence system to beat a professional player at the game of Starcraft II. Starcraft and its sequel Starcraft II belong to the RTS genre (real-time strategy), a genre of games that is particularly challenging due to its characteristics: real-time play, partial observability, no single dominant strategy, complex rules and a large and varied action space. A capture of this game is shown below, in Figure 2.4.

AlphaStar is trained initially using imitation learning to mimic human play, and then improved through the use of population-based training to keep a population of agents that trains against each other. Its a memetic algorithm that uses Lamarckian evolution:

the neural networks of the agents are optimised through backpropagation in an inner loop, while in the outer loop networks are picked using one of several selection methods (such as binary tournament selection), with the winner's hyperparameters overwriting the loser's, and applying a mutation on those. [5]

Figure 2.4: AlphaStar vs Grzegorz "MaNa" Komincz from Team Liquid

CHAPTER 3
Background

In this chapter we will detail the necessary concepts to follow the contributions of this project. First, we introduce the Arcade Learning Environment, followed by the OpenAI Gym, which is the environment we use in this project.

In section 3.2 we introduce some basic concepts on classical planning as well as a novel search algorithm that is used for exploration in our proposal.

Next, we will explain the most relevant concepts about Reinforcement Learning as well as the selected algorithm to implement our AI agent.

3.1 Arcade Learning Environment

The Arcade Learning Environment (ALE) is a simple object-oriented framework that enables the development of AI agents for Atari 2600 games. ALE is built on top of Stella, an open-source Atari 2600 emulator which allows the user to interface with the Atari 2600 by receiving joystick motions, sending screen and/or RAM information, and emulating the platform. The Atari 2600 was originally released in 1977 and there were published more than 500 games, spanning a diverse range of genres such as shooters, beat'em ups, puzzle, etc... Some of these games are timeless classics such as *Breakout*, *Pong*, *Space Invaders* or *Kung-Fu Master*.

ALE provides a game-handling layer which transforms each game into a standard reinforcement learning problem by identifying the accumulated score and whether the game has ended. An episode begins on the first frame after a reset command is issued, and terminates when the game ends. The game-handling layer also offers the ability to end the episode after a predefined number of frames. By default, each observation consists of a single game screen (frame): a 2D array of 7-bit pixels (128 colours), 160 pixels wide by 210 pixels high.

The action space consists of up to the 18 discrete actions defined by the joystick controller: three positions of the joystick for each axis and a single button. Some of the games on the Atari have a smaller action space, for example, *Space Invaders* has an action space of 6: *Noop* (do nothing), *Fire* (shoot without moving), *Right* (move right), *Left* (move left), *RightFire* (shoot and move right) and *LeftFire* (shoot and move left).

When ALE is running in real-time, the simulator generates 60 frames per second, and is capable of emulating up to 6000 frames per second. The reward at each time-step is defined on a game by game basis, typically by taking the difference in score or points between frames.

Figure 3.1: Photo of the Atari 2600

ALE allows us to access several dozen games through a single common interface, and adding support for new games is relatively straightforward. It also provides the functionality to save and restore the state of the emulator. When issued a save-state command, ALE saves all the relevant data about the current game, including the contents of the RAM, registers, and address counters. The restore-state command similarly resets the game to a previously saved state. This enables the use of ALE as a generative model to study topics such as planning and model-based reinforcement learning [7]. Due to this characteristics, ALE is an ideal testbed for evaluating and comparing agents.

3.1.1. OpenAI Gym

OpenAI Gym is a toolkit for developing and comparing reinforcement learning algorithms written in *Python*. It aims to combine the best elements of previous benchmark collections to become a software package that is convenient and accessible.

OpenAI was created with the intention of removing the problem of lack of standardization in papers along with the aim to create better benchmarks by giving versatile numbers of environment with great ease of setting up. The aim of this tool is to increase reproducibility in the field of AI research and provide tools with which everyone can learn about basics of AI [10].

OpenAI Gym focuses on the episodic setting of reinforcement learning, where the agent's experience is broken down into a series of episodes. In each episode, the agent's initial state is randomly sampled from a distribution, and the interaction proceeds until the environment reaches a terminal state. The goal in episodic reinforcement learning is to maximize the expectation of total reward per episode, and to achieve a high level of performance in as few episodes as possible.

The main functions that a Gym Environment provides are:

- **Step()**: The Step function receives an action, then the environment runs one timestep and returns a tuple composed of:
  - *Observation*: an environment-specific object representing your observation of the environment. For example, pixel data from a camera, joint angles and joint velocities of a robot, or the board state in a board game.

  - *Reward*: the amount of reward returned after the previous action was executed
  - *Done*: a boolean which indicates whether the episode has ended or not. If an episode has ended, further calls of step() will return undefined results.
  - *Info*: a dictionary with useful information for diagnostic and debugging. The information contained in info is specific of the environment, and sometimes, there is useful information that can be used to play the environment.

- **Reset()**: This function resets the environment to its initial state and returns the initial observation of the environment. To initiate and episode of training, we must call this function.

- **Render()**: Renders the environment and returns the observation. There are different modes of rendering, and the modes supported vary depending of the environment, existing environments which don't support any kind of rendering. The most common are:

  - *Human*: With this mode, the environment is rendered to the current display or terminal and nothing is returned to the agent. Its done to view the agent interacting with the environment.
  - *Rgb_array*: Returns an array with shape (x, y, 3), which represents the RGB values for an x-by-y pixel image.
  - *Ansi*: Returns a string containing a terminal-style text representation. The text can include newlines and ANSI escape sequences (e.g. for colours).

- **Close()**: Method used to stop and close the environment, freeing up all the physics' state. Environments will automatically close() themselves when garbage collected or when the program exits.

- **Seed()**: Sets the seed for this environment's random number generator. If the environment uses more than one random number generator, then this method accepts a list of numbers. It returns the list of seeds used by the environment's random number generator.

The environment also provides useful information in these attributes:

- **Action_space**: Contains actions that are valid in the environment.

- **Observation_space**: The shape of the observations in number of pixels.

- **Reward_range**: A tuple corresponding to the min and max possible rewards.

Some environments provide more attributes and or functions. For example, the Atari games in Gym, which work using the ALE to run, provide a function called *get_action_meanings()* to known the meanings of each action in the environment. They also provide function to clone and restore the game state, allowing the use of search algorithm in these games. The observations in these environments can be images or the contents of the Ram, but the environment only supports the rendering of the game observations in *human* mode and in *rgb_array*.

Gym also serves as an interface with a wide variety of environments such as 2D and 3D robots simulations, using the physics engine *MuJoCo*[49] or *Roboschool*, another 3D robot simulation which uses the Bullet engine instead of *MuJoCo*. The classic shooter game *Doom* can also be used as an environment with Gym using *ViZDoom*, a *"Doom based AI Research Platform for Reinforcement Learning from Raw Visual Information"* [26].

3.2 Planning with IW algorithms

Automated planning in Artificial Intelligence (AI) is defined as the art of building control algorithms for dynamic systems. More precisely, a planning task is a search problem whose purpose is finding a set of actions that leads the system to an objective state from a given initial situation. The vast majority of approaches model planning as a single-agent procedure, in which a single entity or agent carries out the entire search process, developing the complete course of action to solve the task at hand.

In this section, we first introduce the principal elements of a planning task. Subsequently, we present a novel pruning criteria for tree search and its usage in a popular search algorithm for solving planning problems. Finally, we summarize some results obtained with such algorithm in classical planning.

3.2.1. Planning task

Single-agent planning is a search process in which starting from an initial situation, the agent has to find a plan or course of actions that allows it to reach a final state that includes the goals to achieve. Classical planning adopts a series of assumptions to reduce the complexity of the problem and define its components more easily.

- The world is represented through a finite set of situations or states.
- The world is fully observable. In other words, the single agent has complete knowledge of the environment.
- The world is deterministic; that is, the application of an action can only generate a single other state.
- The world is static, the state of the world does not evolve until an action is applied.
- The planner handles explicit and immutable goal states.
- The planning process is carried out offline, so a planner does not consider external changes that occur in the world.

A state is represented by a set of instantiated state variables named literals. The literals reflect those characteristics of the world that are interesting for the task at hand. The states of the world change through the application of the planning actions. Actions define the conditions that must hold in the world for an action to be applicable and the effects that result from the application of the action. Conditions are statements quering the value of a variables and effects are statements assigning a value to a variable.

We will denote the set of *fluents* (propositional state variables) describing a state as F. A *literal* or instantiated state variable l is a valuation of a fluent $f \in F$, i.e. either $l = f$ or $l = \neg f$. A set of literals L represents a partial assignment of values to fluents. We use $\mathcal{L}(F)$ to denote the set of all literal sets on F; i.e. all partial assignments of values to fluents.

A *state* s is a full assignment of values to fluents; i.e. $|s| = |F|$, so the size of the state space is $2^{|F|}$.

A *classical planning frame* is a tuple $\Phi = \langle F, A \rangle$, where F is a set of fluents and A is a set of actions. An action $a \in A$ is defined with *preconditions*, $\mathsf{pre}(a) \subseteq \mathcal{L}(F)$, *positive effects*, $\mathsf{eff}^+(a) \subseteq \mathcal{L}(F)$, and *negative effects* $\mathsf{eff}^-(a) \subseteq \mathcal{L}(F)$. We say that an action $a \in A$ is *applicable* in a state s iff $\mathsf{pre}(a) \subseteq s$. The result of applying action a in state s is the

successor state $\theta(s,a) = \{s \setminus \text{eff}_c^-(s,a)) \cup \text{eff}_c^+(s,a)\}$ where $\text{eff}_c^-(s,a) \subseteq \textit{triggered}(s,a)$ and $\text{eff}_c^+(s,a) \subseteq \textit{triggered}(s,a)$ are, respectively, the triggered *negative* and *positive* effects.

A *classical planning problem* is a tuple $P = \langle F, A, I, G\rangle$, where I is an initial state and $G \subseteq \mathcal{L}(F)$ is a goal condition. A *plan* for P is an action sequence $\pi = \langle a_1, \ldots, a_n\rangle$ that induces the *state trajectory* $\langle s_0, s_1, \ldots, s_n\rangle$ such that $s_0 = I$ and a_i $(1 \leq i \leq n)$ is applicable in s_{i-1} and generates the successor state $s_i = \theta(s_{i-1}, a_i)$. The *plan length* is denoted with $|\pi| = n$. A plan π *solves* P iff $G \subseteq s_n$; i.e. if the goal condition is satisfied in the last state resulting from the application of the plan π in the initial state I.

An important aspect is how to represent the components of a planning task with a compact and expressive language. One of the first planning languages is STRIPS (STanford Research Institute Problem Solver) [15], which has influenced most of the existing planners. STRIPS is a compact and simple language that allows the specification of planning domains and problems. Despite its advantages, STRIPS has some limitations that make it difficult to describe some real problems. As a result, many extensions have been developed over the past years, enriching its expressiveness and simplifying the definition of planning domains. One of these extensions is Planning Domain Definition Language (PDDL) [16], the standard language used in the International Planning Competitions (IPC) within the planning community.

When designing a PDDL problem, we need to define two separate blocks. On the one hand we must define the **domain** that includes the rules that govern the world of the problem. On the other hand we must define the **problem**, that entails to expose a particular situation within the domain previously described. Into the problem we must specify the initial state, as well as the objectives to be solved. The domain describes the general features of a particular domain, such as the types of objects, the predicates that describe situations of the world and the operators that can be applied by the planning entity to solve the task. The problem block models the specific details of the task, such as the actual objects in the world, the initial situation of the task and the goals that must be achieved in order to solve the planning task.

An example of PDDL code can be seen in the figure 3.2 where an action of the game *Sokoban* is defined within the domain block.

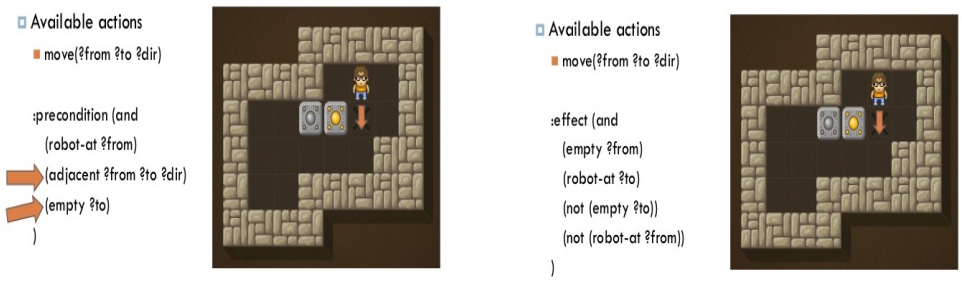


Figure 3.2: Action 'move' in the game *Sokoban*

State space planning

The simplest classical planning algorithms are state-space search algorithms. These are search algorithms in which the search space is a subset of the state space: each node corresponds to a state of the world, each arc corresponds to an action or state transition, and the current plan corresponds to the current path in the search space. Particularly, a

solution plan is the sequence of actions of a path that leads from the initial situation I represented in the root node of the search space to a node (state) that contains the goal condition G.

Most state-space planners use *forward-search* algorithms, starting the construction of the plan in the initial state I and moving forward using the available actions until a final state that contains G is reached. Forward-search is complete but the search space is usually much larger than it needs to be. There are various ways to reduce the size of the search space, by modifying the algorithm to prune branches of the search space (i.e., cut off search below these branches).

Forward-state planners usually apply heuristics to guide the search. A heuristic function classifies states according to their desirability and the next state is selected according to this ranking. Some of the most relevant state-space planners are:

- The **Heuristic Search Planner (HSP)** is one of the first state-based systems which uses domain-independent heuristic search. The additive heuristic of HSP is defined as the sum of costs of the individual goals in G, where the cost of a single atom is estimated by considering a relaxed planning task in which all delete lists of the actions are ignored.

- The **Fast Forward (FF)**planning system is one of the most influential approaches to state-based planning. It uses the relaxed plan heuristic h_{FF} , which is defined as the number of actions of a plan that solves the relaxed planning task. FF works with a Enforced Hill Climbing search, that is searching exhaustively nodes with a better heuristic value.

- **Fast Downward (FD)** is a heuristic-based planner that uses a multi-valued representation for the planning tasks. FD use SAS+ [22] to model the facts that conform states. Each variable has associated a Domain Transition Graph (DTG). This structure reflects the evolution of that variable according to the actions of the task. DTGs are used to compile the Causal Graph in which are reflected the dependencies between different state variables. FD sue a best-first multi-heuristic search alternating h_{FF} and h_{CG} a heuristic inferred of the Causal Graph

- **LAMA** satisficing planner apply landmarks to improve the accuracy of the heuristic search. A landmark is a fact that holds at some point in every solution of a planning task. LAMA is based in FD planning but reuses the multi-heuristic search strategy of FD to alternate a landmark-based estimator and a variant of the h_{FF} heuristic.

3.2.2. Width

Some recent investigations have looked into the so called *width-based search*. *Width* is a parameter that bounds the complexity of classical planning problems and domains. When width is used along with a simple but effective blind-search procedure, planning runs in time that is exponential in the problem width. Unlike the classical notion of reachability of states, the width parameter comes up instead as a different reachability relation over tuples (conjunctions) of literals of bounded size [31].

Roughly speaking, the width of a goal formula G composed of a single literal is the minimum number of literals w required to be present simultaneously in a state s so that every optimal plan that reaches w in s can be extended by appending an optimal action sequence that reaches G. If G is true in s then the width of G is 0; i.e., $w(G) = 0$. Intuitively, the width indicates the indispensable number of literals that are needed to generate another literal. For instance, let be the following example from the *Blocksworld* do-

main. Given $s = \{on(A,B), on(B,C), clear(A), ontable(C)\}$, and $G = ontable(B)$, we can affirm $w(G) = 1$ as stated in the path $clear(A), hold(A), ontable(A), hold(B), ontable(B)$. That is, the optimal plans for $hold(A)$ can always be extended with the action $putdown(A)$ into optimal plans for $ontable(A)$, while the optimal plans for $ontable(A)$ from the above situation can all be extended with the action $unstack(B,C)$ into optimal plans for $hold(B)$. Likewise, the optimal plans for $hold(B)$ extended with the action $putdown(B)$ lead to an optimal plan for $ontable(B)$. This is all saying that in order to reach $ontable(B)$ we just require $clear(A)$ to optimally generate $hold(A)$, and $hold(A)$ to optimally generate $ontable(A)$, and subsequently $hold(B)$ is achievable because $hold(A)$ implies $clear(B)$. Therefore, only one atom (literal) is required to successively extend the optimal plans for $hold(A)$ and obtain $ontable(B)$.

The above example is an illustration to show that $w(G) = 1$ for any goal $G = ontable(b)$ in the *Blocksworld* domain. It turns out indeed that for single atom goals, the width of domains like *Blocksworld*, *Logistics* and *n-puzzle* is at most 2 [31].

3.2.3. Iterated Width Search

The idea of *search for novelty* is first introduced in the work [29] as a search technique that ignores the objective of the search and searches for behavioral novelty. Specifically, a *novelty search algorithm* searches with no objective other than continually finding novel behaviors in the search space. Yet because many points in the search space collapse to the same point in behavior space, it turns out that the search for novelty is computationally feasible.

Using the concept of *width* explained above as a pruning criterion in a search for novelty has given rise to the *Iterated-Width* (IW) algorithm, a novelty-based pruned breadth-first search (BFS) that uses a set of **atoms** (i.e., pairs of state variables with their corresponding associated value) to represent a state and prunes states that do not satisfy a given **novelty** condition. IW(i) is an i-width search that is complete for problems whose width is bounded by i and has complexity is $O(n^i)$, where n is the number of problem variables. Globally speaking, IW(i), where i is the value of width, is a plain forward-state BFS with just one change: right after a state s is generated, the state is pruned if it does not pass a simple novelty test that depends on i [31, 32].

In a IW algorithm, a state is composed of a set of **state variables**.

$$V = \{v_1, v_2, \ldots, v_N\}$$

Each state variable $v_j \in V$ has a finite and discrete **domain** D_{v_j} that defines the possible values of that variable. A **state** is a total assignment of values to the state variables

The **IW(i) algorithm** is an implementation of a standard BFS, starting from a given initial state s_0, that prunes any state that is considered not *novel*, where the novelty condition is defined as follows:

Definition 3.2.1. State novelty. When a new state s is generated, IW(i) contemplates all n-tuples of atoms of s with size $n \leq i$. The state is considered *novel* if at least one tuple has not previously appeared in the search, otherwise the state is pruned.

Assuming that the N state variables have the same domain D, IW(i) visits at most $O((N \times |D|)^i)$ states. A key property of the algorithm is that while the number of states is exponential in the number of atoms, IW(i) runs in time that is exponential in only in i. In particular, IW(1) is linear in the number of atoms, while IW(2) is quadratic. IW(i)

is then a blind search algorithm that eventually traverses the entire state-space provided that i is large enough.

To understand the algorithm, let us present as example a simple search task. The objective of the *counters problem* is defined as finding a given number with a predefined number of counters. It is a simple search task that allows us to explain the different concepts of the IW algorithms and illustrate the potential of these algorithms.

In the *counters problem* the state variables are integers numbers whose domain is $[0 \dots 9]$ each representing the value of a counter. For example, if we have 3 counters, we will need 3 variables x_1, x_2 and x_3. The initial state is by convention the situation where all counters are fixed to 0, in our example, $x_1 = 0$, $x_2 = 0$, $x_3 = 0$. The goal or final state will be to reach a predetermined number. In our example, the goal will be $x_1 = 3$, $x_2 = 3$, $x_3 = 3$. To transit between different states we define a function per counter that increments the counter value in one.

$$f_1 \rightarrow x_1 \mathrel{+}= 1 \ , \ f_2 \rightarrow x_2 \mathrel{+}= 1 \ , \ f_3 \rightarrow x_3 \mathrel{+}= 1$$

In IW(i) atoms are a subset of variables of size i with an specific value; for example in the initial state of our example task, there are three true atoms of size 1 $\{(x_1 = 0), (x_2 = 0), (x_3 = 0)\}$. For IW(2) there are three atoms of size 2, $\{(x_1 = 0, x_2 = 0), (x_1 = 0, x_3 = 0)(x_2 = 0, x_3 = 0)\}$ and so on. The number of possible atoms increases exponentially in function of i, for IW(1) there are 30 possible atoms whereas for IW(2) the number raises to 300 and for IW(3) reaches 1000 atoms.

Figure 3.3 shows the trace of the IW(1) in our *counters problem* example. *Nodes in red* are pruned because they do not satisfy the novelty condition. That is, other nodes of the tree previously discovered the atoms that appear in the pruned node. For example, the first red node has three atoms $x_0 = 1$, $x_1 = 1$ and $x_2 = 0$; the first atom of this node appears in the parent node, the second atom appears in the second node of previous level, and the last atom comes from the root node. Hence, the node $\{x_0 = 1, x_1 = 1, x_2 = 0\}$ does not provide any new atom and does not satisfy the novelty condition for IW(1). *Green nodes* represent atoms that satisfy an individual goal (we remind that the goal of the problem in our example is to reach $\{x_0 = 3, x_1 = 3, x_2 = 3\}$). We can observe that IW(1) always finds individuals goals in an optimal path.

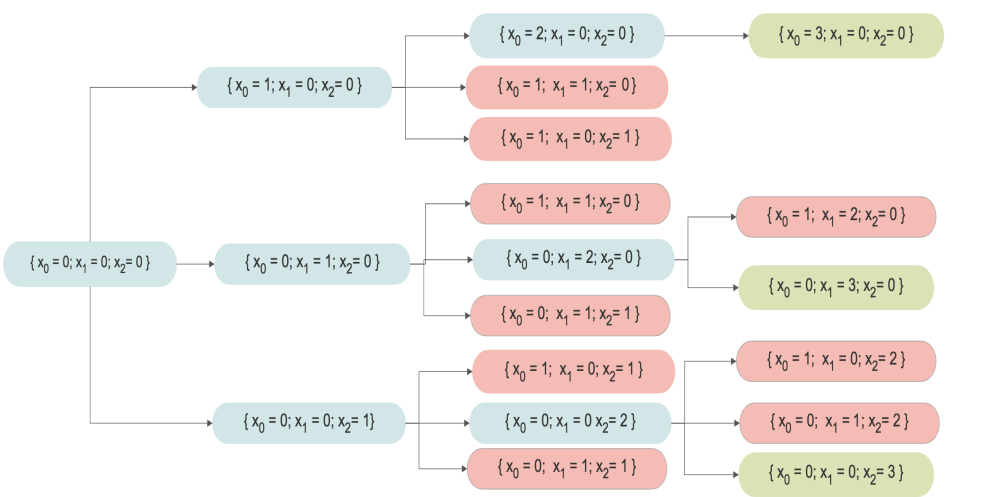


Figure 3.3: Trace of IW(1) to the *counters problem*.

Since we are not able to solve a problem with IW(1), we would increment the width (number of parameters) and come up with more complex goals. IW(2) will account for two simultaneous goals at the cost of augmenting the number of produced nodes. And IW(3) would solve the complete problem achieving the goal of the problem; i.e., the number 333.

The same problem can be solved by running IW(1) until one objective is achieved, and then starting the algorithm again in that state to find the subsequent objectives (because in this particular problem goals are serializable). The first execution allows the IW(1) algorithm to reach the goal $x_0 = 3$ as in the Figure 3.3. Starting from the first green node, the second goal will be reached in the second iteration. And the last iteration will eventually achieve the global goal. With this serial implementation of the IW(1) algorithm, we can achieve the given three objectives in one same execution. This is possible provided that the objectives of the problem are independent. That is, when the achievement of one objective does not interfere in the accomplishment of another objective. In the case that the objectives are dependent to each other, it will be needed to increase the width level of the algorithm to achieve them. For instance, in the game *Adventure* game, it is necessary to first grasp a key in order to open a door or grasp a sword to beat an enemy (the avatar can only hold an object at a time). With IW(1) we will be able to reach the key or the door, but we may not achieve the two objectives. However, if we try first to reach the key, we will succeed in the second objective, opening the door.

To summarize, by using IW(i), we can obtain as many objectives as the value of the width equal to the i parameter. On the other hand, with a serialized implementation of IW we can reduce the i parameter to achieve independent objectives.

3.2.4. IW-based planning

Width-based search algorithms were developed in the setting of classical planning to show that instances of many existing domains can be solved in low polynomial time when they feature atomic goals.

The ideas developed in width-based search have also been used to yield state-of-art results in classical planning over the standard benchmarks of the IPC [31, 32], and more recently in the Atari games [35, 42], and those of the General Video-Game AI competition [17].

Width-based methods have also been used to avoid plateaus in heuristic search. In contrast to goal-oriented search (exploitation), width-based search (seeking novel states) is a form of structural exploration of the search space. The combination of both techniques have proven to yield a search scheme, best-first width search, that is better than both and which results in classical planning algorithms that outperform the state-of-the-art planners [34].

More recently, the usage of best-first width search in the context of (decentralised) multi-agent privacy-preserving planning has been addressed in the work [19]. In particular, authors show that best-first width search is a very effective approach over several benchmark domains, even when the search is driven by heuristics that roughly estimate the distance from goal states, computed without using the private information of other agents.

3.3 Reinforcement Learning

Reinforcement learning (RL) studies action selection, in an *unknown* environment, with the aim of maximizing some notion of cumulative reward. In RL the environment is considered *unknown* because typically the action model and the reward function of the environment are both unknown.

Reinforcement learning is one of the three basic Machine Learning paradigms, alongside *supervised learning* and *unsupervised learning*. RL differs from *supervised learning* in that RL agents collect the learning examples by themselves, through interaction with the environment. On the other hand RL differs from *unsupervised learning* in that learning examples are labeled with a reward value.

With this regard, RL agents try to find a balance between *exploration* and *exploitation* (useful to maximize the cumulative reward): Where *exploration* refers to the discovery of new knowledge about the environment, and *exploitation* refers to leveraging the collected knowledge of the environment.

RL problems are typically formulated as optimization problems within a *Markov Decision Process* (MDP). A MDP can be formalized with the following four elements:

- S, a set of states (that are typically factored using a set of state variables).
- A, a set of actions. Every action $a \in A$ is applicable at every state $s \in S$.
- $P_a(s, s') = Prob(s_{t+1} = s' | s_t = s, a_t = a)$, the probability of transition from state s to state s' under the execution of action a at the time step t.
- $R_a(s, s')$ is the immediate reward obtained after transition from s to s' with the execution of action a.

In RL problems both $P_a(s, s')$ and $R_a(s, s')$ are initially unknown, despite they can be sampled and estimated by interacting with the environment (i.e. by executing actions and observing the obtained outcomes). In more detail, at a time step t, the RL agent observes the current state s_t, executes action a_t that transits to the next state s_{t+1}, and obtains a new reward value $r_{t+1} = R_{a_t}(s, s_{t+1})$.

The goal of a RL agent is to maximize its total (future) reward. This potential reward is a weighted sum of the expected values of the rewards of all future steps starting from the current state. Usually the goal metric to maximize is formulated as a discounted function of the accumulated reward: $\sum \gamma^t r_t$, where γ is the *discount factor* (a constant between 0 and 1) that determines the relative value of delayed versus immediate rewards and r_t is the obtained reward value at time step t.

Solutions to a RL problem are typically specified as an action selection *policy* that specifies the most promising action to apply at each reachable state. Thus, the decision strategy is represented by a policy $\pi : S \rightarrow \Delta(A)$; i.e. a mapping from states to probability distributions over actions.

Following we review one of the most popular approaches for solving RL problems.

3.3.1. Q-learning

Q-learning is a RL algorithm that computes a $Q : S \times A \rightarrow R$ function to represent and update the *quality* of the state-action combinations. $Q(s, a)$ stands then for the *quality* of an action a taken in a given state s.

Before learning begins, the $Q(s,a)$ function is initialized to a possibly arbitrary fixed value (e.g., chosen by the programmer). Then, at each time step t the RL agent selects an action a_t, enters a new state s_{t+1}, observes a reward r_t, and updates accordingly the $Q(s,a)$ function. The core of the *Q-learning* algorithm is the update of the $Q(s,a)$ function, a simple value iteration update, using the weighted average of the old value and the new information (α is *the learning rate* and it is a constant between 0 and 1 that balances the weight of the old and new values):

$$Q(s_t,a_t) := (1-\alpha)Q(s_t,a_t) + \alpha(r_t + \gamma max_a Q(s_{t+1},a))$$

An episode of the *Q-learning* algorithm ends when state s_{t+1} is a terminal state. For all terminal states s_f its $Q(s_f,a)$ value is never updated, but instead is set to the reward value r_f observed for that state. In most cases, $Q(s_f,a)$ can be taken to equal zero.

If the discount factor is lower than 1, *Q-learning* algorithm guarantees that the action values are finite, even if the problem can contain infinite loops. For any finite Markov Decision Process, *Q-learning* guarantees also to find a policy that is *optimal* in the sense that it maximizes the expected value of the total reward over any and all successive steps, starting from the current state.

Q-learning can also be extend by using a *Deep neural network* to represent and update the $Q(s,a)$ value function of a RL agent. This approach is known as *Deep Q-learning* and makes it possible to apply the RL framework to problems with huge state-spaces (and even continuous state-spaces). In addition the use of *neural network* to represent and update the $Q(s,a)$ value function speed up learning, due to the fact that they can generalize earlier experiences to previously unseen states.

3.4 Deep Learning

Deep learning is a family of *Machine Learning* algorithms that are based on *neural networks*, and that leverage multiple layers of *artificial neurons* to progressively extract higher level features from the raw input data.

The word "deep" refers then to the number of layers through which the data is transformed. Each layer learns to transform its input data into a slightly more abstract and composite representation. For example imagine a video-game playing application, where the raw input is a matrix with the values of the screen pixels. In this case lower layers may abstract the pixels and identify edges, while higher layers may identify more relevant concepts for the video-game playing, such as the kind and position of objects in the screen.

Despite the impressive performance of *deep learning* techniques at a wide range of diverse real-world application, a main criticism concerning *deep learning* algorithms is the high number of parameters that affects the final performance of this kind of algorithms. For example, varying the numbers of layers, layer sizes and structure provide different degrees of abstraction and hence, dramatically different performances. In addition *deep learning* methods are considered *black box* methods since the validation of their performance is too often empirical rather than theoretical.

Most modern deep learning models are based on artificial neural networks, specifically, *Convolutional Neural Networks*. Next we review this particular kind of artificial neural networks.

3.4.1. Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are regularized versions of *multilayer perceptrons* that are commonly applied to analyzing visual imagery.

Multilayer perceptrons usually mean fully connected networks, that is, each artificial neuron in one layer is connected to all neurons in the next layer. The strong connectivity of these networks makes them prone to overfitting data. Typical ways of regularization include adding some form of magnitude measurement of weights to the loss function. However, CNNs take a different approach towards regularization: they take advantage of the hierarchical pattern in data and assemble more complex patterns using smaller and simpler patterns. Therefore, on the scale of connectivity and complexity, CNNs are on the lower extreme.

CNNs leverage *convolution*, in place of general matrix multiplication, in at least one of their layer. Further their activation function is commonly a *RELU layer*, and is subsequently followed by additional convolutions such as pooling layers, fully connected layers and normalization layers, referred to as *hidden layers* because their inputs and outputs are masked by the activation function and final convolution. The final convolution, in turn, often involves backpropagation in order to more accurately weight the end product.

CNNs use relatively little pre-processing compared to other image classification algorithms. This means that the network learns the filters that in traditional algorithms were hand-engineered. This independence from prior knowledge and human effort in feature design is a major advantage.

Convolutional networks were inspired by biological processes in that the connectivity pattern between neurons resembles the organization of the animal visual cortex. Individual cortical neurons respond to stimuli only in a restricted region of the visual field known as the receptive field. The receptive fields of different neurons partially overlap such that they cover the entire visual field.

CHAPTER 4

Integrating Planning and Learning for the Atari Video-games

AI planning algorithms, such as the ones in the IW family [33], do not leverage any kind of past information. This means that the performance of planners in a given problem (or domain) does not improve over time, despite the planner addresses similar or even the same planning problems.

In this chapter we show how to integrate Machine Learning techniques with the IW planning algorithms to exploit past data of previous planning episodes and improve the performance of the planning process as more experience is available. The integration we follow is based on the recent *Deep Reinforcement Learning* (DRL) framework proposed by Junyent et al. 2019 [24].

The chapter finalizes with the presentation of two enhacements to achieve more consistent planning and learning processes in the framework of the presented baseline integration.

4.1 A Deep Reinforcement Learning framework for the ALE

Reinforcement Learning (RL) frameworks traditionally capture the knowledge learned from past experience as an *action selection policy* [47]. An action selection policy is a function that maps (state, action) pairs into a numeric value. In stochastic environments this value represents the *expected reward* that can be achieved by taking the given action at the given state. The higher this value, the more promising the application of the action in its associated state.

In interesting real-world problems the state space is usually too large to explicitly represent the policy and store the value for each possible (state,action) pair. Instead, a policy function can be *learned*, which does not store but *estimates* the value of actions with regard to the given input state. Even more, a set of *state features* can be used to abstract further the actual value of the state variables and achieve more compact representations of the policy function. Related to this, the recent approach of *Deep Reinforcement Learning* (DRL) builds on top of *Deep Neural Networks* (DNNs) to effectively reduce the dimensionality of input images. Thanks to DNNs the Deep Reinforcement Learning systems can compactly represent action selection policy functions despite inputs states are screenshot images [36].

In the Atari video-games the state variables represent the values of the screen pixels. The input images are defined by a pixel array 160 wide and 210 high, with pixels that may have up to 128 colors. In this project we follow the DRL approach and use a Deep Neural

Network to compactly represent the policy estimate that maps pairs of (screen,action) into their corresponding value estimate.

In a RL framework the policy is used in turn:

1. to guide the planning algorithm, preferring promising paths according to the policy, and
2. to represent the knowledge learned from past experience (the policy generalizes past planning episodes)

Next we will detail how the planning and learning processes are implemented in the integration followed by this project.

4.1.1. Softmax action selection policy

The *softmax policy* consists of a softmax function that converts output to a distribution of probabilities and it is commonly used in RL to obtain a probability for each possible action. Although greedy action selection is an effective and popular means of balancing exploration and exploitation in reinforcement learning, one drawback is that when it explores it chooses equally among all actions. This means that it is as likely to choose the worst-appearing action as it is to choose the next-to-best action. In tasks where the worst actions are very bad, this may be unsatisfactory. The obvious solution is to vary the action probabilities as a graded function of estimated value. The greedy action is still given the highest selection probability, but all the others are ranked and weighted according to their value estimates. These are called softmax action selection rules. The most common softmax method uses a Gibbs, or Boltzmann, distribution. It chooses action a on the t^{th} play with probability:

$$Q(s_t, a_t) = \pi(a_t|s_t) = \frac{e^{Q(s_t,a_t)/\tau}}{\sum_{k \in A} e^{Q(s_t,a_k)/\tau}}$$

where τ is a positive parameter called the *temperature*. High temperatures ($\tau \to \infty$) cause the actions to be all (nearly) equiprobable. Low temperatures cause a greater difference in selection probability for actions that differ in their value estimates (the probability of the action with the highest expected reward tends to 1). In the limit as $\tau \to 0$, softmax action selection becomes the same as greedy action selection. In short, high values of τ favor exploration whilst low values of τ favor exploitation.

4.1.2. Guiding IW-based planning algorithms with softmax policy

The planning algorithm used in this project is the Rollout version of IW(1) [6]. The reason to use this version is that it is closer to the *Reinforcement Learning* setting since Rollout-IW(1) can deal with trajectories, as most RL methods do, with the restriction that a simulator needs to be resetable to a previous state.

In more detail, the Rollout-IW(1) planning algorithm explores at its termination the same set of states than the original IW(1) but replaces the breadth-first construction of IW(1) by a sequence of depth-progressing rollouts. This mechanism provides better anytime behavior than the original IW(1) algorithm and requires less from the simulator: While tree search algorithms like IW(1) need the ability of expanding nodes, i.e., the application of all actions to a node, the rollouts of a Rollout-IW(1) algorithm apply just

one action per node, like the family of the Monte-Carlo tree search algorithms that are traditionally used in RL.

The implementation of the Rollout-IW(1) version requires:

- A *novelty table* that keeps track of the minimum depth at which a feature is found for the first time.
- An extension of the notion of novelty. A state is now considered novel in any of these two situations: (a) a feature is found for the first time or (b) a feature appeared before but at a deeper level in the search tree.

To guide the Rollout-IW(1) planning with an action selection policy such as the softmax policy, the Rollout-IW(1) algorithm is modified as follows. Rollout-IW(1) needs to sample actions to build the trajectories that replace the breadth-first construction of IW(1). Instead of sampling actions with uniform probability, Rollout-IW(1) enables to leverage the action selection policy to bias this sampling towards the actions with a higher value (according to the learned softmax policy) and hence it prefers rollouts with associated higher policy values.

The *temperature* parameter of the softmax policy (see equation in section 4.1.1) is also added to the sampling mechanism to control the exploration/exploitation trade-off of the planning episodes. When the value of the temperature tends to infinite, we have a pure exploration setting that corresponds to the original Rollout-IW(1) algorithm (which ignores the learned policy performing a blind exploration). When temperature equals to 0 sampling is fully biased by the learned softmax policy.

4.1.3. Learning the policy estimation from past planning episodes

A planning episode of the Rollout-IW(1) algorithm terminates in a number of rollouts that is bounded by $|F|^2 \times b$, where F is the number of state features, and b is the branching factor (maximum number of actions that can be applied at a give node).

Once the planning algorithm terminates, the score values obtained in the terminal states (including the states that are pruned by absence of novelty) are back-propagated to their parent states according to the following expression:

$$R_i = r_i + \gamma max_{j \in children(i)} R_j$$

With this regard, we have a single learning example for each pair (state, action) that corresponds to an explored transition and such that the label of each example is the corresponding back-propagated score.

The deep neural network that represents the policy estimation is then a non-linear regression of the collected labeled examples and it is trained supervised by sampling a batch of transitions (a maximum of T transitions are kept discarding outdated transitions in a FIFO manner).

4.1.4. Abstracting states with Deep Neural Networks

Rollout IW(1) is guaranteed to reach every 1-width goal in time that is polynomial in the number of state features. In addition to this, symbolic state features can make planning more effective when the width of a problem is effectively reduced by the information encoded in these features. The selection of the right set of state-features is then key for

the performance of width-based algorithms however the learning of informative state features for general/arbitrary planning problems is still an open challenge.

Here we show that learned action selection policies in the form of a Deep Neural Network can also be seen as a state *autoencoder* for the input screen-shots [24]. This autoencoder reduces the dimensionality of the state space while it extracts relevant state features that are useful for the planning algorithm. In this project, as in the (DRL) framework proposed by Junyent et al. 2019, we take the last hidden layer of the Deep Neural Network that encodes the policy as the set of state features to reduce the dimensionality of the screen pixels. In particular we use the output of the rectified linear units, that we subsequently discretize in the simplest way, resulting in binary features (0 for zero outputs and 1 for positive outputs).

Besides this mechanism for dynamically extracting informative state features from the input images, planning algorithms can also leverage static mappings that effectively reduces the dimensionality of the state space. An example is the *basic PROST* features defined by Liang et al. for ALE [30] that are also used in the experimental section of this project. These features map screen pixels into a set of visual features by splitting the Atari screen into 16×14 disjoint tiles, each comprised of 10×15 pixel patch and track whether the input image contains a pixel of a given colour in the tile(i,j).

4.2 Improving the RL framework for the ALE

In this section, we first we present a refinement over the Rollout version of IW(1) that produces a larger exploration of the state space. The aim of this refinement of the planning procedure is reaching states with higher associated rewards that allow us to lead to better action selection policies.

Then we also show that *Deep Neural Networks* can be tuned to produce richer sets of state-features for the presented integration scheme of planning and learning.

4.2.1. Improving the planning algorithm

The number of states that are not pruned in IW(1) is $O(n)$. In some situations (especially in problems with a high *width* value) this number may be too small to produce states with a sufficiently high reward value.

One approach to alleviate the exploration limitation of the IW(1) algorithm is to run instead IW(k), with increasing values of the k parameter. However, the implementation of the novelty check of the IW(k) algorithm for values of k that are greater than one results in a too expensive search for huge state-spaces (like it happens in the Atari video-games). In particular the *novelty table* becomes easily too large to be stored in memory.

Our proposal is to implement a new version of the IW algorithm called IW(3/2), a version halfway between IW(1) and IW(2) where some *informative* atoms are handpicked to build 2-atom tuples. This intermediate approach IW(3/2) has already shown effective in video-game playing [18].

For implementing IW(3/2) we decided to include the atoms chosen for the basic version IW(1) plus some variable (feature) with which to form two-atom tuples. This extra information of the environment can be the score, the position of the avatar, its health, its type, if it is holding an object, etc. While most of these variables are specific to each game and so they are environment-dependent features, we can though use as extra feature the *score* of a game, which is present in all games and it is available in every OpenAI Gym environment tested in this project.

Thus, IW(3/2) will prune a node when the pair (feature,score) is not novel. In other words, the IW(3/2) version considers a novelty state if it provides a new value of the 'feature' or the state has a score equal or higher than the one in the *novelty table*. With this modification, we reduce the number of potentially pruned states, which in turn widens the search space and allows IW(3/2) to achieve more complex goals than IW(1). All these benefits come along without the asymptotic complexity of a full IW(2) that considers two-atom tuples for every state-feature.

The number of times that the score changes during the game will have an impact on the development of the search algorithm. That is to say, a game in which the score is not updated until the victory, will not be affected by the modification of the IW(3/2) algorithm, whereas a game that updates the score continuously will delay the pruning to a greater extent, contemplating many nodes that until now were not considered.

In environments where the increases in the reward reflect that the agent is approaching the goal, IW(3/2) should perform better because more nodes will be considered in the search. Alternatively, in environments with sparse rewards IW(3/2) should perform like IW(1) because both will prune roughly the same nodes.

With this regard, we implemented a Rollout version of the IW(3/2) algorithm that, at its termination, explores the same set of states than the original IW(3/2) but replaces its breadth-first construction by a sequence of depth-progressing rollouts.

4.2.2. Improving the learning algorithm

In this section we present the modifications undertaken in the Neural Network (NN) so as to come up with a more informative feature space and a better guiding policy.

The performance of the planning step, like the performance of learning algorithms, is sensible to the choice of features. Features that capture meaningful structure normally yield better results than raw features that do not.

To obtain a better performance, we have modified the architecture of the NN and we have fine-tuned some of the hyperparameters based on empirical tests. To do so, we increased the number of neurons in the last hidden layer from 256 to 1024 to increase the number of features that will be used by the IW algorithm. Increasing the number of units to more than 1024 decreased the performance of the game agent in our tests.

Subsequently, we added another fully-connected layer of 1024 units and activation function *ReLU* after the convolutions and added another convolution layer with 64 filters, a *kernel* size of 3x3, *stride* 1 and activation function *ReLU*. We also doubled the number of filters of the other two convolutions. The final architecture is shown in Figure 4.1.

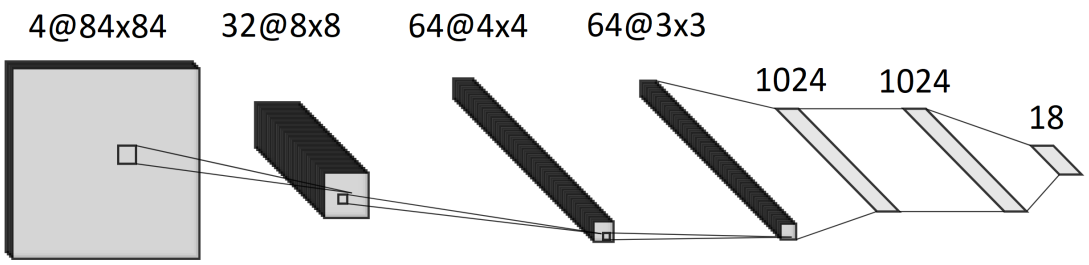


Figure 4.1: Architecture of the Neural Network

Regarding the changes in the hyperparameters, we increased the *batch size* from 32 to 128 and we increased accordingly the *learning rate* from 0.0005 to 0.001 due to the relation

between the *learning rate* to *batch size* ratio and the performance of the agent. Using a larger batch size leads to a speedup in training [20] at the cost of performance, but this can be mitigated by increasing the learning rate accordingly [46]. The change in the batch size also reduced the standard deviation of the scores obtained by the agent. We also increased the decay of *RMSProp* from 0.99 to 0.975 because we obtained better results empirically. We also tried to fine-tune other hyperparameters, but none of the changes lead to a better performance.

The enhancements in the NN will also help obtain a better guiding policy. The new hyperparameters of the NN (θ) are combined with the state features to produce a distribution over actions following the softmax policy explained in section 4.1.1. It is common to use an NN to represent the policy estimate ($\widehat{\pi_\theta}$) and define:

$$\widehat{\pi_\theta}(a|s) = \frac{e^{h_a(s,\theta)/\tau}}{\sum_{k \in A} e^{h_k(s,\theta)/\tau}}$$

as the softmax combination of the NN outputs $h_a(s,\theta)$, $a \in A$ (reward obtained when applying action a in state s under the parameters θ), where τ is the temperature that controls exploration.

CHAPTER 5
Empirical evaluation

In this chapter we will present the games from the ALE that we have chosen for the empirical evaluation. Next, we will present the results obtained in our experiments with the two planning & learning versions proposed in this chapter, the IW(3/2) algorithm and the IW(3/2) with an enhanced NN, as well as a through analysis of the strengths and weaknesses of our version versus IW(1).

5.1 Game selection

We have chosen 17 games from the ALE to compare the performance of our game agents against the IW(1) algorithm. The games can be roughly classified by the genre they belong to and by their mechanics in the following way:

- **Maze games**: Games which it's gameplay revolves around their "world" design, where the players has to navigate the environment and plan their movements to reach the goal. There is usually obstacles, enemies and or trap which makes beating the game difficult. The games we include in this category are: *Alien*, *Ms. Pac-man*, *Q*bert*, *Tutankham* and *Venture*.
- **Shooter games**: In games that belong to this genre players use ranged weapons to participate in the action, which takes place at a distance, and must defeat the enemies and evade the shots of the enemies. In this category we include: *Assault*, *BattleZone*, *Centipede*, *Demon Attack*, *James Bond 007* and *Space Invaders*.
- **Reactive games**: We include in this category games which emphasize physical challenges that require hand-eye coordination and motor skill to overcome. They are centered around the player, who is in control of most of the action. The games that belong to this category are: *Asterix*, *Kung-Fu Master*, *Pong*, *Road Runner*, *Skiing* and *Tennis*.

We include a resume, an explanation of the gameplay and a capture of each game.

5.1.1. Maze games

- **Alien** is a maze game for the Atari 2600 published by *20th Century Fox* in 1982, based on the 1979 film with the same name.

  The player controls a member of the human crew pursued by three aliens in the hallways of a ship. The goal is to destroy the alien eggs laid in the hallways (like

the dots in Pac-Man). The player is armed with a flamethrower which can temporarily stun the aliens. Additionally, "pulsars" (like the power pills in Pac-Man) occasionally appear, and serve to turn the tables on the aliens, allowing the player to overpower them.

Figure 5.1: Capture of the Atari 2600 game *Alien*

- **Ms. Pac-man** is a maze arcade game developed by *General Computer Corporation* and published by *Midway Games*. It is the sequel to *Pac-Man* (1980), and the first entry in the series to not be made by *Namco*.

  The gameplay of Ms. Pac-Man is very similar to that of the original Pac-Man. The player earns points by eating pellets and avoiding monsters. The contact with one causes Ms. Pac-Man to lose a life. Eating an energizer (or "power pellet") causes the monsters to turn blue, allowing them to be eaten for extra points. Bonus fruits can be eaten for increasing point values, twice per round. As the rounds increase, the speed increases, and energizers generally lessen the duration of the monsters' vulnerability, eventually stopping altogether.

  There are also some differences from the original *Pac-Man*:

  The game has four different mazes that appear in different colour schemes, and alternate after each of the game's intermissions are seen. The pink maze appears in levels 1 and 2, the light blue maze appears in levels 3, 4, and 5, the brown maze appears in levels 6 through 9, and the dark blue maze appears in levels 10 through 14. After level 14, the maze configurations alternate every 4th level. Three of the four mazes (the first, second, and fourth ones) have two sets of warp tunnels, as opposed to only one in the original maze. The walls have a solid color rather than an outline, which makes it easier for a novice player to see where the paths around the mazes are. The monsters' behavioral patterns are different, and include semi-random movement, which prevents the use of patterns to clear each round. Blinky (red) and Pinky (pink) move randomly in the first several seconds of each level, until the first reversal. Inky (cyan) and Sue (orange) still use the same movement patterns from the previous game to their respective corners, again until the first reversal. Instead of appearing in the center of the maze, the fruits bounce randomly around the maze, entering and (if not eaten) leaving through the warp tunnels. Once all fruits have been encountered, they appear in random sequence for the rest of the game, starting on the eighth round.

  The Atari 2600 rendition of *Pac-Man* was infamous for its flashing ghosts. This is because there are only two sprite reset registers, and used one for the player's sprite and one for the ghosts, but they couldn't render everything on the same frame, so the ghost flicker because in each frame only one is being drawn. This flicker can also be seen in *Ms. Pac-man*, although much less, due to the reuse of the register to

draw the same sprite with a different position and colour, as long as its not on the same horizontal line. When this happens, we can see the flickering on *Ms. Pac-man* because its switches to the other technique to render the screen.

Figure 5.2: Capture of the Atari 2600 game *Ms. Pac-man*

- **Q*bert** is an arcade game developed and published for the North American market by *Gottlieb* in 1982.

  It's an action game with puzzle elements played from an axonometric third-person perspective to convey a three-dimensional look. The game is played using a single, diagonally mounted four-way joystick. The player controls *Q*bert*, who starts each game at the top of a pyramid made of 28 cubes, and moves by hopping diagonally from cube to cube. Landing on a cube causes it to change colour, and changing every cube to the target colour allows the player to progress to the next stage.

  At the beginning, jumping on every cube once is enough to advance. In later stages, each cube must be hit twice to reach the target colour. Other times, cubes change colour every time *Q*bert* lands on them, instead of remaining on the target colour once they reach it. Both elements are combined in subsequent stages. Jumping off the pyramid results in the character's death.

  The player is impeded by several enemies, introduced gradually to the game:

  - *Coily*: It first appears as a purple egg that bounces to the bottom of the pyramid and then transforms into a snake that chases after *Q*bert*.
  - *Ugg* and *Wrongway*: Two purple creatures that hop along the sides of the cubes. Starting at either the bottom left or bottom right corner, they keep moving toward the top right or top left side of the pyramid respectively, and fall off the pyramid when they reach the end.
  - *Slick* and *Sam*: Two green creatures that descend down the pyramid and revert cubes whose colour has already been changed.

  A collision with purple enemies is fatal to the character, whereas the green enemies are removed from the board upon contact. Coloured balls occasionally appear at the second row of cubes and bounce downward; contact with a red ball is lethal to *Q*bert*, while contact with a green one immobilizes the on-screen enemies for a limited time. Multicoloured floating discs on either side of the pyramid serve as an escape from danger, particularly *Coily*. When *Q*bert* jumps on a disc, it transports him to the top of the pyramid. If *Coily* is in close pursuit of the character, he will jump after Q*bert and fall to his death, awarding bonus points. This causes all enemies and balls on the screen to disappear, though they start to return after a few seconds.

25 points are awarded for each colour change, defeating *Coily* with a flying disc awards 500 points, the remaining multicoloured discs at the end of a stage also award points on higher stages (50 and later 100) and catching green balls (100) or Slick and Sam (300 each). Bonus points are also awarded for completing a screen, starting at 1,000 for the first screen of Level 1 and increasing by 250 for each subsequent completion, up to 5,000 after Level 4. One extra live is granted after completing the first five stages, and then one extra for every four stages completed thereafter.

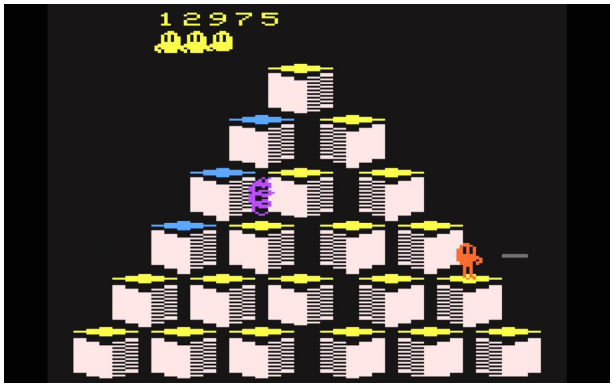


Figure 5.3: Capture of the Atari 2600 game *Q*bert*

- **Tutankham** is a 1982 maze Shooter developed by *Konami*.

  Taking on the role of an explorer grave robbing the maze-like tomb of Tutankhamun, the player is chased by asps, vultures, parrots, bats, dragons, and curses, all of which kill the explorer on contact. The explorer wields a laser weapon that only fires left and right, as well as a single screen-clearing "flash bomb" per level or life. Warp zones teleport the player to another location in the level, which enemies cannot use.

  To progress, the player collects keys to open locked doors throughout each level and well as optional treasures for bonus points. When a timer reaches zero the explorer can no longer shoot. Passing through the large exit door ends the level, and any remaining time is converted to bonus points.

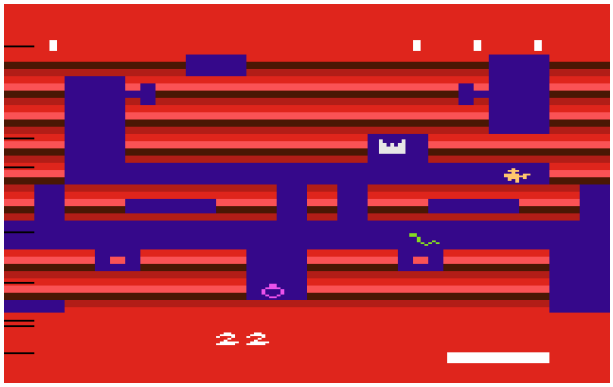


Figure 5.4: Capture of the Atari 2600 game *Tutankham*

- **Venture** is a 1981 fantasy-themed arcade game by *Exidy*. The goal of Venture is to collect treasure from a dungeon as a round smiley-face named *Winky*.

  Winky is equipped with a bow and arrow and explores a dungeon with rooms and hallways. The hallways are patrolled by large, tentacled monsters named *Hallmonsters*, which cannot be killed, injured, or stopped in any way. Once in a room,

Winky may kill monsters, avoid traps and gather treasures. If he stays in any room too long, a *Hallmonster* will enter the room, chase and kill him. The more quickly the player finishes each level, the higher their score.

The goal of each room is only to steal the room's treasure. In most rooms, it is possible to steal the treasure without defeating the monsters within. Some rooms have traps that are only sprung when the player picks up the treasure.

Winky dies if he touches a monster or *Hallmonster*. Dead monsters decay over time and their corpses may block room exits, delaying *Winky* and possibly allowing the *Hallmonster* to enter. Shooting a corpse causes it to regress back to its initial death phase. The monsters themselves move in specific patterns but may deviate to chase the player, and the game's AI allows them to dodge the player's shots with varying degrees of "intelligence".

The game consists of three different dungeon levels with different rooms. After clearing all the rooms in a level the player advances to the next. After three levels the room pattern and monsters repeat, but at a higher speed and with a different set of treasures.

The different dungeons in each level are as follows:

- Level 1: The Wall Room, The Serpent Room, The Skeleton Room, The Goblin Room
- Level 2: The Two-Headed Room, The Dragon Room, The Spider Room, The Troll Room
- Level 3: The Genie Room, The Demon Room, The Cyclops Room, The Bat Room

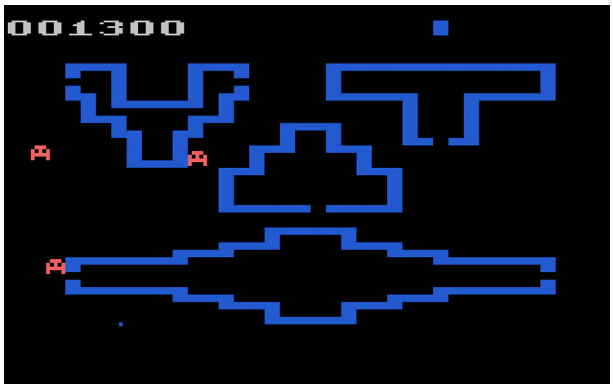


Figure 5.5: Capture of the Atari 2600 game *Venture*

5.1.2. Shooter games

- **Assault** is a 1983 fixed Shooter video game developed and published by *Bomb*.

  The player must fight an alien mother ship, which continually deploys three smaller ships during play. The mother ship and the smaller vessels shoot at a weapon the player is in command of, and the player's aim is to eliminate the opposition while preventing the weapon from receiving enough damage to destroy it.

- **Battle Zone** is a first-person Shooter tank combat arcade game from *Atari, Inc.* released in November 1980. The player controls a tank which is attacked by other tanks and missiles.

Figure 5.6: Capture of the Atari 2600 game *Assault*

The gameplay occurs on a flat plane with a mountainous horizon featuring an erupting volcano, a distant crescent moon, and various geometric solids like pyramids and blocks. The player can hide behind the solids or, once fired upon, maneuver in rapid turns to buy time with which to fire again. The geometric solid obstacles are indestructible, and can block the movement of the player's tank. They can also be use as shields as they block enemy fire as well.

The player views the screen, which includes an overhead radar view, to find and destroy the slow tanks, or the faster-moving supertanks. Saucer-shaped UFOs and guided missiles occasionally appear for a bonus score. The saucers differ from the tanks in that they do not fire upon the player and do not appear on the radar.

Figure 5.7: Capture of the Atari 2600 game *Battle Zone*

- **Centipede** is a vertically oriented fixed Shooter arcade game produced by *Atari, Inc.* in June 1981. The player fights off centipedes, spiders, scorpions and fleas, completing a round after eliminating the centipede that winds down the playing field.

  The player's fighter is represented by a small insect-like head at the bottom of the screen. The player moves it around the bottom area of the screen and fires small darts at a segmented centipede advancing from the top of the screen down through a field of mushrooms. Each segment of the centipede becomes a mushroom when shot; shooting one of the middle segments splits the centipede into two pieces at that point. Each piece then continues independently on its way down the screen, with the rear piece sprouting its own head. If the head is destroyed, the segment behind it becomes the next head. Shooting the head is worth 100 points while the other segments are 10. The centipede starts at the top of the screen, travelling either left or right. When it touches a mushroom or reached the edge of the screen, it

descends one level and reverses direction. The player can destroy mushrooms by shooting them, but each takes four shots to destroy. At higher levels, the screen can become increasingly crowded with mushrooms due to player/enemy actions, causing the centipede to descend more rapidly.

Once the centipede reaches the bottom of the screen, it moves back and forth within the player area and one-segment centipedes will periodically appear from the side. This continues until the player has eliminated both the original centipede and all heads. When all the centipede's segments are destroyed, another one enters from the top of the screen. The initial centipede is 10 or 12 segments long, including the head; each successive centipede is one segment shorter and accompanied by one detached, faster-moving head. This pattern continues until all segments are separate heads, after which it repeats with a single full-length centipede.

The player also encounters other creatures besides the centipedes. Fleas drop vertically and disappear upon touching the bottom of the screen, occasionally leaving a trail of mushrooms in their path when only a few mushrooms are in the player movement area; they are worth 200 points and takes two shots to destroy. Spiders move across the player area in a zig-zag pattern and eat some of the mushrooms; they are worth 300, 600, or 900 points depending on how close the player shoots it. Scorpions move horizontally across the screen, turning every mushroom they touch into poisonous mushrooms. Scorpions are also worth the most points of all enemies with 1,000 points each. A centipede touching a poisonous mushroom will change colour and hurtle straight down toward the bottom, then return to normal behaviour upon reaching it. This "poisoned" centipede can be very challenging to avoid if it appears as multiple separated segments.

The fighter will be destroyed when hit by any enemy, after which any poisonous or partially damaged mushrooms revert to normal. 5 points are awarded for each regenerated mushroom. The player gains extra lives every 12,000 points.

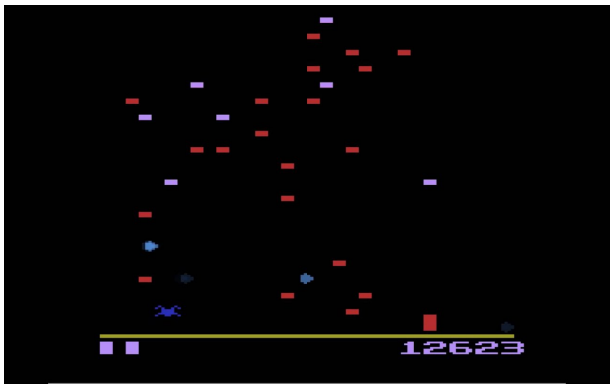


Figure 5.8: Capture of the Atari 2600 game *Centipede*

- **Demon Attack** is a fixed Shooter published by *Imagic* in 1982. Marooned on the ice planet Krybor, the player uses a laser cannon to destroy legions of demons that attack from above. Visually, the demons appear in waves similar to other space-themed Shooters, but individually combine from the sides of the screen to the area above the player's cannon.

  Each wave introduces new weapons with which the demons attack, such as long streaming lasers and laser clusters. Starting in Wave 5, demons also divide into two smaller, bird-like creatures that eventually attempt descent onto the player's cannon. Starting in Wave 9, the demons' shots follow directly beneath the monsters, making it difficult for the player to slip underneath to get in a direct shot.

Figure 5.9: Capture of the Atari 2600 game *Demon Attack*

- **James Bond 007** is a 1983 side-scrolling video game developed and published by *Parker Brothers*.

  The player controls the titular character of James Bond across four levels. The player is given a multi-purpose vehicle that acts as an automobile, a plane, and a submarine. The vehicle can fire shots and flare bombs, and travels from left to right as the player progresses through each level. The player can shoot or avoid enemies and obstacles that appear throughout the game, including boats, frogmen, helicopters, missiles, and mini-submarines.

  The game's four levels are loosely based on missions from various James Bond films:

  – Diamonds are Forever (1971): The player rescues Tiffany Case from an oil rig.
  – The Spy Who Loved Me (1977): The player destroys an underwater laboratory.
  – Moonraker (1979): The player destroys satellites.
  – For Your Eyes Only (1981): The player retrieves radio equipment from a sunken boat.

Figure 5.10: Capture of the Atari 2600 game *James Bond 007*

- **Space Invaders** is a 1978 arcade game created by Tomohiro Nishikado. Within the Shooter genre, *Space Invaders* was the first fixed Shooter and set the template for the shoot 'em up genre.

  The player controls a laser cannon by moving it horizontally across the bottom of the screen and firing at descending aliens. The aim is to defeat five rows of eleven aliens (six rows of six aliens in the Atari 2600 version) that move horizontally back and forth across the screen as they advance toward the bottom of the screen. The

player's laser cannon is partially protected by several stationary defense bunkers (three in the Atari 2600 version) that are gradually destroyed from the top and bottom by blasts from either the aliens or the player.

The player defeats an alien and earns points by shooting it with the laser cannon. As more aliens are defeated, the aliens' movement and the game's music both speed up. Defeating all the aliens on-screen brings another wave that is more difficult, a loop which can continue endlessly. A special "mystery ship" will occasionally move across the top of the screen and award bonus points if destroyed.

The aliens attempt to destroy the player's cannon by firing at it while they approach the bottom of the screen. If they reach the bottom, the alien invasion is declared successful and the game ends tragically; otherwise, it ends generally if the player's last cannon is destroyed by the enemy's projectiles.

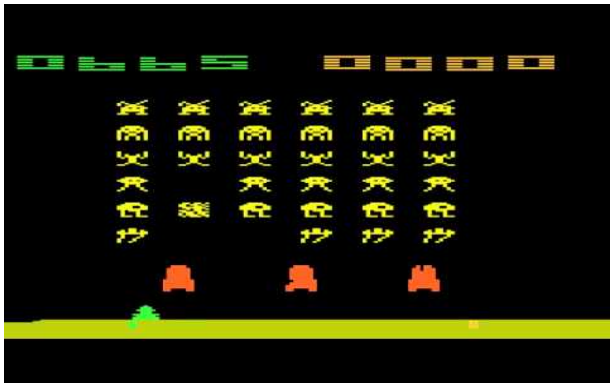


Figure 5.11: Capture of the Atari 2600 game *Space Invaders*

5.1.3. Reactive games

- **Asterix** (Published in North America under the name of *Taz*), is a video game developed and released by *Atari, Inc.* in 1983. Within the game, *Asterix* only appears as a crude sprite slightly resembling his head.

  The player must move between 8 lanes to pick up various treasures like magic cauldrons and avoid the deadly lyres of *Assurancetourix*. The game loops infinitely making the goal of the game simply to acquire the highest score without dying. The lyres speed up as the player gains more points, making it a challenging battle of reflexes and luck to get the highest score. As the game progresses, the magic cauldrons gets replaced by other sprites, like helmets of Roman soldiers, and the reward for picking up the items is increased every time the sprite is changed. A capture of the can be seen in Figure 5.12.

- **Kung-Fu Master** is a side-scrolling beat 'em up game produced by *Irem* as arcade game in 1984 and distributed by *Data East* in North America. The players control Thomas, the titular *Kung-Fu Master*, as he fights his way through the five levels of the *Devil's Temple* in order to rescue his girlfriend Sylvia from the mysterious crime boss Mr. X. *Kung-Fu Master* is regarded as the first beat 'em up video game.

  The player controls Thomas with a four-way joystick and two attack buttons to punch and kick. Unlike more conventional side-scrolling games, the joystick is used not only to crouch, but also to jump. Punches and kicks can be performed from a standing, crouching or jumping position. Punches award more points than kicks and do more damage, but their range is shorter.

Figure 5.12: Capture of the Atari 2600 game *Asterix*

Underlings encountered by the player include *Grippers*, who can grab Thomas and drain his energy until shaken off; *Knife Throwers*, who can throw at two different heights and must be hit twice; and *Tom Toms*, short fighters who can either grab Thomas or somersault to strike his head when he is crouching. On even-numbered floors, the player must also deal with falling balls and pots, snakes, poisonous moths, fire-breathing dragons, and exploding confetti balls.

The Devil's Temple has five floors, each ending with a different boss. In order to complete a floor, Thomas must connect with enough strikes to completely drain the boss's energy meter. Then the player can climb the stairs to the next floor. Thomas has a fixed time limit to complete each floor; if time runs out or his meter is completely drained, the player loses one life and must replay the entire floor. Upon completing a floor, the player receives bonus points for remaining time and energy. The boss of the fifth floor is Mr. X, the leader of the gang that kidnapped Sylvia. Once he is defeated, Thomas rescues Sylvia and the game restarts at a higher difficulty level.

Figure 5.13: Capture of the Atari 2600 game *Kung-Fu Master*

- **Pong** is one of the earliest arcade video games. It is a table tennis sports game featuring simple two-dimensional graphics. The game was originally manufactured by *Atari*, which released it in 1972. *Pong* was the first commercially successful video game, which helped to establish the video game industry along with the first home console, the *Magnavox Odyssey*.

  Pong is a two-dimensional sports game that simulates table tennis. The player controls an in-game paddle by moving it vertically across the left or right side of the screen. They can compete against another player controlling a second paddle on the opposing side. Players use the paddles to hit a ball back and forth. The goal is

for each player to reach twenty one points before the opponent; points are earned when one fails to return the ball to the other.

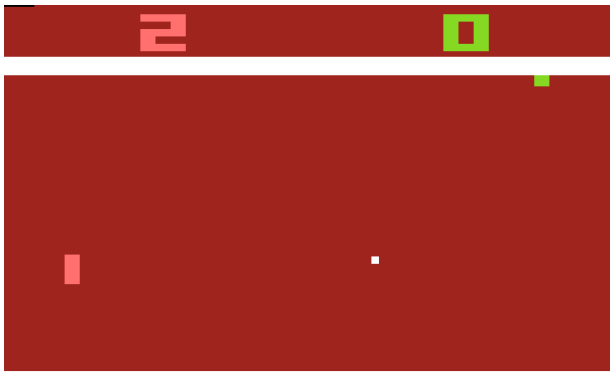


Figure 5.14: Capture of the Atari 2600 game *Pong*

- **Road Runner** is a racing game based on the *Wile E. Coyote* and Road Runner shorts. It was released by Atari Games in 1985.

  The player controls *Road Runner*, who is chased by *Wile E. Coyote*. In order to escape, Road Runner runs endlessly to the left. While avoiding *Wile E. Coyote*, the player must pick up bird seeds on the street (100 points, every next we pick up increases its value by 100, up to 1000 if none are missed, else the value of the next one will be 100) and steel shots (100 points), avoid obstacles like trucks, mines, cannonballs or rocks. Sometimes *Wile E. Coyote* will just run after the Road Runner, but he occasionally uses tools like rockets, roller skates, and pogo-sticks. Getting Wile E. Coyote hit by a mine, cannonball, rock, etc... will increase the score by 100 points, and 1000 if a truck hits him. Picking up steel shots makes Wile E. Coyote to use a magnet to catch us, increasing its speed and forcing us to evade him until we can force him to stop using a mine, a truck, etc...

  The Atari 2600 port was one of Atari Corporation's last games for the system, being released in 1989.

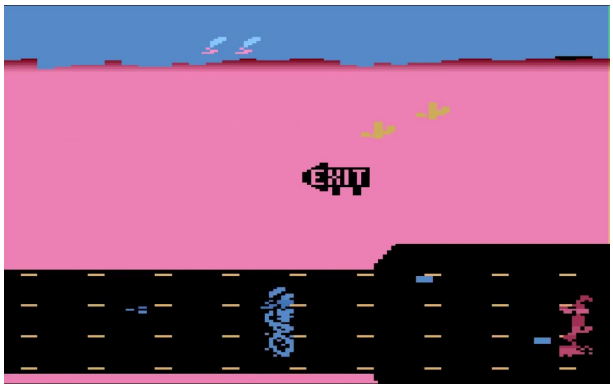


Figure 5.15: Capture of the Atari 2600 game *Road Runner*

- **Skiing** is a video game for the Atari 2600 authored by Bob Whitehead and released by *Activision* in 1980.

  Skiing is a single player only game, in which the player uses the joystick to control the direction and speed of a stationary skier at the top of the screen, while the background graphics scroll upwards, thus giving the illusion the skier is moving. The player must avoid obstacles, such as trees and moguls. The game cartridge contains five variations each of two principal modes:

- *Downhill*: the goal of the player is to reach the bottom of the ski course as rapidly as possible, while a timer records his relative success.
- *Slalom*: the player must similarly reach the end of the course as rapidly as he can, but must at the same time pass through a series of gates (indicated by a pair of closely spaced flagpoles). Each gate missed counts as a penalty against the player's time.

Figure 5.16: Capture of the Atari 2600 game *Skiing*

- **Tennis** is a video game for the Atari 2600 which was published by *Activision* in 1981.

  The game offers singles matches for one or two players; one player is colored pink, the other blue. The game has two user-selectable speed levels. When serving and returning shots, the tennis players automatically swing forehand or backhand as the situation demands, and all shots automatically clear the net and land in bounds.

  The first player to win one six-game set is declared the winner of the match (if the set ends in a 6-6 tie, the set restarts from 0-0). This differs from professional tennis, in which player must win at least two out of three six-game sets.

Figure 5.17: Capture of the Atari 2600 game *Tennis*

5.2 Experimental results

In this section we present the results in terms of game score obtained by different implementations of IW-based game agents.

Table 5.1 shows the scores obtained by the following approximations:

1. The first column corresponds to the scores obtained by IW(1) using a predefined pixel-based representation of states named B-PROST [6]. B-PROST is made of the sum of three types of features:

   - *Basic* consists of splitting the screen pixels into disjoint tiles, and for each tile and each colour, there is a boolean value which is true if the colour c appears in the tile *(i,j)*.
   - *B-PROS* (Basic Pairwise Relative Offsets in Space features) tracks the relative distances among pairs of basic features in the same screen. There is a feature for each colour c in tile t and colour c' in tile t', for all possible combinations of colours and different tiles.
   - *B-PROT* (Basic Pairwise Relative Offsets in Time features) represents pairwise relative offsets between basic features obtained from the screen at two different time points. There is a feature for each colour c in tile t in the previous screen and colour c' in tile t' in the current screen, for all combinations of tiles and colours.

2. The second column of Table 5.1 corresponds to the rollout version of IW(1), where the search is depth-first instead of breadth-first. It also uses B-PROST for the state representation.

3. The third column, π-IW, is a rollout IW(1) which uses a CNN as an auto-encoder for the feature extraction, as commented in Section 4. The NN is also used to learn a game policy and to guide the state/action selection in the planning phase.

4. The fourth column, π-IW(3/2), corresponds to our first enhancement over the version of the third column. That is, using the score of the game to lessen the pruning of states during the search process of the IW(3/2) algorithm.

5. The last column, π-IW(3/2)-NN, corresponds to both of our enhancements, the new IW(3/2) version of the width-based algorithm plus the changes made to the NN.

IW (first column) and Rollout IW (second column) have a budget of 0.5s for planning. The values shown in the last two columns correspond to the average score of the last ten episodes played by the game agent. Due to time constraints, we decided to omit the version that includes only the NN enhancement in the π-IW algorithm. Likewise, due to the excessive training time of the algorithm, some of the scores of π-IW(3/2)-NN are not included either.

Following, we proceed to delve into the results obtained with each of the approximation presented in Table 5.1. The explanation is organized following the game classification presented in section 5.1. Table 5.1 is split in three parts, each group of rows representing one game category. The first group composed of the first five rows are the maze games; the second group corresponds to the Shooter games and the final group is the reactive games.

5.2.1. Performance in Maze games

As we can observe in Table 5.1, the scores obtained by π-IW(3/2) for most of the Maze games are very similar to the scores obtained by π-IW, with a slight increase in the game *Ms. Pac-man* and a significant raise in score in the *Q*bert* game.

However, the scores obtained by the π-IW(3/2)-NN version, which implements both enhancements discussed in Section 4.2, obtain significantly better scores than π-IW. Fur-

Game	IW	Rollout IW	π-IW	π-IW(3/2)	π-IW(3/2)-NN
Alien	1.316,0	4.238,0	5.081,4	4.948,0	**6.287,0**
Ms. Pac-man	2.578,0	9.178,4	9.006,5	10.371,4	**15.084,8**
Q*bert	515,0	3.375.0	248.572,5	**416.722,5**	–
Tutankham	71,2	128,4	197,7	**206,7**	–
Venture	0,0	0,0	0,0	0,0	**530,0**
Assault	268,8	285,6	**3.879,3**	3.325,2	2.729,7
Battle Zone	6.800,0	39.600,0	137.500,0	**242.400,0**	–
Centipede	**88.890,0**	36.980,2	32.531,7	77.595,3	39.176,0
Demon Attack	106,0	2.780,0	8.690,1	**66.416,0**	26.439,5
James Bond 007	40,0	450,0	551,0	570,0	**5.415,0**
Space Invaders	280,0	2.628,0	2.385,9	**2.999,5**	2.969,0
Asterix	1.350,0	**45.780,0**	6.852,0	28.780,0	–
Kung-Fu Master	440,0	2.080,0	17.406,0	28.760,0	**30.240,0**
Pong	-20,8	**-7,4**	-19,6	-18,5	-18,9
Road Runner	200,0	2.360,0	100.882,0	26.340,0	**365.410,0**
Skiing	-16.511,0	**-15.738,8**	-26.081,0	-19.907,4	-24.362,0
Tennis	-23,4	-18,6	**-12,2**	-23,5	-23,8
#Top scores	1	3	2	5	6

Table 5.1: Results of the experiments

thermore, the results of π-IW(3/2)-NN outperform the scores of π-IW(3/2). This is an indication that these games largely benefit from the NN enhancements.

As the games in this category have a rather "uniform" game environment, in the sense that the design of the world is not frequently altered, we believe that the Maze games benefit more from a better policy estimate than from the feature enhancement of the NN. Thus, a better guiding policy helps the planning stage choose better actions. In other words, the policy estimate help improve the game strategy.

Specifically, we can observe an increase of 1.205,6 points in the score of the game *Alien* and 6.078,3 in *Ms. Pac-man*, compared to the scores obtained by π-IW. Furthermore, π-IW(3/2)-NN is the only approach able to obtain a score greater than 0 in the *Venture* game, which no other version was capable of.

In the case of the *Q*bert* game, the π-IW(3/2) version greatly outperforms π-IW. This may be due to the fact that the increases in score are associated to how close to the goal the player is. Although we do not have the scores of the π-IW(3/2)-NN version for the *Q*bert* and *Tutankham* games, we foresee that the same trend observed in the other games would also happen here, thus suggesting that the scores would have been higher than π-IW(3/2)'s scores.

In summary, the Maze games appear to benefit from the enhancements to the Neural Network, as it allows the agent to exploit better the learned policy, and for that policy to be more complex. This makes sense if we regard a Maze game as a problem that features rather few large environment variations and where achieving success in this type of games heavily relies on a good game strategy.

5.2.2. Performance in Shooters games

In contrast with the Maze games, the games who belong to the Shooter category seem to benefit more from the more sophisticated planning stage (π-IW(3/2)) than from the

enhancements applied to the Network Architecture. This conclusion follows from the worse scores obtained with π-IW(3/2)-NN compared to those obtained with π-IW(3/2).

On the contrary, we see in Table 5.1 a noticeable increase in the scores obtained by π-IW(3/2) against π-IW's scores. This behaviour indicates that Shooter games benefits from the enhancement of the planning part, as it allows the agent to explore a greater search space, as reflected by the scores increases. And yet, the highest score in *Centipede* is obtained by IW, which uses B-PROST as a feature representation of the game, and seems particularly suited for this specific game.

There are two exceptions to the general behaviour of the games in this category. In *Assault*, we obtain a somewhat lower score with π-IW(3/2), and an even worse one with π-IW(3/2)-NN.

The other exception is the game *James Bond 007*, where the π-IW(3/2)-NN version obtains a significant increase in score. This is probably due to the following reason: the difficulty of the other games in this category augments as the game progresses by increasing the number of enemies, speeding them up, changing their type, etc.; however, in *James Bond 007*, the game flow and the scenario change significantly, and actions that do not cause an effect at the beginning of the game have a noticeable effect later. For example, moving down in the surface of the desert does not achieve anything, but later in the game, moving down when sailing causes the avatar to submerge in the water. And continuing with the example, when we are in the desert, the player can only shoot towards the air, but when the player is in the water, it can also shoot depth charges to attack enemies underwater. This behaviour also happens with the game *Road Runner*, which is discussed below. Both games seem to benefit from the changes to the Neural Network, as the agent is capable of learning more complex behaviour.

As a whole, we can say that Shooter games feature a more dynamic behaviour than Maze games, with frequent environment switches that force the agent to adapt the strategy to the new context. This is the reason why an approach that weighs more the exploration than exploitation of the reinforcement learning approach results in a better performance. Generally speaking, the larger search space explored by the π-IW(3/2) reveals to be very helpful to detect potential environment changes.

5.2.3. Performance in Reactive games

In the reactive games, we can see in Table 5.1 that the scores of π-IW(3/2) are far way better than the scores obtained by π-IW in the games *Asterix* and *Kung-Fu Master* , better in the *Skiing*, and more or less equal to much worse in the games *Pong*, *Road Runner*, and *Tennis*. The reason for the low performance in *Pong* and *Tennis* can be explained due by two different factors. First, in these two games, the rewards are not monotonically increasing. In both, the reward is calculated by subtracting the rival score from our score. Secondly, both games have very sparse rewards, where a long chain of actions is required to obtain a positive score increase. Due to this two factors, π-IW(3/2) does not provide an enhancement to the performance of the agent.

The low score of π-IW(3/2) in *Road Runner* is due to the absence of correlation between the increases in the score and the goal. Particularly, a score increase in this game makes it difficult to win.

Furthermore, some changes in the game flow are a consequence of advancing to the next level, picking up a steel shot, etc., what increases the difficulty of learning to play the game. More particularly, picking up the steel shots, for example, increases the score but allows Wile E. Coyote to reduce the distance with the player significantly. This forces the player to evade using all the directions, instead of the two (up and down) that are used

in the "normal" gameplay, what makes the game be more difficult. Also, in the first level, jumping does not achieve anything, but in the second level we have to jump constantly to get to the next level.

These changes in the gameplay are probably the cause of the increase in performance obtained by π-IW(3/2)-NN, much the same as in the *James Bond 007* game, as the agent seems to benefit from learning a more complex strategy.

We can also see that in *Kung-Fu Master*, the enhancement in planning brings an increase in the score of 11.354 points. Although the π-IW(3/2)-NN obtains an even better score, we can say confidently that *Kung-Fu Master* benefits more from the planning component than the learning one. This may be due to the action-like component of the gameplay.

In general, we cannot identify a clear advantage of the planning or the learning component for reactive games. The nature of this type of games and the sparse rewards make the NN enhancements not being particularly helpful. In the next chapter, we present some methods to overcome the intrinsic difficulties of games such as *Pong* or *Tennis*.

5.2.4. Conclusions of the results

As we can see in Table 5.1, π-IW(3/2) obtains a higher score than π-IW in 12 out of 17 games, and it outperforms π-IW in 11 out of those 12 by a large margin. If we compare the results of π-IW(3/2)-NN against π-IW, we see that π-IW(3/2)-NN outperforms π-IW in 10 out of 13 games. Comparing π-IW(3/2) and π-IW(3/2)-NN, we see that π-IW(3/2) obtains better results in 7 out of 13 games.

We would also like to highlight the score obtained by IW in the game *Centipede*, as it outperforms every other version, and is the only game in which IW obtains the top score. Rollout IW also outperforms every other version in three games: *Asterix*, *Pong* and *Skiing*. Three particular games in which π-IW(3/2) and π-IW(3/2)-NN perform significantly worse than Rollout IW.

Taking into account the top scores obtained by both of our versions, we obtain the top scores of 11 out of 17 games, indicating that this is a promising direction for videogame playing.

CHAPTER 6
Conclusions and future work

As we stated in Chapter 1, our goal in this project was to develop an agent for general game playing, and more specifically, for the Arcade Learning Environment games. To do so, we employed a novel algorithm which combines planning and learning. Specifically, we use a Iterated Width-based algorithm for the planning component, and a Convolutional Neural Network for the learning one. This mixture of planning and learning allows us to get the best of both approaches, as the IW-based algorithms do not make use of past experiences and the NN gets to benefit from the informed exploration that the algorithm provides. The NN is used as an auto-encoder to extract the features from the game screen in a compact way. Furthermore, the NN also learns the policy estimation of the game, which is used in the planning step to guide the search of the best move. More details of the algorithm are discussed in detail in Chapter 4.

We also propose two enhancements over the base method. The first enhancement uses the value of the scores to lessen the pruning of IW, so the search space is increased. The second enhancement consists of a fine-tunning of the hyperparameters and a modification of the NN to improve the features extracted and their number.

We tested our agent in a set of games from the ALE and the results were shown in Section 5.2. Our analysis of the behaviour of the enhancements shows that, depending of the category which the game belongs to, we can expect a increase or decrease of the average score for that game. Specifically, the games which belong to the Maze genre obtain an increase in performance with the version that employ both enhancements (π-IW(3/2)-NN), as this kind of game seem to benefit from a better policy estimation. The Shooter games benefit clearly from the version which employ the first enhancement (π-IW(3/2)), signifying that the agents in this genre, which has more actions, will perform better thanks to the planning step. The results of the Reactive games vary depending on the characteristics of each game, as we have games which perform worse, a game that benefits clearly from the planning improvement and a game that benefits from the NN enhancement. We discuss the possible reason for this behaviour in the analysis in Section 5.2.3.

We will now discuss several research directions that can be followed to improve the performance of our agent.

- Create a multi-agent extension to be able to play team games such as *football*, *basketball*, *Dota 2* or *League of Legends*, which are games where each agent may have a specialized policy for the particular role it plays in the team.

- Change the current experience replay and implement Prioritized Experience Replay from [41] to learn from the most informative experiences stored.

- After the previous one, implementing Ape-X DQN, which utilizes Distributed Prioritized Experience Replay [21], a distributed version that uses multiple actors which explore their instance of the environment, generate experience, add it to a shared experience replay memory, and compute initial priorities for the data. The single learner samples from this memory and updates the network and the priorities of the experience in the memory. The actors' networks are periodically updated with the latest network parameters from the learner. The actors are given different exploration policies, broadening the diversity of the experience they jointly encounter.
- Experiment with more games, from the Atari Learning Environment and other settings, like the General Video Game Competition, with the aim to see the behaviour of the algorithm in a larger and more diverse sets of games.

Bibliography

[1] Mario A. http://www.marioai.org/. Accessed: 2019-07-29.

[2] Stockfish. https://stockfishchess.org/. Accessed: 2019-06-29.

[3] The GVG-AI Competition. http://www.gvgai.net/. Accessed: 2019-06-03.

[4] AlphaGo. https://deepmind.com/research/alphago/. Accessed: 2019-07-03.

[5] Kai Arulkumaran, Antoine Cully, and Julian Togelius. AlphaStar: An Evolutionary Computation Perspective. *CoRR*, abs/1902.01724, 2019.

[6] Wilmer Bandres, Blai Bonet, and Hector Geffner. Planning With Pixels in (Almost) Real Time. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18)*, pages 6102–6109, 2018.

[7] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The Arcade Learning Environment: An Evaluation Platform for General Agents. *CoRR*, abs/1207.4708, 2012.

[8] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[9] Darse Billings, Aaron Davidson, Jonathan Schaeffer, and Duane Szafron. The challenge of poker. *Artificial Intelligence*, 134(1-2):201–240, 2002.

[10] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. arXiv:1606.01540, 2016.

[11] Mat Buckland. *Programming game AI by example*. Jones & Bartlett Learning, 2005.

[12] Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134(1-2):57–83, 2002.

[13] Michael Chung, Michael Buro, and Jonathan Schaeffer. Monte carlo planning in rts games. In *CIG*. Citeseer, 2005.

[14] N. Cole, S. J. Louis, and C. Miles. Using a genetic algorithm to tune first-person shooter bots. In *Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No.04TH8753)*, volume 1, pages 139–145 Vol.1, June 2004.

[15] Richard E Fikes and Nils J Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189–208, 1971.

[16] Maria Fox and Derek Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003.

[17] Tomas Geffner and Hector Geffner. Width-Based Planning for General Video-Game Playing. In *Proceedings of the Eleventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE*, pages 23–29, 2015.

[18] Tomas Geffner and Hector Geffner. Width-based planning for general video-game playing. In *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2015.

[19] Alfonso Emilio Gerevini, Nir Lipovetzky, Francesco Percassi, Alessandro Saetti, and Ivan Serina. Best-First Width Search for Multi Agent Privacy-Preserving Planning. In *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling, ICAPS*, pages 163–171, 2019.

[20] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: training imagenet in 1 hour. *CoRR*, abs/1706.02677, 2017.

[21] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver. Distributed prioritized experience replay. *CoRR*, abs/1803.00933, 2018.

[22] Ruoyun Huang, Yixin Chen, and Weixiong Zhang. Sas+ planning as satisfiability. *Journal of Artificial Intelligence Research*, 43:293–328, 2012.

[23] Andreas Junghanns and Jonathan Schaeffer. Sokoban: Enhancing general single-agent search methods using domain knowledge. *Artificial Intelligence*, 129(1-2):219–251, 2001.

[24] Miquel Junyent, Anders Jonsson, and Vicenç Gómez. Deep policies for width-based planning in pixel domains. *arXiv preprint arXiv:1904.07091*, 2019.

[25] Richard Kaye. Minesweeper is np-complete. *The Mathematical Intelligencer*, 22(2):9–15, 2000.

[26] Michal Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jaskowski. Vizdoom: A doom-based AI research platform for visual reinforcement learning. *CoRR*, abs/1605.02097, 2016.

[27] Donald E Knuth and Ronald W Moore. An analysis of alpha-beta pruning. *Artificial intelligence*, 6(4):293–326, 1975.

[28] Richard E Korf. Finding optimal solutions to rubik's cube using pattern databases. In *AAAI/IAAI*, pages 700–705, 1997.

[29] Joel Lehman and Kenneth Stanley. Exploiting open-endedness to solve problems through the search for novelty. *Artificial Life - ALIFE*, pages 329–336, 01 2008.

[30] Yitao Liang, Marlos C Machado, Erik Talvitie, and Michael Bowling. State of the Art Control of Atari Games Using Shallow Reinforcement Learning. In *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*, pages 485–493. International Foundation for Autonomous Agents and Multiagent Systems, 2016.

[31] Nir Lipovetzky and Hector Geffner. Width and Serialization of Classical Planning Problems. In *ECAI 2012 - 20th European Conference on Artificial Intelligence*, pages 540–545, 2012.

[32] Nir Lipovetzky and Hector Geffner. Width-based Algorithms for Classical Planning: New Results. In *ECAI 2014 - 21st European Conference on Artificial Intelligence*, pages 1059–1060, 2014.

[33] Nir Lipovetzky and Hector Geffner. A Polynomial Planning Algorithm that Beats LAMA and FF. In *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling, ICAPS*, pages 195–199, 2017.

[34] Nir Lipovetzky and Hector Geffner. Best-First Width Search: Exploration and Exploitation in Classical Planning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, pages 3590–3596, 2017.

[35] Nir Lipovetzky, Miquel Ramírez, and Hector Geffner. Classical Planning Algorithms on the Atari Video Games. In *Learning for General Competency in Video Games, Papers from the 2015 AAAI Workshop*, 2015.

[36] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[37] Santiago Ontanón, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, David Churchill, and Mike Preuss. A survey of real-time strategy game ai research and competition in starcraft. *IEEE Transactions on Computational Intelligence and AI in games*, 5(4):293–311, 2013.

[38] Juan Ortega, Noor Shaker, Julian Togelius, and Georgios N Yannakakis. Imitating human playing styles in super mario bros. *Entertainment Computing*, 4(2):93–104, 2013.

[39] Diego Perez-Liebana, Spyridon Samothrakis, Julian Togelius, Tom Schaul, and Simon M Lucas. General video game ai: Competition, challenges and opportunities. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[40] Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *science*, 317(5844):1518–1522, 2007.

[41] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. In *4th International Conference on Learning Representations, ICLR*, 2016.

[42] Alexander Shleyfman, Alexander Tuisov, and Carmel Domshlak. Blind Search for Atari-Like Online Planning Revisited. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI*, pages 3251–3257, 2016.

[43] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529(7587):484–489, 2016.

[44] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[45] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *CoRR*, abs/1712.01815, 2017.

[46] Samuel L. Smith, Pieter-Jan Kindermans, and Quoc V. Le. Don't decay the learning rate, increase the batch size. *CoRR*, abs/1711.00489, 2017.

[47] Richard S Sutton, Andrew G Barto, et al. *Introduction to reinforcement learning*, volume 2. MIT press Cambridge, 1998.

[48] Gerald Tesauro. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural computation*, 6(2):215–219, 1994.

[49] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, 2012.

[50] Julian Togelius, Noor Shaker, Sergey Karakovskiy, and Georgios N Yannakakis. The mario ai championship 2009-2012. *AI Magazine*, 34(3):89–92, 2013.

[51] Tom Wijman. The Global Games Market Will Generate 152.1 Billion in 2019 as the U.S. Overtakes China as the Biggest Market. https://newzoo.com/insights/articles/the-global-games-market-will-generate-152-1-billion-in-2019-as-the-u-s-\overtakes-china-as-the-biggest-market/, 06 2019. Accessed: 2019-09-02.