TRABAJO FIN DE GRADO

INGENIERÍA INFORMÁTICA

# Learning to Select Goals with Deep Q-Learning in GVGAI


**Autor**

Carlos Núñez Molina

**Director**

Juan Fernández Olivares



Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación


—

Granada, julio de 2020

# Aprendiendo a Seleccionar Objetivos con Deep Q-Learning en GVGAI

Carlos Núñez Molina

**Palabras clave**: planificación automática, razonamiento con objetivos, deep q-learning

## Resumen

Las técnicas de Planificación Automática han sido aplicadas exitosamente para resolver una amplia gama de problemas. Sin embargo, estas técnicas no son adecuadas para escenarios con estrictas restricciones temporales, como es el caso de los sistemas de ejecución online. Este Trabajo Fin de Grado propone una arquitectura de planificación y actuación que aprende a seleccionar subobjetivos con Deep Q-Learning para disminuir la carga del planificador. En este proyecto, hemos diseñado e implementado dicha arquitectura, la hemos entrenado en un entorno de videojuegos conocido como GVGAI, hemos evaluado su rendimiento en niveles diferentes de los usados en el entrenamiento para así poder medir su capacidad de generalización y, finalmente, hemos redactado una publicación científica con los resultados obtenidos.

# Learning to Select Goals with Deep Q-Learning in GVGAI

Carlos Núñez Molina

**Keywords**: automated planning, goal reasoning, deep q-learning

**Abstract**

Automated Planning techniques have been successfully applied to solve a wide range of problems. However, they are not suited for scenarios with tight time restrictions, such as online execution systems. This Final Degree Project proposes a planning and acting architecture which learns to select subgoals with Deep Q-Learning in order to decrease the load of the planner. In this project, we have designed and implemented this architecture, trained it in a video game environment called GVGAI, evaluated its performance on different levels from the ones used for training in order to measure its generalization abilities and, finally, written a scientific paper presenting our results.

Yo, Carlos Núñez Molina, alumno de la titulación Ingeniería Informática de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 76655197-S, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Firmado por NUÑEZ MOLINA CARLOS - 76655197S el día 27/06/2020 con un certificado emitido por AC FNMT Usuarios

Fdo: Carlos Núñez Molina

Granada a 11 de junio de 2020.

D. **Juan Fernández Olivares**, Profesor Titular del Departamento de Ciencias de la Computación e Inteligencia Artificial de la Universidad de Granada.

**Informa:**

Que el presente trabajo, titulado ***Aprendiendo a Seleccionar Objetivos con Deep Q-Learning en GVGAI***, ha sido realizado bajo su supervisión por **Carlos Núñez Molina**, y autoriza la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expide y firma el presente informe en Granada a 11 de junio de 2020.

**El director:**

FERNANDEZ OLIVARES JUAN - 26215645S

Firmado digitalmente por FERNANDEZ OLIVARES JUAN - 26215645S
Fecha: 2020.06.11 15:05:54 +02'00'

**Juan Fernández Olivares**

## Acknowledgements

Firstly, I thank Vladislav Nikolov-Vasilev for implementing the PDDL Parser used in this project.

Secondly, I thank the Spanish MINECO for partially funding this Final Degree Project in the context of the R&D Project TIN2015-71618-R and RTI2018-098460-B-I00.

Finally, I want to thank my tutor Dr. Juan Fernández Olivares for granting me the opportunity to participate in a research project to apply Goal Reasoning in Automated Planning and which would eventually turn into this Final Degree Project.

# Contents

# 1 Introduction

Automated Planning [9] is a subfield of Artificial Intelligence devoted to providing goal-oriented, deliberative behaviour to both physical and virtual agents, e.g. robots or video game automated players. An automated planner takes as input a planning domain, an initial state and a goal and carries out a search process that returns a plan (sequence of actions) that guides the behaviour of the agent in order to reach the given goal from the initial state. The planning domain describes the actions an agent can execute as well as the dynamics of the environment where the agent is expected to act. The initial state is a set of facts describing the context in which the agent initiates its behaviour, and the goal is a set of conditions which need to be accomplished at the end of the plan.

Automated Planning has traditionally been one of the most widely used techniques in AI and has been successfully applied in real-world applications [5, 7]. However, it has been difficult to integrate it into online execution systems, i.e., systems used in real-time scenarios which interleave planning and acting. Firstly, planning is too slow for real-time scenarios. In most real-world problems the search space is enormous so, despite the use of heuristics, finding a suitable plan usually takes very long. Secondly, since most real-world environments are highly dynamic, it is very likely that the environment has changed before a long plan has finished being executed.

Due to this, despite great advances in the integration of planning and acting into online architectures [18], most recent works which apply AI to guide agents behaviour in real-time scenarios, like video games, do not integrate planning into their agent architecture. This can be clearly seen in [25]. In this impactful work, an agent is trained to play *Starcraft*, a highly competitive real-time strategy (RTS) game. This seems like a perfect problem for planning: players need to establish a long-term, goal-oriented strategy in order to achieve victory and all the dynamics of the game are known, so they can be represented into a planning domain. However, Vinyals et al. choose to integrate Deep Learning [14] with Reinforcement Learning [21] to model the behaviour of the agent.

Architectures which rely on Machine Learning (ML) and Reinforcement Learning (RL) present some advantages over planning: they usually require very little prior knowledge about the domain (they do not need a planning domain) and, once trained, they act quickly, since they do not perform any type of planning. Nevertheless, they also have some drawbacks. Firstly, they are very sample inefficient. They require a lot of data in order to learn, in the order of hundreds of thousands or even millions samples [23]. Secondly, they usually present bad generalization properties, i.e., have difficulties in applying what they have learnt not only to new domains but also to new problems of the same domain [27].

Since both Automated Planning and Reinforcement Learning have their

own pros and cons, it seems natural to try to combine them as part of the same agent architecture, which ideally would possess the best of both worlds. For that purpose, we have resorted to *Goal Reasoning* [1], a design philosophy for agents in which its entire behaviour revolves around goals. They learn to formulate goals, select goals, achieve the selected goals and select new goals when *discrepancies* are detected.

This Final Degree Project presents a novel RL-based Goal Selection Module and its integration into a planning and acting architecture to control the behaviour of an agent in a real-time environment. We have trained and tested our approach in the GVGAI video game framework [19]. GVGAI is a framework intended to evaluate the behaviour of reactive and deliberative agents in several video games. Its ultimate goal is to help advance the state of the art in General Artificial Intelligence.

The Goal Selection Module here described is based on a Convolutional Neural Network (CNN) [13] which has been trained with the RL algorithm known as Deep Q-Learning [15]. The training experience has been extracted from the execution of hundreds of episodes of an agent in the GVGAI environment, on different levels from the ones used for testing, so as to evaluate the generalization ability of the module.

The CNN receives as input an image-like encoding of the current state of the game $s$ and an eligible subgoal $g$ and outputs the predicted length of the plan which starts at $s$, achieves $g$ and then achieves the final goal (wins the game). The Goal Selection Module selects the subgoal $g^*$ whose associated plan has the minimum predicted length. After selecting $g^*$, the Planner Module finds a suitable plan from $s$ to $g^*$, trying to minimize its length, which will then be executed by the agent in GVGAI.

Addressing Goal Selection with Deep Q-Learning and a CNN has two main advantages. Firstly, as the results of our experiments show, the Goal Selection Module learns to generalize. The use of a CNN allows it to apply what has learnt on the training levels to new levels it has never seen before. Secondly, thanks to the use of Deep Q-Learning, the Goal Selection Module learns to select goals *thinking in the long term*, i.e, taking into account the subgoals it will have to achieve afterwards to beat the game.

The structure of this report is the following. It starts by presenting an overview of the tools needed to understand this work. After that, it shows the scheduling of the different tasks accomplished throughout the project. Then, it describes the general system architecture and, in the next section, focuses on the Goal Selection Module and how it learns to select subgoals. Consecutively, it explains the empirical study conducted to assess the quality of the proposed approach. Then, it briefly explains other works related to the topic of this project. Finally, the last section presents the conclusions of this work and summarizes what I have learnt during the development of this project.

# 2 Background

This section presents an overview of the main tools and techniques used in this work.

## 2.1 GVGAI

To test our planning and acting architecture we have used the General Video Game AI (GVGAI) Framework [19]. This framework provides a game environment with a large quantity of tile-based games which are also very different in kind. For example, it comprises purely reactive games, such as *Space Invaders*, and also games which require long-term planning in order to be solved successfully, such as *Sokoban*.

In this work, we have chosen to use a deterministic version of the GVGAI game known as *Boulder Dash*. We use this game to extract the experience of episodes of planning and acting our Goal Selection Module is trained on. In our version of Boulder Dash, the player must collect nine gems and then go to the exit, while minimizing the number of actions used. In order to do that, it must traverse the level (one tile at a time) while overcoming the obstacles: the player cannot pass through walls and boulders must be broken with its pickaxe before passing through. Also, the player must select which gems to collect, since there are more than nine gems available. All of this makes it really hard to find the shortest plan which collects nine gems and then gets to the exit. Figure 1 shows a level of Boulder Dash.

Figure 1: A level of the Boulder Dash game.

One very important reason we have chosen GVGAI is because it makes available a mechanism for easily creating and integrating new games and levels. This way, we can create as many new levels for a given game as we

want, which allows us to test the generalization abilities of a planning and acting architecture when its Goal Selection Module has already been trained. The Video Game Description Language, VGDL, [19] is the method used to define the dynamics and interactions of all the objects in each of the games. Every level in the game is defined by a level description file, which contains the layout of the level and the initial positions of the objects. Listing 1 shows the associated level description file of the game level shown on Figure 1. Each type of object has a character associated: *w* for walls, *o* for boulders, *x* for gems, *A* for the player, *e* for the exit, . for tiles and - for empty tiles, which are the same as normal tiles.

```
wwwwwwwwwwwwwwwwwwwwwwwwww
w . . . o . xx . o . . . . . o . . xoxx . . w
w . . . oooooo . . . . . . . . o . . o . . . w
w . . . . xxx . . . . . . . . . o . oxoo . ow
wx . . . . . . . . . . . . . . . oxo . . . oow
wwwwwwwwww . . . . . . . . . o . . . wxxw
w . — . . . . o . . . . . . . . . . . . . wxxw
w — — . . . . . Ao . . . . o . . . . wxxw
wooo . . . . . — . . . . w . . w
w . . . . . x . . . . wwwwx–x . oow . . w
w . — — . . . . . x . . ooxxo — . . . . w . . w
w———..e . . . . . — — — — — . . w
wwwwwwwwwwwwwwwwwwwwwwwwww
```

Listing 1: The level description file of the level shown on Figure 1.

## 2.2 PDDL

To encode the inputs for our Planner Module, we have used PDDL. PDDL [8] is a standardized language used in Automated Planning for representing planning domains and problems. The PDDL Domain file contains the information about the planning domain: a description of the predicates used to represent a problem state and the preconditions and effects of every action. The PDDL Problem file contains the description of a given planning problem: a representation of its initial state and the goal to achieve. A planner receives as inputs these two files and returns a plan, which is constituted by an ordered sequence of instantiated PDDL actions.

Given a GVGAI game, we can create its associated planning domain. This domain will encode the game dynamics, i.e., the different entities of the game (Listing 2) and all the actions the player can do to interact with them (Listing 3). Each game level will have a different planning problem associated, representing its initial state (Listing 4) and the goal to achieve

(Listing 5). For instance, in Boulder Dash a goal corresponds to getting a gem present at the level.

```
(:predicates
        (at ?l - Locatable ?c - Cell)
        (connected-up ?c1 ?c2 - Cell)
        (connected-down ?c1 ?c2 - Cell)
        (connected-left ?c1 ?c2 - Cell)
        (connected-right ?c1 ?c2 - Cell)
)
```

Listing 2: Predicates used to represent the position of entities in the BoulderDash domain. We use the predicate *at* to represent which cell an object is in and *connected-** to encode the adjacent cells to a given cell.

```
(:action move-up
        :parameters (?p - Player ?c1 ?c2 - Cell)
        :precondition (and
          (at ?p ?c1)
          (oriented-up ?p)
          (connected-up ?c1 ?c2)
          (not (exists (?b - Boulder)
               (at ?b ?c2)))
          (not (terrain-wall ?c2))
        )
        :effect (and
          (when
            (not (terrain-empty ?c2))
            (terrain-empty ?c2)
          )
          (not (at ?p ?c1))
          (at ?p ?c2)
        )
)
```

Listing 3: Preconditions and effects of the *move-up* action in the BoulderDash domain. The player must be oriented *up* and there cannot be a wall or a boulder in the cell it is going to.

```
(:init
        (at gem1 c_5_3)
        (connected-up c_5_3 c_5_2)
        (connected-down c_5_3 c_5_4)
        (connected-right c_5_3 c_6_3)
        (connected-left c_5_3 c_4_3)
```

```
)
```

Listing 4: Part of the predicates describing the initial state of a given BoulderDash problem. The gem *gem1* is in cell *c_5_3*, which is next to cells *c_5_2*, *c_5_4*, *c_6_3* and *c_4_3*.

```
(:goal
        (and
          (got gem13)
        )
)
```

Listing 5: Goal description of a given BoulderDash problem. In this problem, the player must pick the gem *gem13*.

## 2.3 Deep Q-Learning

Q-Learning [26] is one of the most widely used techniques in Reinforcement Learning, RL, [21]. As every RL technique, it learns a policy $\pi$ that, in every state $s$, selects the best action $a$ in the set of available actions $A$ in order to maximize the expected cumulative reward $R$, i.e., the expected sum of all the (discounted) rewards $r$ obtained by choosing actions according to the same policy $\pi$ from the current state $s$ until the end of the episode. According to the *Reward Hypothesis*, all goals can be described as the maximization of $R$. This means that, no matter the goal an agent is pursuing, its behaviour can be modeled and learnt (more or less successfully) using a RL technique, such as Q-Learning.

Q-Learning associates a value to each $(s, a)$ pair, known as the Q-value $Q(s, a)$. This value represents the expected cumulative reward $R$ associated with executing action $a$ in state $s$, i.e., how good $a$ is when applied in $s$. This way, the policy $\pi$ learnt with Q-Learning corresponds to, given a state $s$, selecting the action $a^*$ in $A$ with the maximum Q-value associated.

One of the main problems Q-Learning has is that it needs to learn the associated Q-value for each of the $(s, a)$ pairs, known as the *Q-table*. If the action or state space are too big, the Q-table grows and the learning problem becomes intractable. Deep Q-Learning [15] solves this problem. Instead of learning the Q-table, it uses a Deep Neural Network (DNN) to learn the Q-values. Thanks to the use of a DNN, it is able to generalize and correctly predict the Q-values for new $(s, a)$ pairs never seen before by the network. In our work, we select the best subgoal from a set of possible subgoals. The set of possible subgoals depends on the current state $s$. Since the state space is enormous, the size of the set of possible subgoals across all different states is also really big. For this reason, we use Deep Q-Learning in pursuit of the good generalization abilities shown in [15].

# 3 Work Plan

For the development of this project, we decided to follow an agile methodology inspired by Scrum. Each week, I would have a meeting (face-to-face or by Skype) with my project tutor in which we would decide the goal to accomplish the next week. Then, I would work towards that goal, modifying the project in an incremental manner. At the next meeting, we would see if new problems had arisen and, in that case, how to solve them as well as assess the current state of the project and decide if a change of direction was needed. All of this would be taken into account before deciding the goal for the next week.

This methodology made possible to divide quite a big project into manageable parts (iterations) which were achievable in the course of a single week. At the same time, the weekly meetings allowed us to have a global view on the current state of the project and, this way, be able to correctly direct our efforts, easily adapting to new changes and altering our approach when was needed.

The Work Plan I followed for this project can be divided into the following tasks: Learning, Analysis, Design, Implementation, Experimentation and Writing. Figure 2 shows the scheduling of these tasks throughout the development of the project as a Gantt Chart:

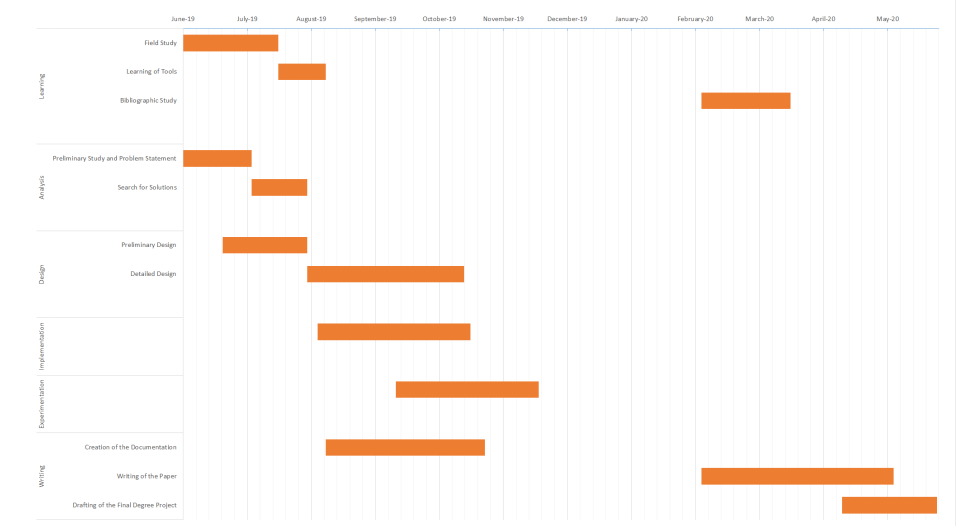


Figure 2: Gantt Chart showing the scheduling of the project.

## 3.1 Learning

This task comprises all the different things I had to learn in order to develop this project.

Firstly, I had to study all the fields related to this project, mainly Automated Planning and Machine Learning (ML). Regarding Automated Planning, I had to review the contents of a subject I took in my third year called *Técnicas de los Sistemas Inteligentes*. However, regarding ML, I had to learn almost everything on my own, since most concepts and techniques I needed were not taught in the subject called *Aprendizaje Automático*. On the one hand, I had to learn a lot about Convolutional Neural Networks (CNN). CNNs are taught in a subject called *Visión por Computador* which, unfortunately, I took after I had already started this project. On the other hand, I had to learn a lot about Reinforcement Learning (RL), which constitutes a fundamental part of this project. Concretely, I focused on a technique known as Deep Q-Learning.

In addition to all of this, I had to learn to use several tools. Firstly, I had to learn about the GVGAI game framework, which constitutes the execution environment for this project. Secondly, I had to learn TensorFlow in order to implement the learning models (neural networks) and, also, to use Jupyter Notebooks, which I employed to test the learning models before integrating them into the system architecture.

Lastly, I had to do a bibliographic study for writing the paper. Although I examined some papers throughout all the development of the project, I read most of them as I was writing the paper. I mainly focused on works which interleaved planning and execution using Goal Reasoning or combined Automated Planning and ML in some way.

## 3.2 Analysis

As soon as I knew the topic of the project, I did a preliminary study of the problem and started thinking how I could frame it. Mainly, this meant enumerating the problem requirements. On the one hand, the functional requirements (the *what*): creating an architecture which interleft planning and execution in a game environment, used Goal Selection for that purpose and learnt to select subgoals using ML. On the other hand, the non-functional requirements (the *how*): the approach had to be generalizable across a wide range of domains and not only applicable to the GVGAI game used in this project, it had to be as *sample-efficient* as possible (be able to learn with little data) and learn to select good subgoals (meet a quality criteria).

After the problem statement, I started evaluating different alternatives to solve the problem. Since GVGAI was the chosen environment, I thought about using RL to learn to select subgoals using the data collected by an agent playing in GVGAI. Then, I considered how to perform the Goal Selection. In Boulder Dash (the game used for this project), each subgoal is associated with picking a gem and player's goal is to complete the level using the smallest number of actions. In order to select subgoals (gems) following this criteria, I first thought of selecting the closest gem to the agent

given the current state of the game. However, I realized that this greedy approach does not always result in an optimum plan, i.e., the shortest one. As I reflected on this issue I noticed how the RL algorithm known as Deep Q-Learning solves this problem: it selects actions considering not only the short-term consequences but also thinking in the long-term. Thus, I saw I could use this algorithm if I considered each subgoal as a different action and used the planner to achieve each selected subgoal. I decided to test and compare these two goal selection approaches, which I called Greedy Model and DQP (Deep Q-Planning) Model.

## 3.3 Design

After completing the analysis, I started with the design. I first made a preliminary design, i.e., a design with a high level of abstraction. The most important task was designing the general system architecture. This architecture was composed of three main modules. One was the Goal Selection Module which, given a set of subgoals and the current game state, would select the best subgoal using either the approach followed by the Greedy Model or the one of the DQP Model. Another one was the Planner Module which, given a selected subgoal, would find a suitable plan that achieved it. The last one was the Execution and Monitoring Module, which would execute the plan and, after finishing its execution, would use the other two modules to select a new subgoal and find a new plan to it. In addition, I noticed that the execution in GVGAI is split into two different phases: a training phase, in which agents learn to play the game, and a validation phase, in which agents' performance is evaluated. Due to this, during the training phase the architecture (concretely the Goal Selection Module) must learn to select subgoals and then, during the validation phase, it needs to apply what has learnt in the training.

Following this preliminary design, I made a more detailed, closer-to-code design. Since I had decided to implement all the code in Python, I thought about which scripts to create. I split the code in two main scripts: *LearningModel.py*, which contains all the TensorFlow code of the learning model, and *Agent.py*, which implements the logic of the agent. Then, I started with the implementation and, during this stage, I designed and implemented the different methods of both scripts. All the code was stored in a GitHub repository. I decided to use several GitHub branches for the development, since I realized I would need to test different alternatives/versions of the architecture and it would be easier to separate them in different branches. As I continued with the development and started doing experiments, I designed and implemented some auxiliary scripts so as to automatize the testing of the different models.

### 3.4 Implementation

This is the task associated with the development of all the code of the project.

As I finished the preliminary design and started making the detailed design, I began implementing the methods I designed. Before integrating the Greedy Model into the architecture I wanted to make sure the TensorFlow code I had implemented worked fine. For that purpose, I extracted a small dataset using GVGAI and created a Jupyter Notebook to test if the learning model worked correctly and was really able to learn.

### 3.5 Experimentation

This taks is composed of two different parts. On the one hand, all the experiments I did in order to select the best parameters for both the Greedy and DQP models: the network architecture, number of training iterations, whether to use Fixed Q-targets to improve DQP Model's performance, etc. On the other hand, after finishing the implementation of the Greedy and DQP models I conducted an empirical study to test and compare the performance of both models. These final results are the ones shown on the paper.

For all these experiments, the learning models have always been trained on datasets collected on the training levels of Boulder Dash. In order to select the model's parameters, the different alternatives were evaluated on the validation levels, different from the training levels. To obtain the final results, the models were evaluated on the test levels, different from the training and validation levels. This way, I have followed the recommended methodology in ML, consisting in splitting the data in disjoint sets for training, validation and testing.

### 3.6 Writing

This task is associated with the writing of all the project documents.

A part of it has been the creation of all the documentation associated with the code implemented. I started doing this subtask as I advanced with the implementation, creating documents in various formats (Word, MarkDown for GitHub, etc.). Once I started writing the documents needed for my Final Degree Project, I unified the documentation and completed its missing parts. This part also includes all the use tutorials I made explaining different things: how the GVGAI framework works, how to install and execute the architecture in GVGAI, etc. These tutorials facilitated the supervision and understanding of the project to my project tutor.

A very important part has been writing the paper. After completing the implementation and experimentation (and my final exams in the first

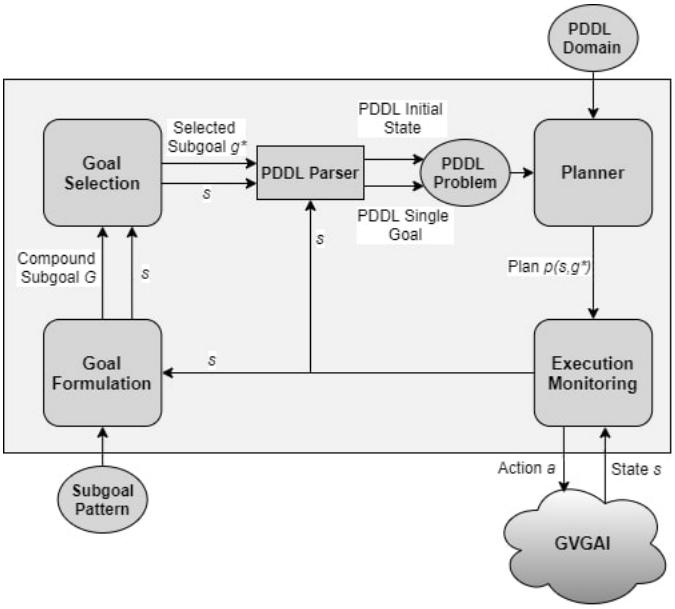


Figure 3: An overview of the planning and acting architecture.

semester) I started writing a paper for the ICAPS Workshop. Once I finished, I had to adapt it (mainly, to greatly reduce its length) for the CAEPIA Conference. However, this event was cancelled due to COVID-19, so I had to adapt the paper again, this time for a journal called Expert Systems with Applications.

As I was writing the paper for the journal, I started drafting the documents needed for my Final Degree Project. Firstly, I unified and completed all the documentation I had created throughout the project. Secondly, I wrote the project report. This was the part I spent the most time on in this phase. Lastly, I spent a week elaborating the presentation for my project.

# 4 The Planning and Acting Architecture

An overview of the planning and acting architecture can be seen on Figure 3. The **Execution Monitoring** Module communicates with the GVGAI environment, receiving the current state $s$ of the game. It also supervises the state of the current plan. If it is not empty, it returns the next action $a$. If it is empty, the architecture needs to find a new plan. The **Goal Formulation** Module receives $s$ and generates the compound subgoal $G$, which is a list of single subgoals $\{g_1, g_2, ..., g_n\}$. The **Subgoal Pattern** contains the prior information about the domain needed to automatically generate $G$ given $s$. In Boulder Dash each $g \in G$ corresponds to getting one of the available gems in $s$ or, if the player has already got nine gems, the final goal $g_f$ (*get to the*

*exit*) is directly attainable and $G = \{g_f\}$. Since all GVGAI games are tile-based, we have associated each subgoal with getting to its correspondent tile. The **Goal Selection** Module receives $G$ and selects the best subgoal $g^* \in G$ given $s$. The **PDDL Parser** encodes $g^*$ as a PDDL Single Goal, i.e., (*at player tile1*) (assuming $g^*$ is associated with *tile1*), and $s$ as a PDDL Initial State, which together constitute the PDDL Problem. The **Planner** Module receives the PDDL Problem along with the PDDL Domain, provided by a human expert, and generates a plan $p(s, g^*)$ which achieves $g^*$ starting from $s$. Finally, the Execution Monitoring Module receives $p(s, g^*)$ and the cycle completes.

## 5 Goal Selection Learning

In order to select the best subgoal $g^* \in G$ for a given $s$, the Goal Selection Module iterates over every $g \in G$ and predicts the length of its associated plan. It then selects as $g^*$ the subgoal whose associated plan has been predicted the minimum length. The Module uses a Convolutional Neural Network (CNN) [13] that receives $s$ and a $g \in G$, both encoded as a *one-hot matrix*, and outputs the predicted plan length. Each position of this one-hot matrix corresponds to a tile of the level and encodes the objects within that tile as a *one-hot vector*, i.e., a vector where each position is associated with a different type of object and which contains *1* if that object is in that tile and *0* otherwise. The subgoal $g$ is also encoded in the one-hot vector of its associated tile.

In this work, we have tested two different approaches for Goal Selection: the **Greedy Model** and the **DQP Model**. The Greedy Model predicts as $l_{p(s,g)}$ the number of actions of the plan $p(s, g)$ that, starting from $s$, achieves $g$. The DQP Model predicts as $l_{P(s,g)}$ the length of the plan $P(s, g)$ that achieves $g$ *and*, after obtaining it, achieves the final goal $g_f$ (after obtaining all the required subgoals). This way, the Greedy Model entails a greedy approach: it only predicts the length of the plan to $g$ and does not care about the rest of the plan to achieve $g_f$. On the other hand, the DQP Model predicts the length of the entire plan, not only the first section of it. For this reason, the subgoal $g^*$ selected by the Goal Selection Module depends on the chosen strategy.

The Greedy Model is constituted by a shallow CNN with two hidden layers: a convolutional layer with 2 filters and *max pooling* and a fully-connected layer with 16 units. The CNN is trained in a supervised fashion, minimizing the squared difference between the predicted $l_{p(s,g)}$ and the real length, during 5000 training iterations.

Unlike the Greedy Model, the DQP Model predicts the length of the entire plan $P(s, g)$. Since only the length of the first section $p(s, g)$ is known, this model cannot be trained in a supervised fashion. To train this model,

we have chosen to apply the methodology followed by Deep Q-Learning [15]. To do so, we establish a correspondence between our problem and Reinforcement Learning (RL). Actions $a$ in RL correspond in our work to achieving a subgoal $g$, the reward $r$ obtained by executing $a$ at $s$ corresponds to the length of the plan $p_{s,g}$ that starts at $s$ and achieves a subgoal $g$, the expected cumulative reward $R$ associated with $(s, a)$ corresponds to the length $l_{P(s,g)}$ of the entire plan $P_{s,g}$, and maximizing $R$ corresponds to minimizing $l_{P(s,g)}$. Table 5 shows this correspondence:

| **RL** | **Our Work** |
|---|---|
| Action $a$ | Subgoal $g$ |
| Reward $r$ | $l_{p(s,g)}$ |
| Cumulative Reward $R$ | $l_{P(s,g)}$ |
| Maximize $R$ | Minimize $l_{P(s,g)}$ |

Table 1: Correspondence between RL and our problem.

The CNN of the DQP Model predicts $l_{P(s,g)}$, which in Deep Q-Learning corresponds to the Q-value $Q(s, a)$. Since its correct value,-target $Q^*(s, a)$, is unknown, it is estimated using other predicted Q-values $Q(s', a')$ in a technique known as *bootstrapping*. This is the method used to learn the Q-values. The network is trained by minimizing the squared difference between $Q(s, a)$ and $Q^*(s, a)$. This loss $L$ formula is called the Bellman Equation and is shown below:

$$L = (Q(s, a) - Q^*(s, a))^2 = (Q(s, a) - (r + \gamma \max_{a' \in A'} Q(s', a')))^2 \qquad (1)$$

where $s'$ is the next state (after applying $a$ in $s$), $A'$ is the set of applicable actions in $s'$ and $\gamma = 0.9$ is the *discount factor*.

The CNN architecture used for the DQP Model is the same as the one used for the Greedy Model except for two things: the convolutional layer uses 4 filters and an additional fully-connected layer with 64 units is added before the other fully-connected layer. Also, in order to make learning more stable, an auxiliary CNN is used to estimate the Q-targets, in a technique known as *Fixed Q-targets* [16]. This CNN has fixed weights and every $\tau = 250$ training iterations the weights of the main CNN are copied to it.

Both the Greedy and DQP models use *offline learning*, i.e., are trained on static datasets. These datasets are populated by performing *random exploration* on the training levels of the corresponding game. Each time the Goal Selection Module must select a new subgoal $g^*$ for the current state $s$, it selects it randomly. Then, when the architecture has found $p(s, g^*)$ and executed it arriving at state $s'$, a new sample is added to the datasets. The datasets of the Greedy Model are constituted by samples of the form

$(s, g^*, r)$ whereas the datasets of the DQP Model are filled with samples of the form $(s, g^*, r, s')$.

# 6 Experiments and Analysis of Results

We have chosen the Fast-Forward (FF) Planning System [10] for our Planner Module since the version of PDDL its parser uses is expressive enough to represent domains such as those of video games. With the optimization options we have used, the plans FF obtains are short but not always optimum (in length).

For the Greedy and DQP models, we have collected datasets of different sizes on the 3 training levels GVGAI provides for Boulder Dash. The number of samples of these datasets are: 500, 1000, 2500, 5000, 7500 and 10000. This way, we can evaluate the performance of both models as the dataset size increases. We used 2 validation levels to choose the best CNN architecture for each model. Then, we evaluated the performance of both models for each dataset size on 6 new test levels, averaging the results obtained across 15 repetitions. The training and test levels are different so as to measure the generalization abilities of the models for different problems (levels) of the same domain (game).

To be able to correctly assess the results obtained, we tried to obtain the shortest plan from the initial state of the game to the final goal $g_f$ for every test game. However, this proved to be an intractable problem for the Planner Module. For this reason, we used a model which selects each subgoal $g^*$ randomly, called the *Random model*. We obtained the average length (across 10 repetitions) of the plans the Random model used to complete each test level. Then, for each test level separately, we obtained the *action coefficients* of the Greedy and DQP models by dividing the average length of their plans by the ones obtained using the Random model. Finally, we averaged the action coefficients of each model across all the test levels. The final results can be seen on Figure 4.

As can be seen in the plot, the DQP model achieves an action coefficient of 0.58 for a dataset of 10000 samples. This is a notable result, specially taking into consideration that we have measured the performance on different levels from the ones used for training. Unlike our model, most RL techniques present bad generalization properties [27]. Our hypothesis is that our model generalizes better because, thanks to our goal-oriented approach, the complexity of the state space is reduced. Instead of using Deep Q-Learning to choose every single action, it is only used to select the next subgoal $g^*$, while relying on a planner for achieving $g^*$. Due to this, the complexity of the learning problem is reduced and overfitting is less likely to occur, thus improving the generalization properties of the model.

The plot shows how, up to 2500 samples, both models are almost in-

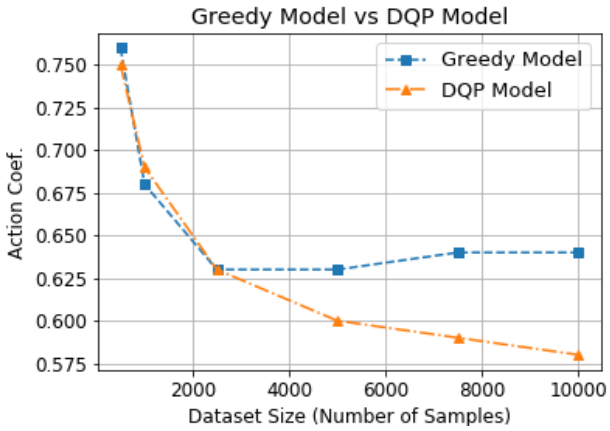


Figure 4: A plot comparing the average performance (*action coefficients*) across all test levels of the Greedy and DQP models. A low action coefficient implies a short plan and, thus, a good Goal Selection.

distinguishable in terms of performance. For bigger datasets however, the performance of the Greedy model remains more or less the same whereas the performance of the DQP model keeps improving. We believe this is because the Greedy model is a simple model that performs well on small datasets. However, as the dataset size increases, it arrives at a *plateau* since its strategy for selecting $g^*$ is suboptimal and, thus, a better, although more complex, model like the DQP one which performs look-ahead when selecting $g^*$ ends up outperforming it. This shows how important is long-term thinking when selecting the subgoals $g^*$.

To conclude, we saw the Planner Module spent 0.06 seconds on average to obtain each plan $p(s, g^*)$, which makes an average of 0.6 seconds per level. When we tried to use the Planner Module to obtain the optimum plan for each test level, it did not find a single plan after 16 hours of execution. This shows the inmense difference in terms of time consumption between searching for the plan that directly achieves the final goal $g_f$ and planning for each of the subgoals $g^*$.

# 7 Related Work

The use of Neural Networks (NN) in Automated Planning has been a topic of great interest in recent years. Some works have applied Deep Q-Learning to solve planning and scheduling problems as a substitute for online search algorithms. [20] uses Deep Q-Learning to solve the *ship stowage planning problem*, i.e., in which slot to place a set of containers so that the slot scheme satisfies a series of constraints and optimizes several objective functions at the same time. [17] also employs Deep Q-Learning, but this time to solve

the *lane changing problem*. In this problem, autonomous vehicles must automatically change lanes in order to avoid the traffic and get to the exit as quickly as possible. Here, Deep Q-Learning is only used to learn the long-term strategy, while relying on a low-level module to change between adjacent lanes without collisions. In our work, we also employ Deep Q-Learning but, instead of using it as a substitute for classical planning, we integrate it along with planning into our planning and acting architecture. Also, we do not focus on solving a specific problem but rather create an architecture which we hypothesize it is generalizable across a wide range of game domains.

There are other works which use neural networks to solve planning problems but, instead of relying on RL techniques such as Deep Q-Learning, train a NN so that it learns to perform an *explicit planning process*. [24] proposes a novel NN architecture known as *Action Schema Networks* (ASNet) which, as they explain in their work, *are specialised to the structure of planning problems much as Convolutional Neural Networks (CNN) are specialised to the structure of images*. [22] uses a CNN that performs the computations of the value-iteration (VI) planning algorithm [2, 3], thus making the planning process differentiable. This way, both works use NN architectures which *learn to plan*.

These NNs are trained on a set of training problems and evaluated on different problems of the same planning domain, showing better generalization abilities than most RL algorithms. [22] argues that this happens because, in order to generalize well, NNs need to learn an *explicit planning process*, which most RL techniques do not. Although our architecture does not learn to plan it does incorporate an off-the-shelf planner which performs explicit planning. We believe this is why our architecture shows good generalization abilities.

Neural networks have also been applied to other aspects of planning. For instance, [6] trains a NN that learns a planning domain just from visual observations, assuming that actions have *local* preconditions and effects. The learnt domain is generalizable across different problems of the same domain and, thus, can be used by a planner to solve these problems.

In this work, we have proposed an architecture which uses Goal Reasoning as the method for interleaving planning and acting. [11] proposes a Goal Reasoning architecture which uses Case-Based Reasoning [12] and Q-Learning in order to learn to detect discrepancies, associate discrepancies to new goals and learn policies that achieve the selected goals. Although our architecture only learns to select subgoals, the use of a NN (integrated into the Deep Q-Learning algorithm) instead of traditional Q-Learning gives our architecture the ability to generalize. For this reason, we believe our architecture scales better when applied to big state spaces than the one proposed by Jaidee et al..

[4] employs an architecture that does use a NN, concretely a CNN, to

select subgoals for navigating a maze in the game known as *Minecraft*. When a subgoal must be selected, the CNN receives an image of the current state of the game, which is used to decide the most suitable subgoal for that state. Unlike our work, a hard-coded expert procedure is used to teach the CNN which subgoal must be selected in each state. As Bonanno et al. recognise, this approach transforms the problem into a classification task, instead of a RL one. Furthermore, the set of eligible subgoals are always the same four regardless of the state of the game. In our work, the compound subgoal $G$ is different for each game state and can contain a different number of single subgoals $g \in G$ to choose from.

# 8 Conclusions and Future Work

We have proposed an architecture which learns to select goals with Deep Q-Learning in order to interleave planning and acting. We have tested our architecture on the GVGAI game known as Boulder Dash, using different levels for training and testing in order to measure its generalization abilities. We have shown how the DQP model outperforms the Greedy model, proving the importance of long-term thinking for selecting subgoals.

Our architecture obtains better results than most RL algorithms. It is more sample-efficient, only needing to be trained on thousands of samples rather than hundreds of thousands [23], and shows better generalization properties [27] when evaluated on different test levels from the ones used for training. However, our approach needs more domain-specific knowledge than standard RL techniques. In future work, we intend to reduce this knowledge gap by augmenting our architecture so that it also learns to formulate goals. In addition, we plan to test our architecture on different GVGAI games to prove it is also generalizable across different domains.

This Final Degree Project has been an extremely enriching experience for me. On the one hand, this has been the first "real" project I have been part of. I have learnt how to pursue a long-term goal and, at the same time, split it into manageable short-term goals. I have discovered that the only realistic way of achieving this is using an agile methodology, since it allows to adapt to new, unexpected situations that arise during the development of the project. On the other hand, this project has introduced me to the research world. I have gained important insight into how to conduct a scientific study and use the obtained results to publish a scientific paper. To sum up, this project has proved to be a valuable first work experience which I am sure will help me throughout my professional career.

## References

[1] David W Aha, Tory S Anderson, Benjamin Bengfort, Mark Burstein, Dan Cerys, Alexandra Coman, Michael T Cox, Dustin Dannenhauer, Michael W Floyd, Kellen Gillespie, et al. Goal reasoning: Papers from the acs workshop. Technical report, Georgia Institute of Technology, 2015.

[2] Richard E Bellman et al. Dynamic programming, ser. In *Rand Corporation research study*. Princeton University Press, 1957.

[3] Dimitri P Bertsekas. Dynamic programming and optimal control 4th edition, volume ii. *Athena Scientific*, 2015.

[4] David Bonanno, Mark Roberts, Leslie Smith, and David W Aha. Selecting subgoals using deep learning in minecraft: A preliminary report. In *IJCAI Workshop on Deep Learning for Artificial Intelligence*, 2016.

[5] Luis Castillo, Eva Armengol, Eva Onaindía, Laura Sebastiá, Jesús González-Boticario, Antonio Rodríguez, Susana Fernández, Juan D Arias, and Daniel Borrajo. Samap: An user-oriented adaptive system for planning tourist visits. *Expert Systems with Applications*, 34(2):1318–1332, 2008.

[6] Andrea Dittadi, Thomas Bolander, and Ole Winther. Learning to plan from raw data in grid-based games. In *GCAI*, pages 54–67, 2018.

[7] Juan Fdez-Olivares, Eva Onaindia, Luis Castillo, Jaume Jordán, and Juan Cózar. Personalized conciliation of clinical guidelines for comorbid patients through multi-agent planning. *Artificial intelligence in medicine*, 96:167–186, 2019.

[8] 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.

[9] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated planning and acting*. Cambridge University Press, 2016.

[10] Jörg Hoffmann. Ff: The fast-forward planning system. *AI magazine*, 22(3):57–57, 2001.

[11] Ulit Jaidee, Héctor Muñoz-Avila, and David W Aha. Learning and reusing goal-specific policies for goal-driven autonomy. In *International Conference on Case-Based Reasoning*, pages 182–195. Springer, 2012.

[12] Janet Kolodner. *Case-based reasoning*. Morgan Kaufmann, 2014.

[13] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[14] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

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

[16] 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):529–533, 2015.

[17] Mustafa Mukadam, Akansel Cosgun, Alireza Nakhaei, and Kikuo Fujimura. Tactical decision making for lane changing with deep reinforcement learning. 2017.

[18] Sunandita Patra, Malik Ghallab, Dana Nau, and Paolo Traverso. Acting and planning using operational models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 7691–7698, 2019.

[19] Diego Perez-Liebana, Spyridon Samothrakis, Julian Togelius, Tom Schaul, Simon M Lucas, Adrien Couëtoux, Jerry Lee, Chong-U Lim, and Tommy Thompson. The 2014 general video game playing competition. *IEEE Transactions on Computational Intelligence and AI in Games*, 8(3):229–243, 2015.

[20] Yifan Shen, Ning Zhao, Mengjue Xia, and Xueqiang Du. A deep q-learning network for ship stowage planning problem. *Polish Maritime Research*, 24(s3):102–109, 2017.

[21] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[22] Aviv Tamar, Yi Wu, Garrett Thomas, Sergey Levine, and Pieter Abbeel. Value iteration networks. In *Advances in Neural Information Processing Systems*, pages 2154–2162, 2016.

[23] Ruben Rodriguez Torrado, Philip Bontrager, Julian Togelius, Jialin Liu, and Diego Perez-Liebana. Deep reinforcement learning for general video game ai. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8. IEEE, 2018.

[24] Sam Toyer, Felipe Trevizan, Sylvie Thiébaux, and Lexing Xie. Action schema networks: Generalised policies with deep learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[25] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

[26] Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. 1989.

[27] Chiyuan Zhang, Oriol Vinyals, Remi Munos, and Samy Bengio. A study on overfitting in deep reinforcement learning. *arXiv preprint arXiv:1804.06893*, 2018.