Università degli Studi di Trieste

Dipartimento di Matematica e Geoscienze
Master degree in Data Science and Scientific Computing

DQN-based approaches for complex board games: investigating hierarchical strategies and graph-based representation in the Board game Settlers of Catan

Candidate:
Giacomo Ettore Rocco

Supervisor:
Prof. Eric Medvet

Co-supervisor:
Dott. Erica Salvato

ACADEMIC YEAR 2022-2023

Abstract

In Machine Learning, computer science, and data science, developing programs able to perform proficiently in video games or board games, has always been a way to put forward stronger Learning techniques. For example, the development of a computer engine that plays chess is old like computer science itself. Scenarios like games represent an easy-to-understand (most of the time) scenario with elements of different kinds interacting with each other. They have a clearly defined objective: get the win condition. They present themselves as a gym and a suitable territory for algorithm comparisons. This thesis is about Reinforcement Learning, in particular the investigation of possible different approaches of a specific algorithm called DQN (Deep Q-Network). This technique will be applied to a particular hard environment in terms of state representation, dynamics, and characteristics: a board game called Settlers of Catan. In this thesis, we employed three different types of neural networks involved in the DQN: feed-forward networks, graph convolutional networks, and relational graph convolutional networks. Our results suggest that this algorithm deserves attention and can learn even in environments with a vast stochastic component.

Contents

1 Introduction **4**
- 1.1 Motivations . . . 4
- 1.2 Thesis organization . . . 4
- 1.3 Introduction to Machine Learning . . . 5
  - 1.3.1 Supervised Learning . . . 5
  - 1.3.2 Unsupervised Learning . . . 5
  - 1.3.3 Reinforcement Learning . . . 5
  - 1.3.4 Key differences of those approaches . . . 6
- 1.4 Settlers of Catan . . . 7
  - 1.4.1 Board, resources, and dice . . . 7
  - 1.4.2 Further elements . . . 9
  - 1.4.3 Initial phase . . . 10
  - 1.4.4 Game phase . . . 10
- 1.5 Trades between player removed . . . 10
  - 1.5.1 How to win, and other rules . . . 10
- 1.6 AI in video games, related works . . . 11
  - 1.6.1 Graph Neural Network on Board-games . . . 11
  - 1.6.2 Hierarchical DQN . . . 11
  - 1.6.3 Self-Learning option . . . 11
- 1.7 Difficulties of our case . . . 12
  - 1.7.1 Stochastic environment . . . 12
  - 1.7.2 Hidden information . . . 13
  - 1.7.3 Overwhelming number of states . . . 13

2 Background **14**
- 2.1 Reinforcement Learning . . . 14
  - 2.1.1 Main elements . . . 14
  - 2.1.2 Markov decision processes . . . 14
  - 2.1.3 Exploration-exploitation dilemma . . . 16
- 2.2 Return value . . . 16
  - 2.2.1 Value funtion . . . 17
- 2.3 Epsilon-greedy . . . 17
  - 2.3.1 Epsilon Decay . . . 18
- 2.4 State space . . . 18
  - 2.4.1 Q-learning . . . 19
  - 2.4.2 Tabular Q-Learning . . . 20
  - 2.4.3 DQN, motivations . . . 21
  - 2.4.4 Small recap on Neural Networks . . . 22
- 2.5 DQN - In Practice . . . 22
  - 2.5.1 DQN Training . . . 23
- 2.6 DQN further elements . . . 24
  - 2.6.1 Experience replay . . . 24
  - 2.6.2 Target network . . . 24
  - 2.6.3 How does the training works . . . 25

2.6.4 Soft update . . . 25
2.6.5 DQN pseudocode . . . 26

3 Environment development **27**
3.1 Developed from scratch: motivations . . . 27
3.1.1 Settler of Catan as a benchmark . . . 27
3.2 Software architecture, brief summary . . . 28
3.2.1 Choices adopted . . . 28

4 State representation modeling **30**
4.0.1 Small Recap on Graphs . . . 30
4.1 Catan Board as a graph . . . 30
4.1.1 Graph Convolutional Networks . . . 30
4.1.2 Relational Convolutional Networks . . . 31

5 Proposed method **33**
5.1 Board features representation . . . 33
5.1.1 Global player features . . . 34
5.2 Net architectures . . . 35
5.2.1 Graph neural network architecture . . . 35
5.2.2 Feed forward net . . . 37
5.3 Action selection approaches . . . 38

6 Hierarchical design **40**
6.0.1 Reward choice . . . 40
6.1 Agents . . . 40
6.1.1 DQN Orchestrator . . . 41
6.1.2 DQN specialized . . . 41
6.1.3 DQN specialized with subgoals variation . . . 42

7 Training **44**
7.1 Training settings . . . 44
7.2 Training results . . . 45
7.3 Self-Learning . . . 46

8 Empirical competition **48**
8.1 Final ranking . . . 50

9 Conclusions and future works **51**

1 Introduction

1.1 Motivations

The aim of this thesis is to implement an algorithm capable of playing the board game Settlers of Catan. Despite its seemingly simple nature, Settlers of Catan remains a challenging game for AI due to several factors. These factors include a significant stochastic component, hidden information, the need for resource optimization, and the vast number of possible actions and strategies. In this study, we evaluate the performance and training capabilities of several agents based on Reinforcement Learning, specifically the DQN algorithm. The DQN was chosen because it has demonstrated success in various other games. We also explore the implementation of a hierarchical version of the DQN algorithm, using three types of neural networks: feed-forward networks, graph convolutional networks, and relational graph convolutional networks. Throughout this thesis, we present and analyze these agents, which differ in the type of neural networks utilized and the characteristics of the hierarchical DQN component.

1.2 Thesis organization

The thesis is structured to address various aspects of Reinforcement Learning and the application of the DQN algorithm in the context of the board game Settlers of Catan.

In the Section 1, an overview is provided on Machine Learning, explaining the different options available and their key differences. Subsequently, the focus shifts to the rules and dynamics of Settlers of Catan, highlighting the challenges it presents for artificial intelligence.

The Section 2 delves into the fundamental algorithms used in the research and provides the necessary theoretical foundations for understanding them.

Following that Section 3 is a brief summary motivating the development of the environment from scratch.

Next, Section 4 we introduce a Deep Learning (DL) approach, namely graph convolutional networks, we thought that it could fit properly our game case. We briefly explain the mechanics involved in representing the game state using this approach.

Furthermore, in Section 5 we describe our proposed method. A description of the DL architecture will be provided.

As well in Section 6 we illustrated the implementation of a hierarchical version of the DQN algorithm. A particular variation, involving the assignment of subgoals to subentities, is illustrated.

In Section 7 we present the training process and outcomes of different agents.

Then in Section 8, we simulated a tournament to empirically evaluate the performance of the agents and determine which one performs better among them.

Finally, Section 9 summarizes the findings of the research, discusses their implications, and suggests potential directions for future work.

1.3 Introduction to Machine Learning

Machine Learning is a branch of artificial intelligence, that involves the development of algorithms and models that enable computers to learn and improve from past experiences without being explicitly programmed. Instead of following specific instructions to solve a task, Machine Learning relies on data analysis and the identification of patterns and relationships to make autonomous decisions and improve performance over time. Within the field of Machine Learning, there are three main categories of approaches: Supervised Learning, Unsupervised Learning, and RL. These methodologies differ in terms of their goals, input data, and Learning techniques employed.

1.3.1 Supervised Learning

Supervised Learning (SL) is a form of Machine Learning where a model is trained on a labeled training dataset of elements. The label provided in this dataset is called "true labels", and we want our agent to learn the relation between data (also called observations) and labels (output or response). The type of tasks are mainly two: classification and regression. In classification, every label is categorical and referred to as a class, instead in regression every label is numerical and is referred to as value. In SL particular kind of Learning, the goal is to find relations between the input features and the true label. This allows making a program able to understand which are the characteristics or reasons that make an element belong to a particular class (in classification) or a particular value (in regression). We want our program able to label correctly elements that were not present in the training dataset. In other words, the model is given input examples associated with corresponding output labels and seeks to learn a function that maps inputs to desired outputs. Once trained, the model can be used to make predictions about the label on new data.

1.3.2 Unsupervised Learning

Unsupervised Learning (UL) is a Machine Learning approach that involves analyzing and exploring a dataset without the use of pre-labeled or pre-classified data. In UL, the algorithm aims to discover underlying patterns, structures, or relationships in the data on its own, without explicit guidance or supervision. The goal is to uncover hidden insights or knowledge that may be present in the dataset. For example, clustering algorithms are commonly used in UL. Clustering algorithms group similar data points together based on their inherent similarities, without any prior knowledge of the true labels. This helps in organizing the data into meaningful subsets, allowing us to understand the relationships between different data points.

1.3.3 Reinforcement Learning

Reinforcement Learning (RL) is a form of Machine Learning where an agent learns to make decisions in a dynamic environment through interactive Learning. Unlike SL or UL, which rely on labeled or unlabeled examples, RL does not require access to labeled data. This makes it particularly useful in situations where obtaining labeled data is unfeasible, impractical, expensive, or time-consuming. In RL, the agent learns effective

behavior through trial and error. It takes actions in specific states of the environment and receives feedback in the form of rewards or penalties. The objective of the agent is to maximize the cumulative reward it receives over time. Through repeated interactions with the environment, the agent learns which actions lead to higher rewards and adjusts its decision-making accordingly. This iterative process involves exploring different actions, observing outcomes, and updating knowledge through a feedback loop. RL has found success in diverse domains like robotics[8], game-playing[12], and autonomous systems [6]. It enables agents to learn from experience and adapt their behavior to achieve specific goals in complex and uncertain environments. In this thesis, RL will be the primary approach discussed in Chapter 2, where it will be explored in more detail.

1.3.4 Key differences of those approaches

The fundamental differences between RL, SL, and UL lie in several aspects. In SL, the algorithm learns from labeled examples where each input is associated with an output label. The goal is to generalize this mapping to make accurate predictions on unseen inputs. In contrast, UL operates on unlabeled data and aims to discover patterns, structures, or relationships within the data without explicit guidance. In RL, the agent learns through interaction with a dynamic environment. It observes the current state, takes actions based on it, and receives feedback in the form of rewards or punishments. This feedback guides the Learning process of the agent to maximize long-term cumulative rewards. While SL focuses on Learning a mapping between input features and output labels and UL focuses on discovering patterns, RL is goal-oriented. The agent learns to perform a specific task or achieve a particular objective. The Reinforcement signal guides the Learning process, encouraging the agent to take actions that maximize cumulative rewards. In RL, there is a time factor that emphasizes sequential decision-making and the optimization of long-term goals. The agent's actions at each time step can have consequences that unfold over multiple subsequent time steps, requiring careful consideration of the temporal aspect of the problem. By incorporating the notion of time, RL algorithms can learn policies that balance immediate rewards with future potential gains, enabling the agent to make informed decisions that maximize cumulative rewards over extended periods. It emphasizes sequential decision-making and the optimization of long-term goals. While Supervised and UL often handle individual data instances independently, RL deals with sequential decision-making. RL agents make decisions in a sequential manner, considering past actions and states. The consequences of an action in RL may not be immediately apparent, as feedback or rewards can be delayed until subsequent states. In SL and UL, the temporal aspect of actions and delayed feedback is typically not considered. RL involves interactive Learning, sequential decision-making, delayed feedback, and the trade-off between exploration and exploitation. Through these aspects, RL enables agents to learn effective policies through trial-and-error interactions with their environment, leading to significant advancements in various domains, for example in a situation of a scenario where the strategy is unknown and there is a clear task to pursue, like in a Board game.

1.4 Settlers of Catan

This thesis is an application of the algorithm DQN, which will be explained later. For obvious reasons, explaining the game must be done to give an idea of the complexity and the challenges involved in the task of building an agent able to play proficiently in this game.

Settlers of Catan, designed by Klaus Teuber in 1995, is a multiplayer strategic board game that revolves around resource management, trading, and territorial expansion. Every player takes the role of a settler inhabiting an island. The game is played on a modular hexagonal game board consisting of various terrain types and numbered tokens. The objective of the game is to accumulate victory points mainly by building settlements, cities, and roads.

1.4.1 Board, resources, and dice

At the beginning of the game, the game board is set up by placing hexagonal terrain tiles randomly, referred to simply as tiles, each representing a specific resource:

- Wood
- Brick, by some called also Clay
- Wheat, by some called also Crop
- Sheep
- Ore, by some called also Iron

Those resources are kept in an element of the game called *Bank*, which contains 19 resources per type, and this is also called **bank limit**, which implies that the maximum number of every resource type in the game is 19. Example: can not exist a situation where the sum of the iron owned by players + iron in the bank is greater than 19.

Each *tile* is marked randomly with a number $n \in [2, 12]$, which are the possible results of the roll of two six-sided dice. Because, every turn two six-sided dice are rolled, and every tile marked with the result of the roll produces a resource, if and only if "the robber" is not placed on it (the robber will be explained later).

A *place* can be every possible vertex of every possible tile as shown in Figure 2, a place can touch one, two, or three tiles. The tiles touched by a place are referred to by *touched tiles*.

The resources are a fundamental component of this game because a player need to make decisions in order to try to use them as best he can. With resources, you can buy the following things:

- Colony: Each player can construct colonies in Places. The owner of the colony receives the resources produced by every tile touched by that place. Building a colony, besides the fact that makes a player gain more resources, gives him 1 victory point.

- City: Each colony can be upgraded in City. Every city produces the double of resources of a colony and gives the player an additional victory point.
- Street: The streets can be places connecting 2 places, and a player can build a colony in a place only if there one of his streets that reaches that place.
- Development Card: a player can draw a development card, and it can give you different effects as will be explained later.

In Figure 1 there is a summary of the costs.

Figure 1: Costs summary

Each player can trade with the bank.

The *standard trade* is: exchanging 4 resources of the same type and obtaining a resource of any chosen type.

Harbors are a particular kind of place. If you own a colony or city in one of these special places, you have discounted trades with the bank.

The harbors are of the following variants:

- 3:1, you can give 3 resources instead of 4 in your standard trades.
- 2:1 *type*-resource, you can give 2 *type*-resources and obtaining a resource of any chosen type

A player can build colonies only in places reached by his own streets (except for the two initial colonies), also every colony must be built only in places that do not have adjacent places with an already existing colony.

The robber is a token of the game, which is moved by players and it denies a particular tile to produce, it starts the game in a tile called the desert, as shown in the figure Figure 2.

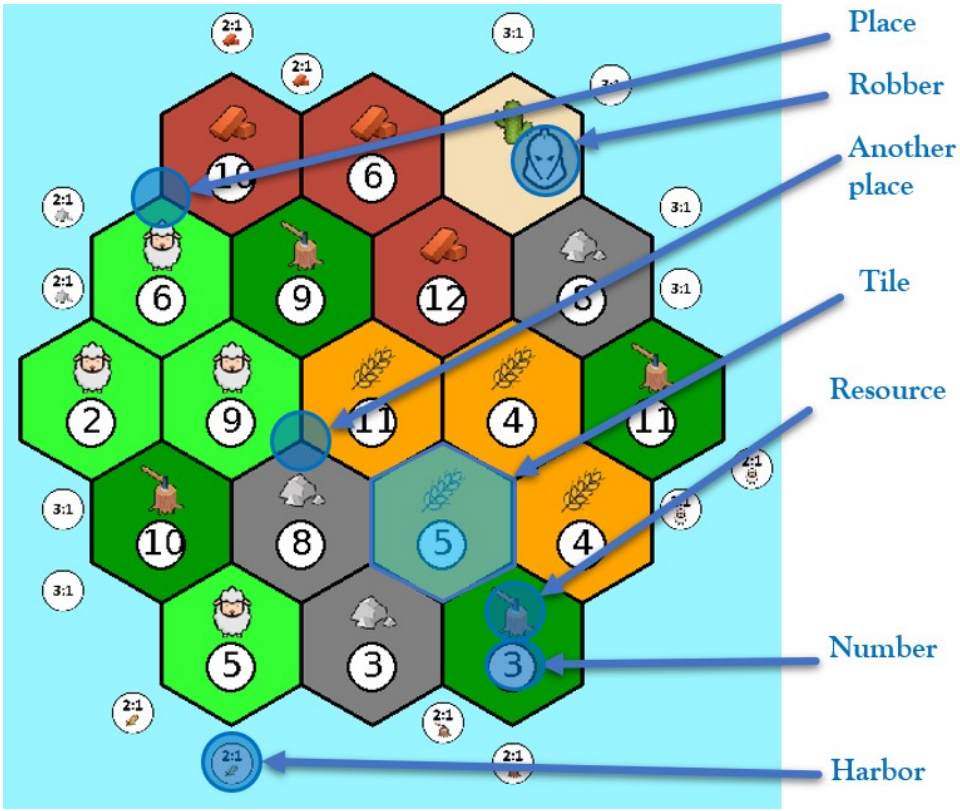

Figure 2: Example of a possible configuration of the tiles

There is an exception when the dice are rolled: the result 7. Tree things happen:

1. Every player that has 7 or more cards must discard half of them rounded up.
2. The in-turn player must move the robber to another tile.
3. The in-turn player must steal a resource from a player chosen among the players with a city or a colony on the places of that tile.

1.4.2 Further elements

The last element of the game is the development cards: they are of 5 different types:

- "victory point card": it gives +1 victory point
- "knight card": it allows you to move the robber in a tile and steal a resource from a player chosen from one of the players who touch it.
- "monopoly card": it allows you to declare a resource, and every player must give you every resource of that type that they have.
- "road building": it allows you to construct two free streets.
- "year of plenty": it allows you to obtain two free resources chosen by you.

1.4.3 Initial phase

Before the game starts, every player starting from the first, clockwise, collocates a colony and a city on the board. And this process is done two times. To overcome the advantage related to the order of the players, the second time the order to collocate the colony is reversed. After this phase, which is called the *initialphase*, the game starts.

1.4.4 Game phase

Players take turns clockwise, with each turn comprising three distinct phases: the phase before the roll, the roll phase, and the development phase.

- Phase before roll: in this particular phase the player in turn can just perform an action: play a knight card.
- Roll phase: The active player rolls two six-sided dice, determining which tile produces resources for all players. Any colonies or cities built adjacent to these tiles receive corresponding resources based on the terrain type.
- Development phase: The active player can perform various actions during this phase until he desires and it has the needed resources, the actions are: trade, build, use a development card, and buy a development card.

1.5 Trades between player removed

In the original Settlers of Catan game, one significant aspect is the ability for players to engage in trading resources with each other.

However, for the purpose of this research, the complexity associated with negotiating and implementing trading interactions between intelligent agents has been omitted, but there are several interesting works on that part of Settlers of Catan [15].

We can just state that AI never proposes trades and never accepts trades proposed, this can be seen as a forced programmed part of the *policy*, not as a simplification, because it is not violating the rules of the game. By excluding player-to-player trading, the focus can be directed towards evaluating the effectiveness of different *state* representations and decision-making strategies.

1.5.1 How to win, and other rules

Every colony built gives you 1 victory point. Every colony build can be upgraded in the city build, it gives 1 additional victory point and it produces double resources. Every colony must be distanced 2 streets apart from any other colony. If you want to construct another colony, you have to construct at least 2 streets starting from one of your colonies.

Additionally, Settlers of Catan incorporates an element of strategy through the concept of the longest road and the largest army.

Players gain 2 additional victory points for achieving the longest continuous road, which means a path of consecutive streets without passing 2 times for the same edge. Players

gain 2 additional victory points for playing the most knight development cards, respectively. The game continues until a player reaches the required number of victory points, typically 10 for 4 players or 15 for 2 players, to be declared the winner. Settlers of Catan offers a dynamic and strategic environment, requiring players to adapt to changing resource availability, and trades, and plan their moves strategically to outmaneuver opponents and achieve victory.

As seen in this chapter, the game has a lot of rules and a lot of elements, in order to simulate the whole mechanics of the game we developed an environment from scratch, as briefly explained and motivated in the Chapter "Environment development".

1.6 AI in video games, related works

In this section, we just briefly mention the inspirational works that we were inspired by to perform our GCN architecture, which even if is not the central topic of this thesis, is where all the ideas came from.

1.6.1 Graph Neural Network on Board-games

Graph neural networks have been utilized in various research, and the works that inspired our thesis were focused on the application of graph convolutional networks in the game of Risk [2] [1]. Those research showcased the potential of graph neural networks in capturing and leveraging the relational information present in board games. By incorporating the underlying graph structure of the game, the agents were able to make informed decisions and improve their gameplay strategies.

1.6.2 Hierarchical DQN

As mentioned before this thesis use an algorithm called DQN, we implemented a hierarchical approach inspired by [9]. Even if the idea is not exactly the same and they approach hierarchical DQN probably in a more effective way, the idea of the subgoal has been inspired by them.

1.6.3 Self-Learning option

In an ideal multi-agent scenario, such as Chess or Go, incredible success has been reached in the context of building agents able to outperform humans [13]. The environment exhibits fixed transaction dynamics, which means that a transaction leads you every time in the same state and there are no cases where a transaction can lead in two different states.

Consequently, the quality of an action is directly influenced by these dynamics and the potential actions taken by other agents. This environment presents also perfect information, which means that a player knows every element in the environment and there is no hidden information. In this context, the objective of the agent is to assess the value of an action, considering all possible responses from other agents, in order to choose the action that leads to the most favorable state, given the possible response of an adversarial agent. This decision-making process relies on the understanding and interpretation of the agent,

which is often incorporated into approaches like self-Learning, where the agent assumes its own response to each action. This approach avoids situations where the action only works against optimal opponents but fails against weaker ones. An analogy to illustrate this concept is a game of chess: if a move proves successful against a strong opponent, it should also be effective against a weaker opponent, ensuring consistent performance regardless of the skill level of the opponent.

However, this reasoning assumes an ideal scenario, such as a chess board, where there are no stochastic components introduced by the environment, in the following section we introduce all the difficulties concerning our specific environment.

Self-Learning is an approach that has been considered in this thesis, and a shy attempt has been trained and used, in the next section we explain the main reason that lead us to let self-Learning be made a concrete manner for future works.

1.7 Difficulties of our case

1.7.1 Stochastic environment

When an environment has stochastic components, like in a game such as Settlers of Catan, the effectiveness of taking a specific action from a given *state* is not always guaranteed to be the same. This introduces additional challenges in the RL process.

When considering the best approach for our study, we made the decision not to delve deeply into self-Learning methodologies. Instead, we opted to train our agents against random policies.

The reason for this choice was to avoid potential complications and challenges associated with self-Learning in the context of our research. While this approach can be powerful, it becomes particularly complex in environments with stochastic components, where the effectiveness of actions from a given state may vary.

In stochastic environments, the outcomes of actions can be uncertain or influenced by random factors. This introduces additional complexities and requires the agent to adapt its Learning strategy accordingly. By training our agent against random policies, we were able to evaluate its performance and assess its ability to learn and improve over time in a controlled and well-understood setting.

While self-Learning in stochastic environments is an intriguing research direction, exploring its intricacies would have required more extensive resources and time. We believed that this work was a necessary step in order to move in that direction. By focusing on training against random policies, we were able to gain valuable insights and results that contribute to our understanding of RL without introducing unnecessary complexities. It is worth noting that future research endeavors could explore the challenges and potential benefits of self-Learning in stochastic environments, as it remains an active area of investigation in the field of RL. The agent cannot predict the outcome of its actions with certainty, even in the same state, as the responses of adversarial agents will also depend on stochastic factors. This uncertainty makes it more difficult for the agent to determine which actions are optimal to take.

Moreover, exploration becomes more challenging in a stochastic environment. The agent needs to explore different actions and their outcomes or responses to gain a better understanding of the probabilities and distribution of results. This exploration is necessary to gather information and make informed decisions in the face of uncertainty. Due to the stochastic nature of the environment, training in a stochastic environment requires more data and a longer training period.

1.7.2 Hidden information

Another element that represents a challenge in this Settlers of Catan scenario is the fact that there is hidden information, a player cannot know which are the development cards of the opponents or their resources. Other works on board games with those characteristic has been made, they did not outperform humans but they are able to play at an expert level, for example in [11].

1.7.3 Overwhelming number of states

Another complexity beyond the random components is that in a game like Settlers of Catan, the number of possible initial board configurations alone is incredibly large, with more than billions of ways to put the tiles, and we are not even considering the numbers in the tiles, and not even the harbors. And furthermore, if a configuration is chosen, to give an idea, just in the initial move, as explained in Section 1.4.3 presents 54 possible spots for placing the first colony, followed by 3 potential street options. Then also for the second player. Subsequently, there are another 50 options for the second colony, each with 3 possible street choices. This complexity arises even before the game phase begins.

2 Background

In order to explain what has been done in this thesis, some elements of background must be at least introduced to give an idea about the reasoning and algorithms applied.

2.1 Reinforcement Learning

Reinforcement Learning is a strong learning algorithm that learns the optimal policy through interaction with the environment without the model of the environment [7]. As said in the introduction, this thesis is mainly about RL. RL is a powerful Machine Learning paradigm used to tackle a wide range of tasks. In this framework, the program acts like an agent which interacts by decision-making with an environment (which can be a simulated environment, as in our case). The environment responds to its actions by presenting new situations and giving rewards to the agents, as shown in Figure 2.

2.1.1 Main elements

- Action: First of all, in RL an *action* is everything that involves the decision-making of the agent (also "do nothing" or "pass" could be valid *action*s).
- State: Each moment or situation in the environment, where the agent must make decisions on which action to take, is referred to as a *state*. A *state* is typically described by variables that can be of various types and represents the modeling of the characteristics of that *state*, hopefully, these characteristics should contain all the information useful to understand the environmental situation and what is going on.
- Reward: given a *state*, the agent performs an available *action* and receives feedback in the form of a value, called *reward*. *Rewards* are often determined by quantifiable and meaningful information, such as the score in video games or the speed achieved in a locomotion task.
- Value function: The objective of the agent is not merely to obtain the highest immediate *reward*, but rather to maximize the cumulative future *reward*. This cumulative future *reward* is the sum of all *rewards* obtained from the current *state* onwards and is also known as the *value* or *quality value* or return.
- Policy: A *policy* defines how the agent behaves at a given state, the best *policy* is the one that maximizes the cumulative reward, and of course, it is our ideal aim. In some cases, a *policy* may be a simple function or a table that maps state to actions, in many cases, the aim would be an approximation of it.

2.1.2 Markov decision processes

If we want to use a Machine Learning method we have to describe formally the environment. In the sense that we want to give a general hypothesis on its properties. In RL it's usually assumed that the environment can be described as Markov decision process (MDP). It is formally defined as follows:

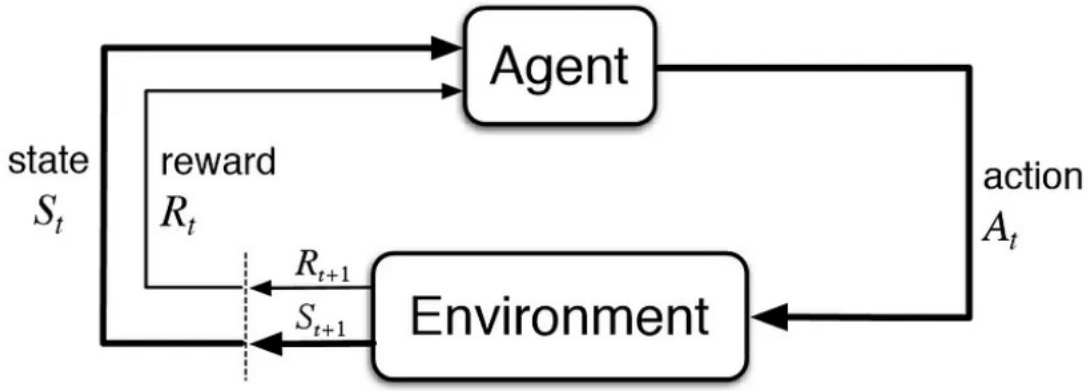

Figure 3: Illustration of Reinforcement Learning mechanism

- let S be a finite set of states s
- let A be a finite set of actions a
- Let $T : S \times A \to \Pi(S)$ be a function which we call transaction function, that gives to every state-action pair a probability distribution on S.
- Let $R : S \times A \times S \to \mathbb{R}$ a function that we call reward function, that assigns a number to every possible transaction.

There are discrete time steps. The interaction state-environment happens in the following way: at time t the agent receives the state as $s_t \in S$. Based on s_t it decides to execute $a \in A$, and the environment responds giving the agents an immediate reward $r_{t+1} = r(s_t, a_t)$ and producing next state $s_{t+1} = \delta(s_t, a_t)$. The functions δ and r are part of the environment and they are not necessarily known by the agent (and they could be non-deterministic). In RL an environment (a problem) satisfies the Markov property when:

$$P[s_{t+1} = s, r_{t+1} = r|s_t, a_t] = P[s_{t+1} = s, r_{t+1} = r|s_t, a_t, r_t, \ldots, r_1, s_0, a_0] \tag{1}$$

This implies that everything that happens at the time t+1, i.e., (s_{t+1}, r_{t+1}), depends only on what happens to the previous instant i.e., (s_t, r_t) and not by anything before i.e., $(s_t, a_t, r_t, ..., r_1, s_0, a_0)$.

If a problem satisfies the Markov property has a dynamic that can be described as a "step dynamic": it means that can be treated as episodic, where an episode is an attempt to reach the goal of the problem.

The Markov property is really crucial and fundamental in RL because every method in this kind of learning is based on the fact that the choices are based on the current state and on the action taken in the previous state.

2.1.3 Exploration-exploitation dilemma

A crucial concept in RL is exploration-exploitation. The agent needs to balance between *exploring* new states it has not encountered before and *exploiting* the knowledge it has gained so far. To obtain a lot of *reward*, an RL agent must prefer actions that it has tried in the past and found to be effective in producing *reward*. To draw a real-world analogy, it is like trying different types of food to accurately determine your preferred choice. The agent has to *exploit* what it has already experienced in order to obtain *reward*, but it also has to *explore* in order to make better *action* selections in the future. The dilemma is that neither *exploration* nor *exploitation* can be pursued exclusively without failing the task. The agent must try a variety of *actions* and progressively favor those that appear to be best. On a stochastic task (like ours), each *action* must be tried many times to gain a reliable estimate of its expected *reward*. This dilemma has been studied by mathematicians for many decades, yet remains unresolved [14].

2.2 Return value

As we mentioned before, rewards are localized in time, and there exists a function, called return, denoted as G, that represents our expectations of rewards in the long run.

The result of this function is also known as the return, which is the cumulative reward obtained during a particular episode. We have two different forms: the infinite horizon discount return and the finite horizon return.

G is defined as the cumulative reward from a given state onwards:

$$G_t \coloneqq r_t + r_{t+1} + r_{t+2} + \ldots + r_T \tag{2}$$

If we introduce the discount factor, γ, which represents our uncertainty about the future, we can write:

$$G_t \coloneqq r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots + \gamma^T r_T \tag{3}$$

By definition, we can express it recursively as:

$$G_t \coloneqq r_t + \gamma G_{t+1} \tag{4}$$

The infinite horizon discount return is defined as follows:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{5}$$

In this form, we consider rewards infinitely far into the future, but we discount each future reward using a discount factor, γ.

Alternatively, we have the finite horizon return:

$$G_t = \sum_{k=0}^{T-t-1} \gamma^k r_{t+k+1} \tag{6}$$

Here, we set a specific number of steps into the future (T) without applying any discounting.

2.2.1 Value funtion

In order to decide which action to take in a time t, it is important for an agent knowing how "good", it to be in a certain state. A way to measure this quality is the value function. It is defined as an expected reward sum. (E^π) that the agent will receive while following a determined policy π starting from a particular state s. The value function $V^\pi(s)$ for the policy π is given by:

$$V\pi(s) = E\pi(G_t \,|\, s_t = s) = E\pi\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k} \,|\, s_t = s\right) \tag{7}$$

Similarly, the action-value function also called *Q-function*, can be defined as the sum of the rewards expected while taking an action a in a state s following the policy π. The action-value function $Q^\pi(s, a)$ is defined as follows:

$$Q\pi(s, a) = E_\pi(G_t \,|\, s_t = s, a_t = a) = E_\pi\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k} \,|\, s_t = s, a_t = a\right) \tag{8}$$

2.3 Epsilon-greedy

In RL, ϵ is a parameter that controls the balance between exploration and exploitation. A high value of epsilon encourages more exploration, where the agent takes random actions to gather information about the environment. On the other hand, a low value of epsilon promotes exploitation, where the agent chooses actions based on its current knowledge and maximizes its expected rewards.

The epsilon-greedy strategy is a widely used exploration-exploitation algorithm in RL. It allows an agent to make decisions based on a trade-off between exploring new actions and exploiting the actions with the highest estimated value.

In the epsilon-greedy strategy, the agent maintains an estimate of the value of each action based on its past experiences. When choosing an action, the agent compares a randomly generated number between 0 and 1 with a parameter called epsilon.

If the generated number is less than epsilon, the agent chooses to explore. In this case, the agent selects a random action uniformly from the set of all possible actions. This exploration step allows the agent to gather information about different actions and their potential values. On the other hand, if the generated number is greater than or equal to epsilon, the agent chooses to exploit. In this case, the agent selects the action with the highest estimated value based on its current knowledge.

Exploitation allows the agent to select actions that have shown promising results in the past. By adjusting the value of epsilon, the agent can control the balance between exploration and exploitation.

The epsilon-greedy strategy is a simple and effective approach to address the exploration-exploitation trade-off in RL. It allows the agent to discover new actions while also taking advantage of the actions it has already learned to be valuable. This strategy is widely used in various RL algorithms to enable agents to learn and improve their decision-making capabilities over time.

2.3.1 Epsilon Decay

Epsilon decay, also known as epsilon annealing or epsilon scheduling, refers to the gradual reduction of the exploration parameter, epsilon, over time in RL algorithms. Epsilon decay is commonly used to shift the behavior of the agent from exploration to exploitation as it gains more experience and knowledge about the environment. The idea behind epsilon decay is that initially when the agent has limited knowledge, a higher value of epsilon helps in exploring the state-action space and discovering potentially valuable actions. However, as the agent learns and accumulates experience, it becomes more beneficial to exploit the learned knowledge and focus on actions with higher expected rewards.

By gradually decreasing epsilon over time, the agent becomes more selective in its exploration and gradually shifts towards exploitation. This allows the agent to make more optimal decisions based on the knowledge it has acquired. There are different ways to implement epsilon decay. One common approach is to linearly decrease epsilon over a specified number of steps or episodes. For example, you can start with a high epsilon value (e.g., 1.0) and decrease it by a small amount (e.g., 0.1) after each episode until it reaches a minimum value (e.g., 0.1). Another approach is to use a decay function, such as exponential decay or logarithmic decay, to gradually reduce epsilon over time.

The choice of the epsilon decay schedule depends on the specific problem and Learning dynamics. It is often determined through experimentation and fine-tuning to strike a balance between exploration and exploitation that leads to optimal Learning performance. Epsilon decay is a useful technique in RL as it allows the agent to explore the environment effectively in the early stages and then exploit the acquired knowledge to make more informed decisions. By adapting epsilon over time, the agent can improve its Learning efficiency and convergence towards an optimal policy.

2.4 State space

RL relies heavily on the concept of state, as input to the *policy* and value function, and as both input and output from the environment, we can think about a state as a signal conveying to the agent some sense of "how the environment is" at a particular time.

In RL, tasks can vary greatly in terms of the number of states they possess. Some tasks may have a vast or even infinite number of states, while others have a relatively small number. The number of states in a task refers to the different possible configurations

or situations that the agent can encounter during its interaction with the environment. When dealing with tasks that have a large or infinite number of states, the complexity of the problem can increase significantly. The agent needs to explore and learn from a vast *state* space, which can be computationally demanding and time-consuming. Additionally, the agent must generalize its knowledge across numerous states to make effective decisions. On the other hand, tasks with a small number of states are often simpler to handle. The agent can more easily explore and learn from the limited *state* space, and decision-making becomes relatively less complex.

However, it is important to note that even in tasks with a small number of states, other factors such as the action space, reward structure, and dynamics of the environment can still pose challenges to the agent. Researchers and practitioners encounter both scenarios-tasks with a high number of states and tasks with a low number of states. Understanding how to effectively tackle these different types of tasks and adapt the Learning algorithms accordingly is crucial for successful RL applications in various domains.

2.4.1 Q-learning

The vast number of states poses a significant challenge for RL algorithms. Traditional approaches like Q-Learning, which rely on explicitly storing and updating values for each state-action pair, become infeasible in such a complex and expansive *state* space. To address this challenge, algorithms like Deep Q-Network (DQN) have emerged. DQN utilizes neural networks to approximate the value function and effectively handle high-dimensional *state* spaces. It has shown promising in handling the immense complexity of games with an enormous *state* space, providing a viable approach for Learning optimal strategies in such environments. Before diving into the powerful Deep Q-Network (DQN) algorithm, let us start with its foundation: Q-Learning. Q-Learning is a popular RL technique often used for tasks with a small number of states.

Q-Learning [16] is a popular RL Value-based algorithm, one of the most applied representative RL approaches, and one of the off-policy strategies. The aim of this algorithm is going to estimate the Q-value state-action. It has the assumption of a discrete action space. It defines a deterministic policy based on value function:

$$\pi(s) = argmax_a Q(s, a) \tag{9}$$

It employs a method called temporal difference Learning (TD) to learn the Q-value function:

$$Q(s_t, a_t) := Q(s_t, a_t) + \lambda \cdot \underbrace{\left[\overbrace{r_t + \gamma \cdot \max_a Q(s_{t+1}, a)}^{\text{TD target}} - Q(s_t, a_t)\right]}_{\text{TD residual}} \tag{10}$$

If we recognize that we start with a random Q-estimate, we need some way to improve it given our experience interacting with the environment. This improvement happens whenever we receive a reward signal. Essentially, the reward signal gives us some idea

about how valuable the current state is. However, the reward is not the only component in the value of that state; there is also the subsequent state that we are going to move into, which also has a value.

So, this form represents the TD target component in the formula above. r_t is the current reward, and γ is the discount factor for the estimate of the subsequent state. We argue that given the current reward, this is a good estimate for the value of our current state. However, we should not forget what we have already learned, which is already encoded in our Q-value function. Therefore, this estimate will be a noisy estimate of the Q-value function.

To update our current estimate value, we form the TD residual by taking the difference between our current estimate of the value and the TD target. We then update our current estimate value by multiplying this difference by a Learning rate, α. The Learning rate can vary between zero and one. If it is zero, there will be no Learning; if it is one, we will always overwrite our estimate of the Q-value with our new information. Typically, it is set to a small value close to zero. The idea is that as we iterate over all these episodes and interact with our environment, the Q-value estimate is going to improve. As a result, our policy will behave better, and we will have a higher chance of reaching the goal.

2.4.2 Tabular Q-Learning

The simplest form of Q-Learning is tabular Q-Learning (TQL). In this approach, we estimate the Q-values by storing them individually in a table, an example is Table 1. The table has rows representing states and columns representing possible actions. For each state, we have a value for every possible action that can be taken in that state.

By using the TQL method, we update the Q-values based on the observed rewards and the estimated future rewards. During the Learning process, the agent interacts with the environment, receives rewards, and updates the Q-values accordingly. The update rule is based on the TD Learning method, where we update the Q-value of a state-action pair using the observed reward and the maximum Q-value of the next state as explained previously. The TQL algorithm iteratively updates the Q-values until convergence. As the agent explores different state-action pairs and receives feedback from the environment, the Q-values gradually converge to their true values.

The advantage of TQL is its simplicity and interpretability. However, it is limited to problems with a small number of states and actions, as the size of the Q-table grows with the number of states and actions.

Table 1: An example of Q-table

Stato (S)	**Azione (A)**			
	Azione 1	**Azione 2**	**Azione 3**	**Azione 4**
Stato 1	Q(1,1)	Q(1,2)	Q(1,3)	Q(1,4)
Stato 2	Q(2,1)	Q(2,2)	Q(2,3)	Q(2,4)
Stato 3	Q(3,1)	Q(3,2)	Q(3,3)	Q(3,4)

2.4.3 DQN, motivations

In TQL, we use a table to store the Q-values for our (state, action) pairs. There is an entry in the table for each state-action pair. However, a limitation of TQL arises when dealing with large state and action spaces. For example, in a scenario with 1,000,000 states and 100 possible actions per state, we would need to maintain a table with millions of entries.

To overcome this limitation, Deep Q-Learning Networks (DQN) [10] was introduced. Instead of Learning a function that directly maps states and actions to their corresponding Q-values, DQN employs deep neural networks as the mapping function. This allows us to approximate the Q-values using the weights of the neural network, significantly reducing the memory requirements compared to maintaining a large table.

One of the significant advantages of DQN over traditional Q-Learning is its ability to handle unseen states. In Q-Learning, if a state has not been encountered before, we randomly select an action. This is a drawback of TQL. However, with DQN, we can still apply the Q-network to unseen states. If the Q-network is well-trained, it can generalize to these states based on their similarity to previously encountered states.

Furthermore, DQN allows us to handle states that do not naturally fit into a table. For example, when DQN was initially proposed to solve video games like Atari, it processed the game states as images using a convolutional neural network (CNN). Traditional Q-Learning would struggle with this representation since it is not clear how to save images in a table. However, DQN leverages the power of neural networks, building upon the success of CNNs in image processing. This enables us to handle much more complex states, including previously unseen states, and benefit from the feature extraction capabilities of the neural network.

In summary, DQN addresses the limitations of TQL by using deep neural networks to approximate Q-values. It allows us to handle large state and action spaces, generalizes to unseen states, and process complex state representations such as images. The neural network acts as a feature extractor, enabling us to perform well on unseen states based on their similarity to previously encountered states.

2.4.4 Small recap on Neural Networks

For the sake of completion, it is worth mentioning several concepts about neural networks. Information flows from left to right via nodes (neurons) and connections.

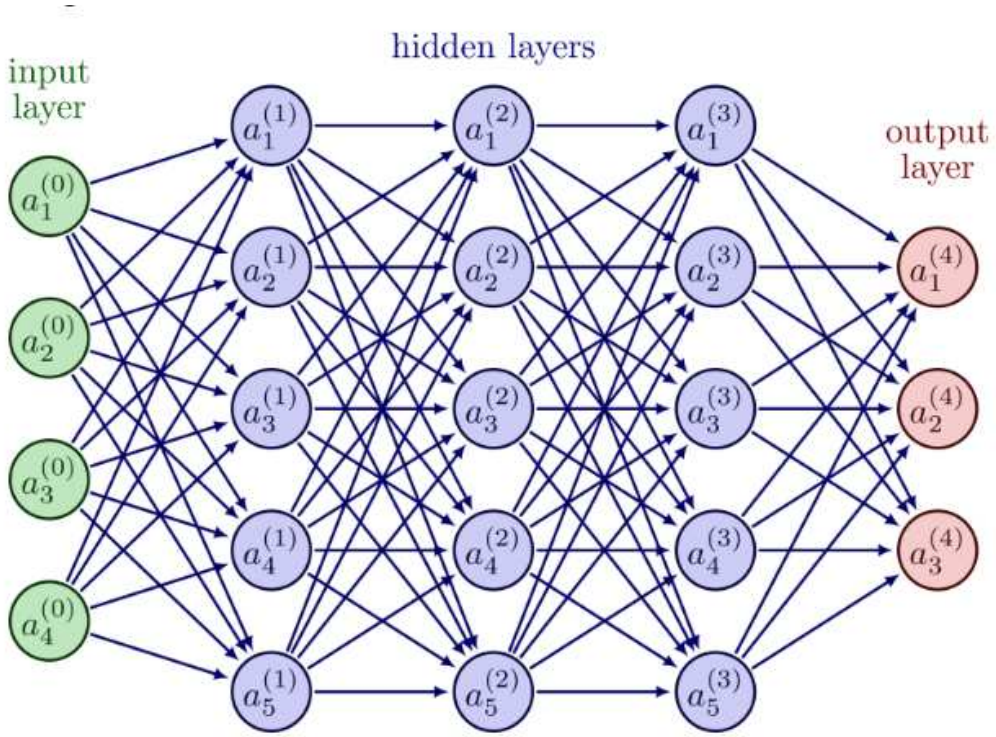

Figure 4: Example of a neural network

The output of a node is computed as a weighted sum of its input which is then passed through a non-linearity function to the following layer.

When trained with enough data can represent a wide variety of functions. The weights in the networks are trained to minimize a loss function on its output. This is done via gradient descent.

2.5 DQN - In Practice

In practice, instead of using our function to estimate $Q(s, a)$ for a single state-value pair, we instead have the function estimate Q(s, a) for all actions of a given state.

$$f(s) = [Q(s, a_1), ..., Q(s, a_n)] \tag{11}$$

This allows us to call our network f only once per state, instead of once per every state-action pair, it takes advantage of this parallelism.

- During inference we perform the action a_i corresponding to the maximum $Q(s, a_i)$ in this vector. If there are multiple maxima, we randomly break ties.

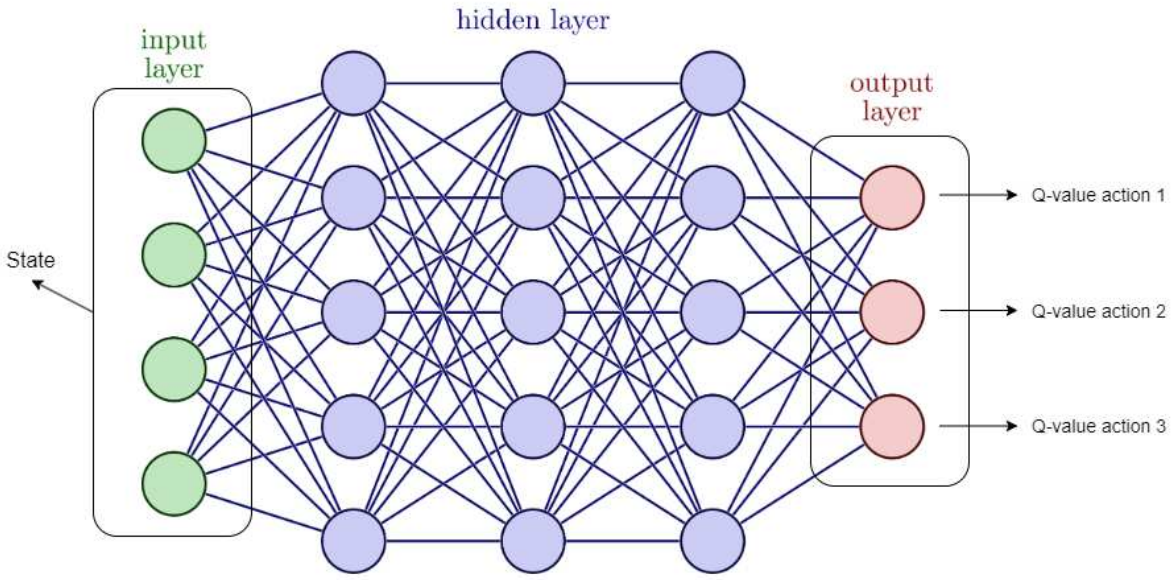

Figure 5: Illustration of a DQN

- During training we use an ϵ-greedy algorithm to balance exploration and exploitation. With probability ϵ select an action at random, and with probability (1-ϵ) select the optimal action (as during inference).

2.5.1 DQN Training

The deep neural network, known as the Q-network, is initialized with random weights. This network takes the current state as input and outputs estimates of the action values (Q-values) for all possible actions.

To train our DQN, we use temporal-difference training Equation (10).

The agent interacts with the environment by taking actions based on an exploration policy, such as epsilon-greedy. During these interactions, the agent collects experiences, which are the state-action-reward-next state transitions as tuples (s_t, a_t, r_t, s_{t+1}) observed during the interaction.

The experiences are stored in a buffer called the replay memory, with a technique called experience replay, as explained in the next section. This memory allows the agent to maintain a collection of past experiences that can be used for future training. Experience replay helps to decorrelate the training data and break the sequential correlation between experiences.

During training, in every step, a mini-batch of experiences is randomly sampled from the replay memory. Using mini-batches allows for efficient training, reducing gradient variance, and leveraging parallelization in computation libraries.

For each experience in the mini-batch, target labels for Q-network training are computed. The target label for the selected action is the sum of the observed reward and the max-

imum Q-value among possible actions in the next state, weighted by a discount factor gamma. In formula, the target label for action a_t in state s_t is given by:

$$Y_t = r_t + \gamma \max_{a'} Q(s_{t+1}, a')$$

Using a loss function between the computed target labels and the estimated action values from the Q-network, the loss is calculated. The Q-network is then trained using optimization algorithms such as gradient descent to minimize the loss and update weights.

To make the training more stable, a second network called the target network is used as explained in the next section. This network is used to compute target labels during training, but its weights are updated more slowly compared to the main Q-network. Periodically, the weights of the Q-network are copied to the target network to stabilize the Learning process. During training, the agent gradually improves its estimates of the action values and refines its action selection policy.

Training in DQN may require a significant number of iterations to converge to a good approximation of the action values. However, with the use of past experiences and deep neural networks, DQN is able to learn effective strategies even in complex environments with large action spaces.

2.6 DQN further elements

2.6.1 Experience replay

Experience replay [17] is a technique that decouples the updates of a model from the stream of experiences gathered during interaction with the environment. The idea is to save transitions, state-action-reward-next state tuples (s_t, a_t, r_t, s_{t+1}), into a memory buffer called replay memory. When updating the model, random samples are drawn from this replay buffer to apply the Q-Learning update. This helps improve the stability of training by reducing correlations between consecutive transitions.

2.6.2 Target network

Another important aspect is the target network [5]. Instead of using the Q-network (also known as the policy network) directly in the TD target, an additional Q-network called the target network is utilized. The reason behind this is that in the standard approach, the TD target is a moving target. As the Q-network gets updated, both the values and the targets change, making the Learning process difficult and unstable. By introducing a target network, the goal is to stabilize the Learning process. This involves maintaining two separate networks, with one being updated every iteration and the other being updated less frequently.

The formula for the Q-Learning update with a target network is as follows:

$$Q(s_t, a_t) := Q(s_t, a_t) + \lambda \cdot \left[\overbrace{r_t + \gamma \cdot \max_a Q^{\text{target}}(s_{t+1}, a)}^{\text{TD target}} - Q(s_t, a_t) \right] \tag{12}$$

2.6.3 How does the training works

What exactly happens during training after storing an experience in reply memory as said, we then sample a random batch of experiences. For ease of explanation, we are going to explain the remaining process for a single sample, rather than for a batch. This idea is generalized to an entire batch.

From a single experience sample from replay memory, we then have $e_t = /s_t, a_t, r_{t+1}, s_{t+1})$, we pass s_t (or a preprocessed version of it), to the policy network as input. The input state data then forward propagates through the network, the network then gives as output an estimated Q-value, for each possible action from the given input state. At this point, the loss is then calculated with a loss function. We do this by comparing the q-value output for the network for the action a_t in the experience e_t that we sampled, in the corresponding optimal Q-value or target Q-value for the same action. This target Q-value is calculated using Equation (12). Then we have to compute: $\max_a Q^{\text{target}}(s_{t+1}, a)$, which involves the state and action that occur in the following time step. Previously in Table 1 we were able to find this max term by just peaking in the Q-table the highest Q-value for a given state. In deep Q-learning, in order to find this max term, s' is given to the policy network, which will output the Q-values for each state-action pair using s' as the state and each of the possible next actions as a'. Given this, we can obtain the max value over all possible actions taken from s', given this max term. Once we have this term we can then calculate eq. (12) for s_t. This enables us to calculate the loss between the Q-value given by the policy network for s_t and a_t and the target optimal Q-value.

2.6.4 Soft update

Finally, the soft update [4] is a mechanism used to gradually update the weights of the target network using the weights of the policy network. It enables a smoother transition between the two models during training. The formula for the soft update is:

$$\theta_{\text{target}} := \tau \cdot \theta_{\text{policy}} + (1 - \tau) \cdot \theta_{\text{target}} \tag{13}$$

In this formula, θ_{target} represents the weights of the target network, θ_{policy} represents the weights of the policy network, and τ is the update coefficient that controls how much the weights of the target network are influenced by the weights of the policy network. Typical values for τ range between 0.001 and 0.01.

2.6.5 DQN pseudocode

for $episode = 1$ **to** $N_{episodes}$ **do**
    Initialize state s;
    for $t = 1$ **to** T_{max} **do**
        Select action a based on ϵ-greedy;
        Execute action a, observe reward r and next state s';
        Store transition (s, a, r, s') in replay memory D;
        Sample random minibatch of transitions from D;
        Calculate target Q-values for each minibatch transition;
        Update Q-network weights using gradient descent;
        Soft update target network weights with Q-network weights using τ;
        Set state $s = s'$;
    end
end

Algorithm 1: DQN Pseudocode with Soft Update

3 Environment development

The development of the environment is introduced briefly and motivated to give all the elements and characteristics involved in this work, and explain a work intrinsically involved in the development of this thesis.

3.1 Developed from scratch: motivations

The choice of developing an environment for Settlers of Catan instead of choosing an already existing one, concern state representation. Our choice to model the Board as a graph makes our environment different from the others. And also, we wanted to be sure that the environment contained an important feature to replace the decision-making strategy of the agents.

More, in general, the aim was to add the possibility to change completely approach in an effective way in order to potentially help to understand the strengths and weaknesses of the different approaches and provide a basis for comparing their performance metrics, including Learning efficiency, strategic decision-making, and adaptability to dynamic game states.

And also, having complete control or even removing some stochastic components if desired, for example fixing the board or letting them choose between a few. And the same thing for dice generation results. This was made in order to understand the effect of the random elements in the agent Learning phase, or just to decrease its challenge in the Learning process that stochasticity introduces.

3.1.1 Settler of Catan as a benchmark

Initially, a motivation underlying the implementation of the environment was a shy yet ambitious endeavor to create an open-source, efficient, and customizable benchmark for algorithm research and studies, within a relatively simple and vast stochastic domain, a thing obtained by creating an environment specifically tailored for Settlers of Catan, which present the just mentioned features. The aim was to provide a benchmark for future research in the field of RL, and more in general in the field of Machine Learning. The developed environment can serve fundamentally as a testing territory for further investigations and advancements in many studies, for example, optimal state representation choices, decision-making and strategic planning, advancement in exploration-exploitation dilemma, and optimization in a complex and stochastic dynamic environment.

Overall, the development of the Settlers of Catan simulation environment aligns with the objectives of this research, allowing us to evaluate different *state* representations and assess the performance of intelligent agents. This environment provides a controlled and reproducible platform for experimentation, enabling us to gain valuable insights into the decision-making capabilities of the agents and contribute to the broader field of artificial intelligence and RL.

3.2 Software architecture, brief summary

The software architecture developed for the Settlers of Catan simulation was made in Python, to enable the easy use of techniques that involves deep learning. It incorporates various design patterns and follows a modular structure to enhance code organization and maintainability.

The architecture leverages Singleton, Command, and Strategy patterns to facilitate flexible and extensible development. The Singleton pattern is employed to ensure that certain classes have a single instance throughout the application, it has been used for the Board class, responsible for managing the game board and its state, and is implemented as a singleton to provide a global access point for interacting with the board. This pattern ensures that there is only one instance of the Board, promoting consistency and preventing multiple conflicting instances.

The Command pattern has been utilized to decouple the invoker of action from the object that performs the action. This pattern enables flexibility in executing commands and allows for easy extension of new commands. In the context of the Settlers of Catan simulation, the Command pattern is employed to handle player actions, such as building settlements, roads, or cities. Each action is encapsulated within a specific command class, which is executed by the game engine upon receiving a request from the player. This pattern allows for the easy addition of new actions management without modifying too much of the existing code.

The strategy pattern was used because, as mentioned before, the aim was the replaceability of different agents in a feasible manner, to encapsulate interchangeable agent algorithms. In the Settlers of Catan simulation, the Strategy pattern is utilized for implementing various player agents. Each agent is represented by a concrete class implementing a common strategy interface. This pattern allows for the easy addition of new player agents to the strategy interface. By leveraging Singleton, Command, and Strategy patterns, along with a modular design approach, the software architecture of the Settlers of Catan simulation facilitates code flexibility, extensibility, and maintainability. These design patterns and architectural choices contribute to the overall efficiency of the software debugging and correctness.

3.2.1 Choices adopted

To streamline the dynamics of the Settlers of Catan we choose to exclude the player-to-player trading and the utilization of a 1 vs 1 game variant. We adopted a 1 vs 1 game variant, where only two players/agents participate in the game. This variant helps in reducing the complexity and computational requirements associated with simulating multiple players' interactions and training efforts. By narrowing the scope to a 1v1 scenario, the focus can be placed on comparing and analyzing the performance of the intelligent agents in a more controlled and manageable setting. These modifications aim to strike a balance between maintaining the essential elements of Settlers of Catan and providing the evaluation and comparison of different approaches and performance of intelligent agents.

The 1 vs 1 version of the game is precisely nailed to play the 1 vs 1 ranked version of

Settlers of Catan presented by colonist.io, in this kind of configuration, the goal point is set to 15 and not 10. In general, this is done to promote the ability, trying to decrease the importance of the stochastic components avoiding "lucky wins". In other words, increasing the number of points allows the players to win because of abilities and strategies and not by stochastic factors.

4 State representation modeling

In the context of the Deep Q-Network (DQN), the choice of *state* representation is crucial for the successful Learning and performance of the algorithm. The *state* representation serves as the input to the neural network, which estimates the Q-values for different actions.

4.0.1 Small Recap on Graphs

A graph is a mathematical structure consisting of a set of nodes (also called vertices) and a set of edges. The nodes represent entities or objects, while the edges represent the connections or relationships between the nodes. Formally, a graph is defined as a pair $G = (V, E)$, where V is the set of nodes and E is the set of edges. The edges can be either directed or undirected, depending on whether they have a specific direction or not. In a directed graph, each edge is represented by an ordered pair of nodes, indicating a one-way relationship. In an undirected graph, the edges are unordered pairs of nodes, representing bidirectional relationships. Graphs are used in various fields, including computer science, mathematics, social networks, transportation systems, and more. They provide a powerful tool for modeling and analyzing complex relationships between different entities.

4.1 Catan Board as a graph

In the context of Settlers of Catan, graph convolutional networks (GCNs) can be leveraged in the graph-based *state* representation to capture and model the complex interactions and dependencies between game elements. A graph-based representation of the game *state* naturally lends itself to utilizing GCNs, which excel at processing structured data with interconnected nodes and edges.

In order to capture the features in this graph, two approaches utilizing GCN (in addition to classical neural networks) have been explored to extract the features of the graph. The first approach employs GCNs, while the second approach employs RGCNs.

GCNs and Relational Graph Convolutional Networks (RGCNs) are two types of convolutional neural networks specifically designed for processing graph-based data. Although they share some similarities, the main distinction between them lies in how they handle the relationships between nodes in a graph.

4.1.1 Graph Convolutional Networks

A Graph Convolutional Network (GCN) operates by aggregating information from neighboring nodes and updating node representations based on this local neighborhood information. GCNs are an extension of traditional neural networks that take advantage of graph structures, such as the graph structure of our study case.

GCN works on a mechanism called message passing, which is a powerful concept in graph algorithms. In message passing, nodes in a graph send and receive messages along their connections with neighbors. This happens in two steps: nodes send messages about

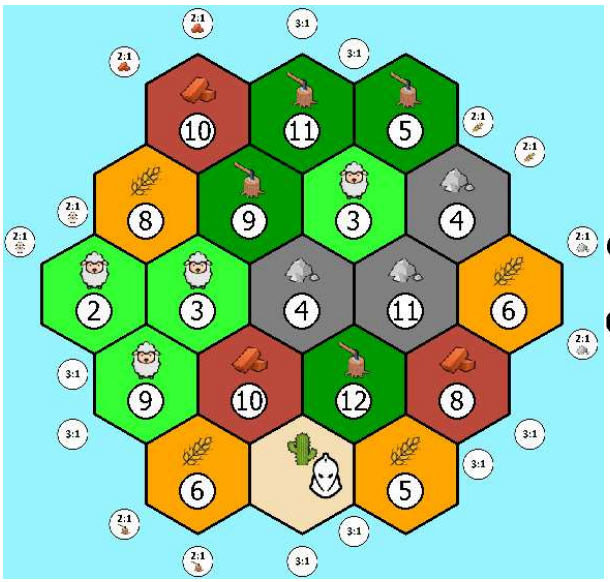

Figure 6: A possible board configuration

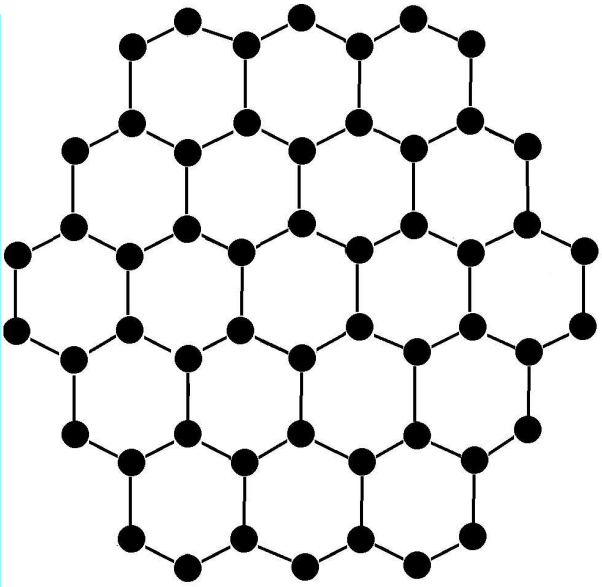

Figure 7: The graph structure of the board

Figure 8: Graph structure representation

themselves to their neighbors and then collect and use the messages received to update themselves and understand their environment. Labels can propagate through the graph by continually passing and collecting messages, which is the essence of the label propagation algorithm.

GCN is a powerful neural network architecture that operates on static graphs. It has been widely used in various graph-based tasks such as node classification, link prediction, and graph classification. The key idea behind GCN is to incorporate information from neighboring nodes into the representation of each node through graph convolutions.

Mathematically, the GCN update rule for a node v in the l-th layer can be defined as:

$$h_v^{(l+1)} = \sigma\left(\sum_{u \in \mathcal{N}(v)} \frac{1}{c_v} \cdot W^{(l)} \cdot h_u^{(l)}\right)$$

where $h_v^{(l)}$ represents the representation of node v at the l-th layer, $\mathcal{N}(v)$ denotes the set of neighboring nodes of v, $W^{(l)}$ is the weight matrix at layer l, and $\sigma(\cdot)$ is a non-linear activation function. The term c_v is a normalization factor that accounts for the variable number of neighbors for different nodes.

However, GCN considers only the direct connections between nodes and neglects more complex relationships that may exist in the graph.

4.1.2 Relational Convolutional Networks

Relational Graph Convolutional Network (RGCN) is an extension of GCN that addresses this limitation by capturing more intricate relationships in graphs. In many real-world

scenarios, graph data often contains multiple types of relationships between nodes, and those relationships are represented by the edges. For instance, in a social network, relationships can include friendship, familial ties, professional collaborations, and more. These relationships can be directional, meaning they have different implications depending on their direction.

RGCN takes into account these multiple and directional relationships by explicitly modeling them as separate "channels" within the network architecture. Each relationship type is treated as a distinct channel, and a specialized graph convolution operation is performed on each channel. Mathematically, the RGCN update rule can be defined as:

$$h_v^{(l+1)} = \sigma\left(\sum_{r=1}^{R}\sum_{u\in\mathcal{N}r(v)}\frac{1}{cvr}\cdot W_r^{(l)}\cdot h_u^{(l)}\right)$$

where R is the total number of relationship types, $\mathcal{N}r(v)$ represents the set of neighboring nodes of v connected by the relationship type r, $W_r^{(l)}$ is the weight matrix associated with relationship type r at layer l, and cvr is a normalization factor specific to each relationship type. This explicit modeling of multiple channels allows RGCN to capture more nuanced and complex relationships between nodes beyond direct connections. By considering multiple channels, RGCN can effectively capture the influence of different relationship types on node representations, resulting in a more comprehensive understanding of the graph structure. The key advantage of RGCN is its ability to model graphs with complex structures and rich relationship patterns. It enables the network to learn more expressive representations, which can lead to improved performance on various graph-based tasks. GCN focuses on the direct connections between nodes in a graph, RGCN takes into consideration multiple and directional relationships. By leveraging different channels and specialized graph convolutions, RGCN enables a more comprehensive and sophisticated representation of information within the graph. This key difference allows RGCN to effectively model graphs with complex structures, providing richer insights and better performance in various graph-based tasks.

RGCNs can potentially work better in this scenario by introducing different weights for the relations between nodes.

5 Proposed method

5.1 Board features representation

In our study case, Settlers of Catan, we represented the game board as a graph, where the vertexes of the tiles were treated as nodes, and the connections between them as edges.

The nodes correspond to what we have called places and the edges are the connection between them, in other words, possible streets.

Every player has an id, $i \in [1, n_{players}]$, which refers to their turn order. For example, if we had 2 players, the first player had id 1, the second player 2. But, in order to give capabilities to learn independently by the player id, and for example being able to train the same policy independently by the fact that has id x or id y, the features state of the node depends on the player in turn. In a sense that a node has label 1 if the player is the owner of the place, -1 if belongs to an opponent, otherwise 0. This way, for example, it allowed the possibility to do self-Learning, which is a way to train the same policy against itself.

Every resource type is associated with a number, specifically:

- Crop: 1
- Iron: 2
- Wood: 3
- Clay: 4
- Sheep: 5

The harbor features of a node have been associated with an id per type, specifically:

- 0 if a place has no harbor
- 1 if harbor type 3:1
- 2 if harbor type 2:1 crop
- 3 if harbor type 2:1 iron
- 4 if harbor type 2:1 wood
- 5 if harbor type 2:1 clay
- 6 if harbor type 2:1 sheep

The features of a node, in summary, are the following, **for every touched tile**:

- Resource-id associated with that tile
- Dice number associated with that tile
- A boolean telling if there is the robber or not in that tile

There is also a last feature for a place, which is the harbor, since every harbor has an id, the last feature is the harbors id if the place is a harbor, otherwise 0. So, because

of the fact that every node can touch at most 3 tiles, every node has 11 features, 1 for the ownership, 9 for the tiles, and 1 for the harbor. If a node touch less than 3 tiles, the corresponding tile has 0 in all the 3 features.

The edges instead have just a variable, intended to be 1 if the play is the owner of that street, 0 if no one is the owner, and -1 if another player owns it as for the places.

So the number of places is 54 and the number of edges is 72, so those are the summary of the board features.

Furthermore, there is also another small number of characteristics, that from here in after will be called "global features".

To give an idea of the state representation of a node, here we give a summary of the features of a node.

Indicating a resource of a tile with r_{tile}, the number of a tile with n_{tile} and b_{tile} the boolean that tells if the robber is in that particular tile and h_{place} its harbor-id as explained above.

Given a node indicated with $node_k$, which represents the place k, that touches $tile_1$, $tile_2$ and $tile_3$, its features are the following:

$$node_k = \left\{ \underbrace{r_{\text{tile}_1}, n_{\text{tile}_1}, b_{\text{tile}_1}}_{\text{First tile}}, \underbrace{r_{\text{tile}_2}, n_{\text{tile}_2}, b_{\text{tile}_2}}_{\text{Second tile}}, \underbrace{r_{\text{tile}_3}, n_{\text{tile}_3}, b_{\text{tile}_3}}_{\text{Third tile}}, h_{\text{place}_n} \right\}$$

Instead given an edge e_i, and indicating with own its ownership status as seen previously, its feature is just:

$$edge_i = \{own\}$$

Which is -1 if belongs to an opponent, 0 if owned by nobody, and 1 if owned by this player.

Since in the board, there are 54 nodes and 72 edges, we have 54 vectors for the nodes and 72 vectors for the edges for a single Board state representation.

5.1.1 Global player features

The global player features are meant to give information regarding the elements that are not dependent on the Catan Graph.

Those features are the following, its victory points $vp \in [0, 15]$, its owned resources $n_{type} \in [0, 19]$, because of the bank limit, as explained in Section 1.4, one per type, for the sake of clarity, the features regarding the resource of the player are those:

- Number of crop-type resources owned
- Number of wood-type resources owned
- Number of iron-type resources owned
- Number of clay-type resources owned

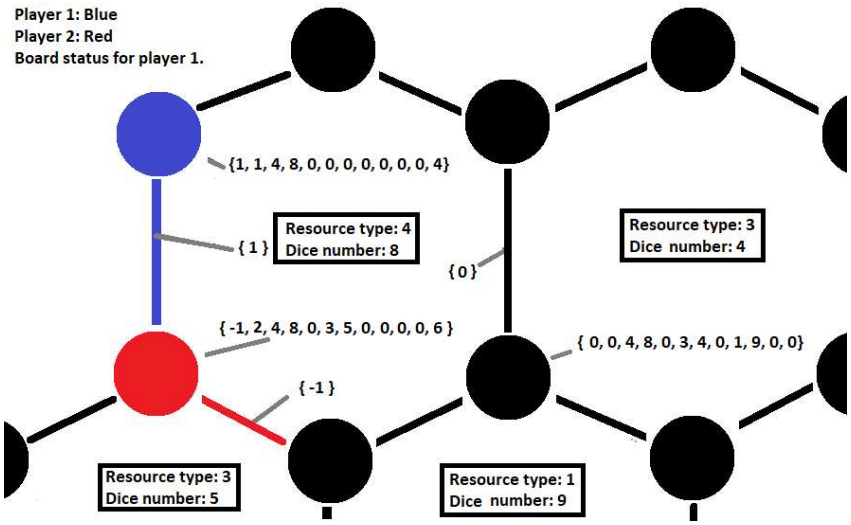

Figure 9: Example of Board features on nodes and edges

- Number of sheep-type resources owned

And also, a number that specifies how many knights that player has used in the game (this is important since the player who used more knights has 2 more victory points as explained in Section 1.4), the number of total cards bought by that player, and the total number of resources outside of the bank, which we believed that could be useful to the use of the Monopoly Card.

So the global features of each player are represented by a vector of 9 elements.

To summarize also the global features, given a player $player_{id}$, indicating its victory points with vp, the number of owned resource types with its id-type in the following way: n_{type}, the number of knights used with k_{used}, the number of card bought with c, and the number of total resources out with tot.

The features of the id-player are:

$player_{id} = \{vp,\ n_{crop},\ n_{wood},\ n_{iron},\ n_{clay},\ n_{sheep},\ k_{used},\ c,\ tot\}$.

5.2 Net architectures

5.2.1 Graph neural network architecture

The architecture of the utilized DQN has been purposefully designed to use the graph convolutional neural network, enabling the DQN to effectively leverage the processed features at a higher abstraction level. This integration facilitates a feature extraction mechanism, allowing the DQN to capture and utilize more meaningful and informative representations of the input data. Consequently, the DQN can dynamically adapt its weights and update them based on this enhanced understanding. The architecture is shown in the figure Figure 7.

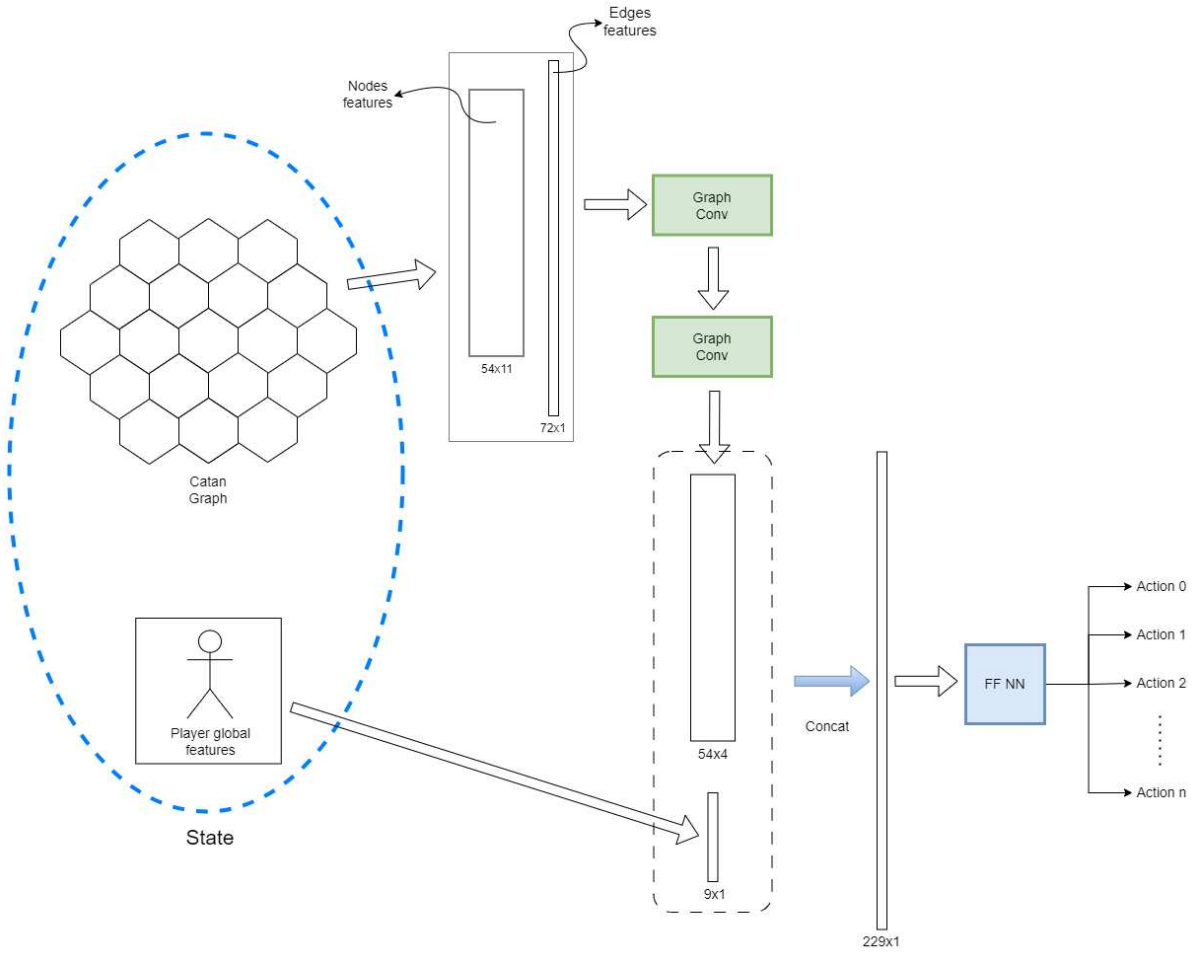

Figure 10: DQN architecture with graph convolutional network and relational graph convolutional network

5.2.2 Feed forward net

In the context of the more conventional approach that employs a feed-forward network, we employed a comprehensive strategy. The representation of each individual node was extracted and simply concatenated with the representation of the corresponding edges. This seamless fusion of node and edge representations culminated in a comprehensive input, which was then seamlessly fed into the feed-forward network for weight updates. By embracing this methodology, the network was empowered to effectively leverage the collective information encapsulated within both the nodes and edges. As shown in Figure 8.

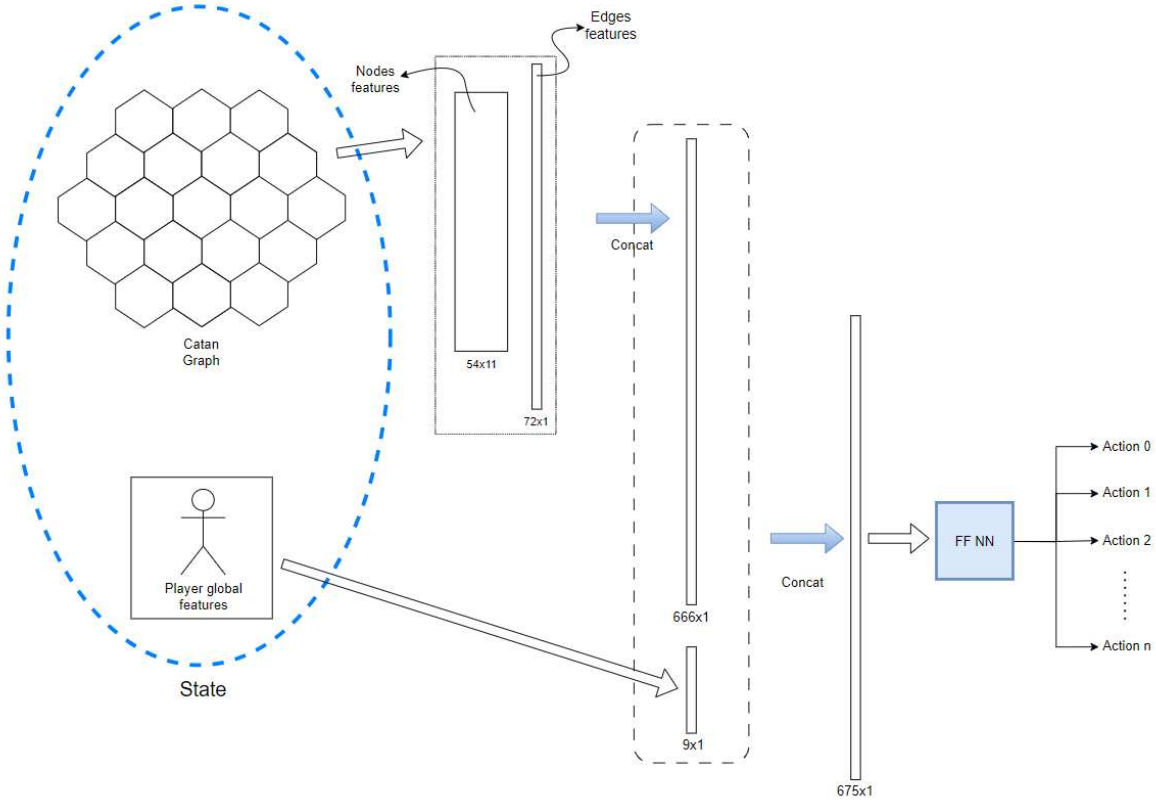

Figure 11: DQN architecture with feed-forward neural network

5.3 Action selection approaches

As seen in the previous chapters, where the game rules are explained, it is evident that there are numerous rules and different action types that can be done at every turn. We define two different concepts, in order to give clarity and avoid confusion. The word build means: buy and collocate. An **action** is one of the following elements:

- Build a colony.
- Build a street.
- Build a city.
- Buy a development card.
- Trade with the bank.
- Use a "Knight" card.
- Use "Monopoly" card.
- Use "Year of plenty" card.
- Use "Road building" card.
- Pass turn.

Every possible way to perform an action is called **execution**.

Example: action chosen "building a colony". Available places: place 3 and place 5. In this example, "building a colony" is the action, and the two different ways how to execute this action (i.e., if build it in place 3 or build it in place 5) are the executions.

As seen in Section 1.4, each action type has many possible executions depending on the state of the board.

There is an execution set for each action. These sets are the following:

- 54 possibles executions for the action build a colony.
- 72 possibles executions for the action build a street.
- 54 possibles executions for the action build a city.
- 1 possible execution for the action buy a development card.
- 25 possibles execution for the action trade with the bank.
- 19 possibles execution for the action use a "Knight" card.
- 5 possible execution for the action use "Monopoly" card.
- 2 times 5 executions for the execution for the action "Year of plenty" card.
- 2 times 72 executions for the execution for the action use "Road building" card.
- possible execution for the action pass turn.

In order to build an effective RL agent, those executions must be explored in an efficient manner.

As said before, in a DQN the number of outputs corresponds to every possible behavior, which in this case would be every possible execution.

If we create a single gigantic DQN with outputs for all the possible execution of every possible behavior, we would have: 54 execution for the "build colony" action (one for each possible colony placement) + 54 execution for the "build city" action (one for each possible colony upgrade) + 72 execution for the "build street" action + 25 executions for the "trade with bank" action + 19 executions for each possible placement of the robber, and even more options for each action related to different types of cards ("monopoly,"year of plenty", and "road building""), with a result of over 200 possible behaviors. This approach would have several problems:

- Management and implementation: specifically selecting the best execution among the available options would be complicated since not all the actions are available every turn (for example, in some turns a player may not have enough resources to build a colony), and most of all, not every possible choice for that action is available every turn (for example, I can build a colony, but not in every place, just in the free ones and where I build connected streets).
- Training problems: since training a network with such a large number of output nodes would require more nodes in the internal layers. A network of this size is certainly computationally expensive to train.
- Exploration problems: If we had a huge gigantic DQN with all the possible execution options while exploring, so while picking a random action, the consequence would be that the action with more execution options would have more probability to be chosen. This is an undesired effect. We would like to choose randomly an action in a way that each action has the same probability to be chosen, and after the action is chosen explore its possible executions with a different exploration.

For this reason, we chose to proceed with and study a hierarchical approach, which will be discussed in detail in the next sections, as it offers several advantages in particular as will be shown a really huge one.

6 Hierarchical design

In this work, we have chosen to design the action selection process in a hierarchical manner. That is, there is a main entity, hereinafter referred to as the *orchestrator*, whose task is to choose which action should be taken. However, its role is not to execute the action itself; that responsibility will be delegated to another entity responsible for its execution, as illustrated in Figure 9. To better explain this concept and our desired behavior, we can draw a natural parallel: an artisan, when faced with a problem, such as something that needs fixing, goes through two thought processes. The first one selects the tool he deems adapted, and then the second one is about how to use that tool effectively.

In the same way, we want to separate hierarchically the phase of choosing an action, and the phase of choosing an execution.

In our case, this approach offers several advantages. First and foremost, it allows us to break down and separate the problem into different specialized policies and also adds the potential interesting ability to enable us to test how the specialization of each component impacts the performance of our agent, however, this aspect is left to future works. Another strong component added is to potentially exploit different training and to replace different specialized executioners if desired. As it will be shown, the final and best improvement, which would never be possible with a non-hierarchical approach, is the possibility to assign different subgoals to each executioner specialized, as shown in the following sections.

6.0.1 Reward choice

Of course, when dealing with an RL problem, the choice of reward is crucial and requires careful consideration. In our case, as the objective of this game is to reach a certain score (15 in the 1v1 setting), we had two options for the reward:

1. **Sparse reward at the end of the episode**: This means assigning a positive reward only if the episode satisfies the goal, otherwise assigning a reward of 0.
2. **Assigning a reward to each move** that corresponds to the current score of the player.

Considering that the sparse reward would significantly increase training times, we decided to proceed with assigning a reward to each move, which corresponds to the current score of the player.

6.1 Agents

A total of nine different types of agents have been created, each varying in terms of the algorithms used for decision-making and action execution, and they will be explained in more detail in the next sections.

The first agent was an agent that choose both action and execution in a random manner.

The next three agent types serve as additional baselines and utilize three distinct DQN decision-making methods. The first agent type employs a feed-forward neural network, the second agent type utilizes a graph convolutional neural network, and the third agent

type incorporates a relational graph convolutional network to extract state features. These agents choose the action but do not execute it, in those first agents, it is executed with a random choice between its possibilities.

Collectively, these three agents demonstrate how the hierarchical approach outperforms the random player, even with a random execution of moves. These agents are respectively referred to as DQN FF/RAN, DQN GCN/RAN, and DQN RGCN/RAN.

Subsequently, three agents have been implemented with a DQN entity responsible for action selection and another DQN entity dedicated to executing a subset of actions. The results indicate that these types of agents perform significantly better than the previous ones. These agents are respectively named DQN FF/FF, DQN GCN/GCN, and DQN RGCN/RGCN.

Finally, the last three agents represent versions with a distinctive subgoal during one of the action executions. These agents have been provided with more specific rewards to showcase the potential of the hierarchical approach. These agents are respectively denoted as DQN FF/FF*, DQN GCN/GCN*, and DQN RGCN/RGCN*.

6.1.1 DQN Orchestrator

The first DQN algorithm implemented was the orchestrator DQN, as shown in Figure 10. It chooses the action, but it does not choose the execution. An agent composed exclusively by this DQN orchestrator has decision-making on the action, but the execution of it randomly. An agent with those characteristics has been created. Its variants are regarding the type of DQN orchestrator involved. That as seen in Section 4 can be of 3 types, one for each architecture, they are respectively referred to: DQN FF/RAN, DQN GCN/RAN, DQN RGCN/RAN. However, since the execution of each execution is still random, major improvements were not expected.

6.1.2 DQN specialized

Furthermore, additional DQNs have been implemented. They have been called DQN specialized, and their purpose is to make the agents able to execute the actions chosen by the orchestrator DQN.

These DQNs are trained in the same way, taking the state representation as input and returning one of the possible options. As said before every action has many possible executions, the outputs of this specialized DQN_{action} are every possible execution of that *action*, as explained in Section 5.3. The reward in this DQN was still the victory points, as for the DQN orchestrator.

For example, in the case of the build colony DQN, the number of possible outputs corresponds to the number of available spots on the board. When the orchestrator DQN determines that the best action is to place a colony, the "DQN build colony" selects the execution, i.e., the specific position to collocate the colony among the allowed options.

This mechanism of delegating the execution to other DQNs specialized in every single action is shown in Figure 12.

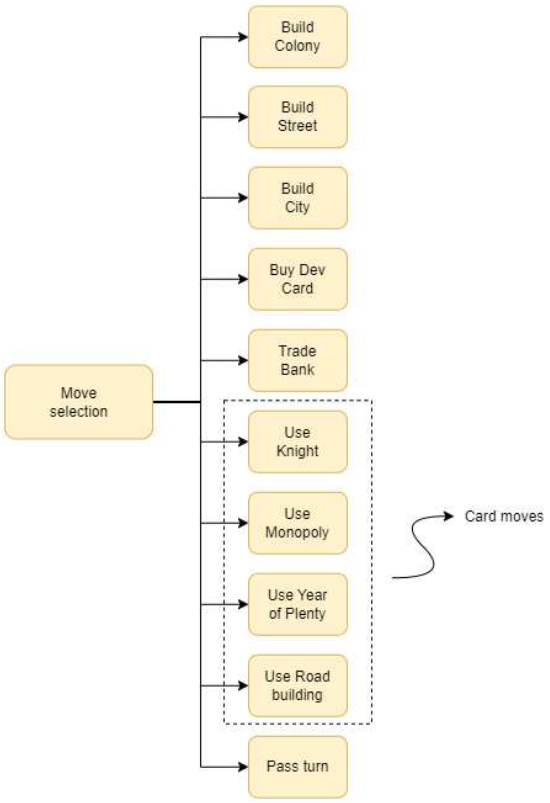

Figure 12: Hierarchical move chosen random and random execution

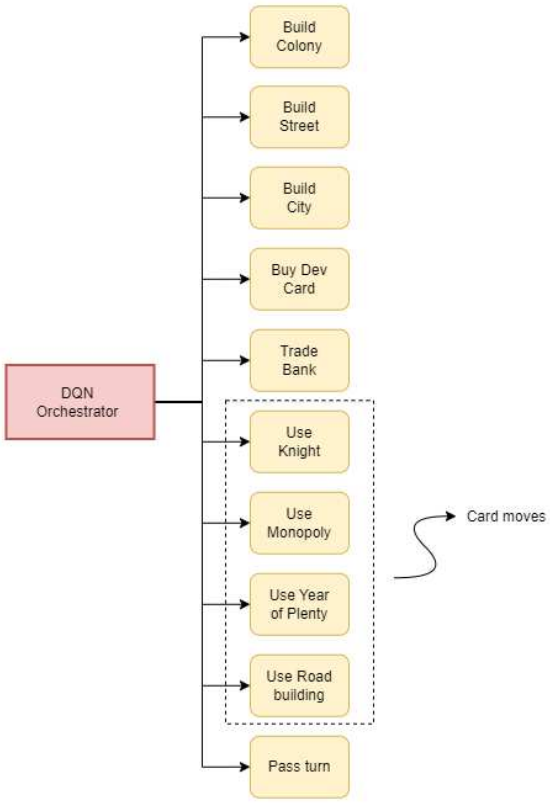

Figure 13: Hierarchical move chosen by orchestrator DQN and random execution

Figure 14: Hierarchical move specialization

To mitigate the increase in training time associated with the involvement of multiple DQNs, we opted for a subset of DQN specialization, focusing on the actions that we consider to be the most important. The purpose of creating this specialization was to demonstrate the concept that performance would improve with a specialized DQN, which was indeed observed in the results.

6.1.3 DQN specialized with subgoals variation

We also wanted to emphasize an important feature that provides additional motivation and illustrates another potential advantage of a hierarchical architecture. This feature entails the capability to assign distinct tasks and goals to specialized DQNs. In Figure 13, we can observe that the DQN "build street" have been assigned a unique subgoal, with its reward based on the length of the longest street, which as explained in Section 1.4 has a central role and gives +2 victory points.

This demonstrates the flexibility and adaptability of the hierarchical approach in accommodating different objectives for specialized agents.

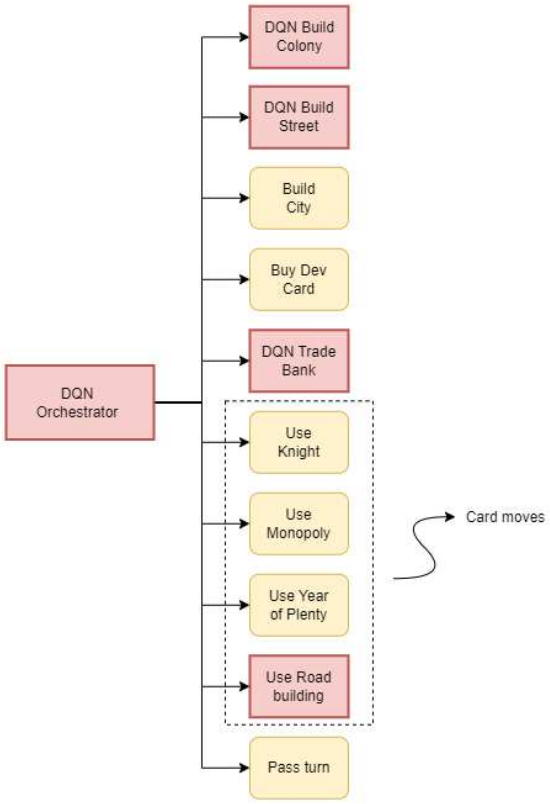

Figure 15: Hierarchical action chosen by orchestrator DQN and random execution

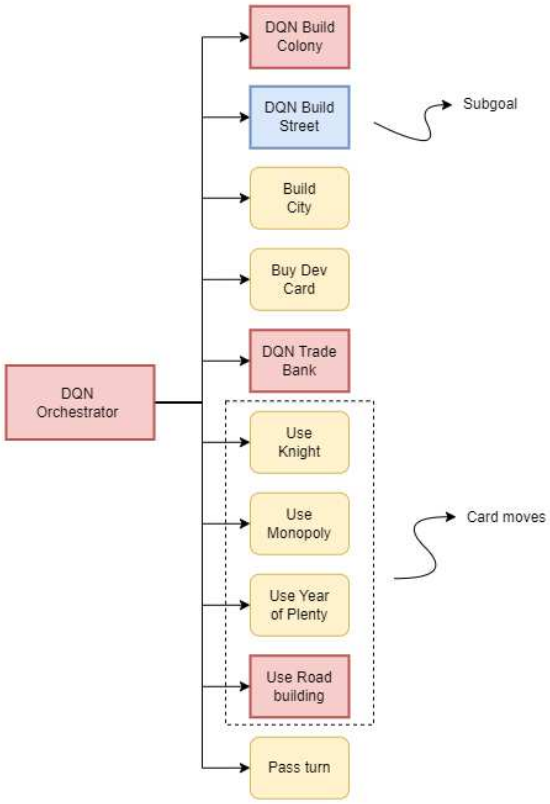

Figure 16: Hierarchical action chosen by orchestrator DQN and executions chosen by DQN specialized

Figure 17: Hierarchical action and execution specialization

7 Training

In this section, a brief summary of the parameters utilized during the training is provided. The training was executed on a Desktop computer with the following characteristics:

- CPU: 11th Gen Intel(R) Core(TM) i7-11700F @ 2.50GHz, 2.50 GHz
- GPU: Nvidia GeForce 3060 RTX
- RAM: 16 GB

Every training iteration is composed of 1000 episodes for each agent. An episode corresponds to a game played against a random policy. Each episode has approximately a duration of 2.5 seconds.

To build a statistical evaluation, we evaluate the run of 5 agents per type in order to plot the interquartile range. So, approximately for every agent type, there were 1000×5 episodes, for a total of approximately 12500 seconds, which is approximately 3.47 hours.

And this was done for every agent type, resulting in a total training duration of 3.47×9 hours, which is approximately **31.23 hours**.

These parameters and the duration of the training reflect the efforts undertaken to ensure a robust and comprehensive evaluation of the performance of each agent type.

7.1 Training settings

To perform the epsilon decay, we introduced a variable called decay β: which was set to 0.996 in the DQN orchestrator, and at the end of every episode, the epsilon was multiplied by β.

In the agents with the DQN specialization, the actions are made fewer times compared to the DQN orchestrator choice. Because of that, they explore less, to overcome this fact β was set at 0.997 for them.

To perform the training, every agent had the following parameters:

- Replay memory size = 1000
- Batch size = 64
- $\gamma = 0.99$
- Initial $\epsilon = 1$
- $\tau = 0.005$
- $\alpha = 1 \cdot 10^{-3}$

To train the networks a SmoothL1Loss has been used. The formula for SmoothL1Loss is, where y is the desired value and $\hat{y}$ are the predicted value:

$$\text{SmoothL1Loss}(y, \hat{y}) = \begin{cases} 0.5 \cdot (y - \hat{y})^2, & \text{if } |y - \hat{y}| < 1 \\ |y - \hat{y}| - 0.5, & \text{otherwise} \end{cases}$$

The AdamW (Adam with Weight Decay) optimizer has been used. Those parameters are the ones considered standard by the literature.[3].

To reduce the stochasticity and favor the training, the initial configuration of the Boards of the environment has been limited to 4 seeds.

7.2 Training results

During the RL training phase, the agent engages in episodes, where each episode corresponds to a game. To assess the performance of the agent, we focused on tracking a specific aspect: the average point scored by the agent at turn 120. This particular turn serves as a practical and informative measure to understand how the agent is progressing. In each episode, we recorded the points accumulated by the agent at turn 120. By analyzing these points across the episodes, we obtained insights into the performance of the agent over time. This approach helped us monitor the progress of the agent and evaluate its Learning capabilities.

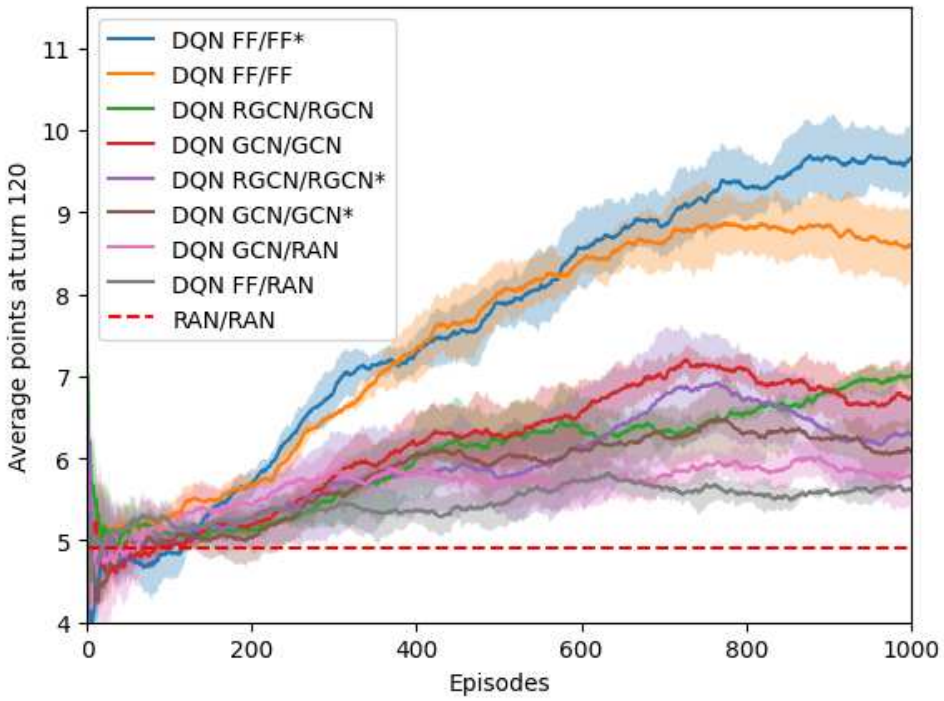

Figure 18: Training trend after 1000 episodes of training, with 4 seeds in the board configuration. "DQN type1/type2" indicates that the orchestrator is of type1 and the executions are of type2. The asterisks (*) denote the subgoal variation. The shaded area is the interquartile range. Every point in the plot is the mean of 100 subsequent episodes.

The first observation is that the agent that performs better is the feed-forward agent with a subgoal variant. At the end of training reaches a mean of 9.5 points, which is a promising result compared to the random policy and to the other agents.

In the plot, one notable observation is that the mean reward initially exhibits a pattern resembling a random policy. This behavior is expected since, during the early stages of

training, the agents lack any prior knowledge about the states they encounter. Additionally, in the beginning, phase, the epsilon value remains relatively high, indicating that the exploration part of the training process plays a dominant role.

However, as training progresses, a significant change becomes apparent. The epsilon value undergoes decay, following a decay factor, if n is the number of the episode, the ϵ can be calculated in the following way: $\epsilon \cdot \beta^n$.

For example, after approximately 200 episodes, if we consider a β of 0.996 (like in the DQN Orchestrator), the calculation $1 \times 0.996^{200} \approx 0.44$ shows how the epsilon value decreases over time.

This change in the exploration-exploitation trade-off leads to a remarkable shift in the behavior of the agent. The immediate increase in performance becomes evident, as the agents gradually acquire knowledge and learn to exploit the optimal actions within the environment. Over subsequent episodes, the performance continues to improve, indicating the effectiveness of the training process and the ability of the agent to make informed decisions.

The observed transition from a random policy to a more refined and effective strategy highlights the Learning capabilities of the agents. It showcases their capacity to adapt and refine their actions based on the feedback and exploration-exploitation balance achieved through the training process.

As expected, even the orchestrator version, which chooses the actions but it executes them randomly, significantly improves performance compared to random rewards. Furthermore, there is a notable difference in performance between the network architectures.

As the results show, the hierarchical DQN with the feedforward neural network performs better after 1000 episodes of training. The feed-forward version outperforms the models with the graph convolutional neural network and relational graph convolutional network.

Initially, we believed that in this dynamic case, the graph neural network would be more effective. However, the results indicate the opposite. The reason for this is that we have a fixed number of parameters in this scenario, which is typically ideal for a feed-forward network. The true potential of graph convolutional networks is likely observed in graphs of larger dimensions.

However, we do not rule out the possibility that with the right amount of training and different architecture parameters, graph convolutional neural networks could be used effectively.

7.3 Self-Learning

As explained in the upcoming section, the primary goal was to conduct an empirical evaluation of the agents through a tournament. Out of curiosity, we introduced an additional agent trained with self-Learning for 3000 episodes, aiming to assess whether it would achieve improved performance.

The agent chosen for self-Learning was our best one: DQN FF/FF*, and it will be referred to as DQN SELF-L FF/FF*, but as mentioned previously was not one of the goals of

this thesis and it is just inserted for a small empirical test.

Similar to the previous plot, we analyzed the average points obtained by this agent at turn 120. However, it is important to note that the agent was not playing against a random policy but rather against itself, hence the values are not directly comparable.

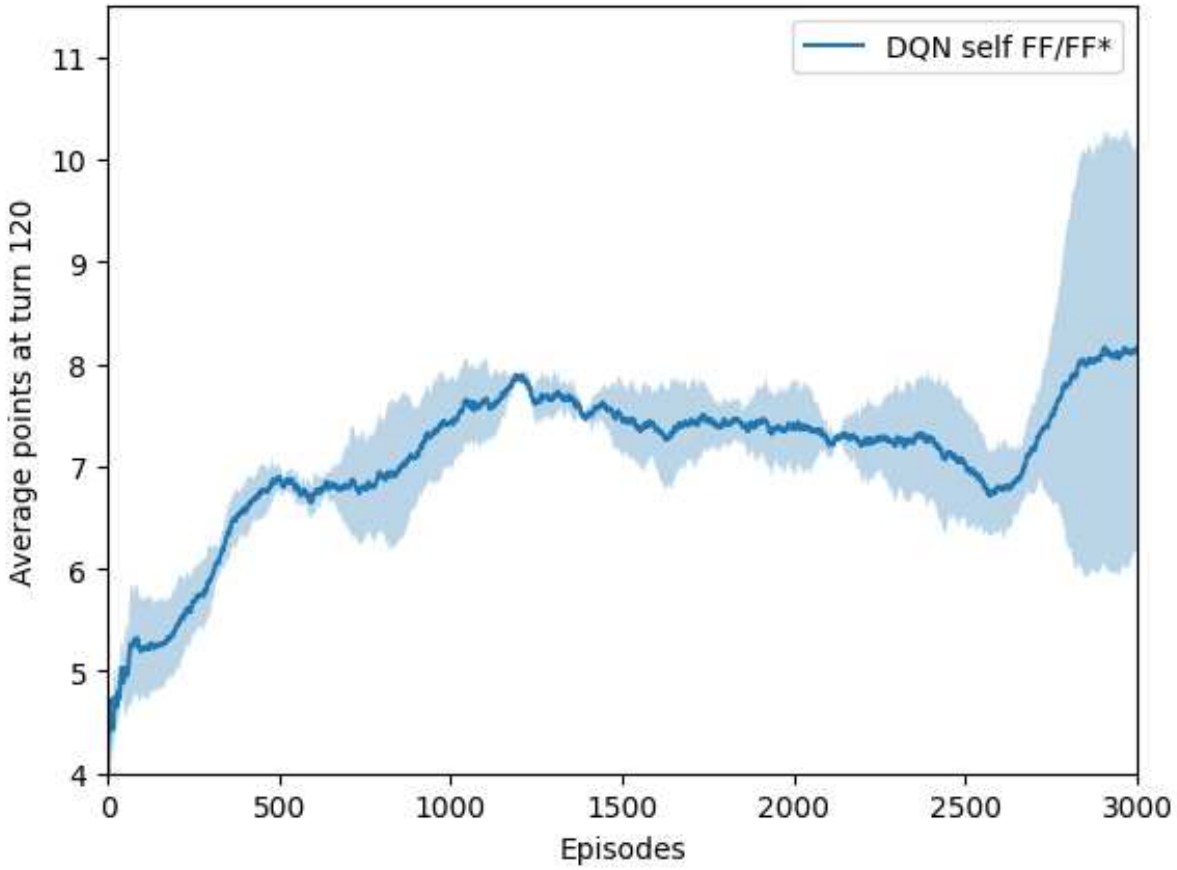

Figure 19: Plot of self-Learning results.

It is noticeable that the final part of the plot exhibits significant variance. This is due to the fact that the agent is not playing against a random opponent; instead, it is competing against itself. Additionally, since both players are making optimal actions and executions, not every game results in a closely contested ending. Conversely, if player 2 is also able to effectively exploit its acquired knowledge, the game may end prematurely.

8 Empirical competition

To empirically evaluate the performance of various types of agents, we organized a tournament. For each agent type, we had three trained network weights. We selected the trained weights that demonstrated superior performance against the random agent for participation.

Consequently, there is one agent per type in the tournament.

The types of agents involved in the tournament are as follows:

- SELF-L FF/FF*
- FF/FF*
- FF/FF
- RGCN/RGCN
- GCN/GCN
- RGCN/RGCN*
- GCN/GCN*
- GCN/RAN
- FF/RAN
- RAN/RAN

The agent characteristics are as explained earlier, where "type1/type2" indicates that the orchestrator is of type1 and the executions are of type2. The asterisks (*) denote the subgoal variation.

The tournament is structured in this way: every agent plays 10 games against every other agent, 5 in first position and 5 in second position. In this way, every player will

	SELF-L FF/FF	FF/FF*	FF/FF	FF/RAN	GCN/GCN*	GCN/GCN	GCN/RAN	RGCN/RGCN*	RGCN/RGCN	RAN/RAN
SELF-L FF/FF*	X	5-0	3-2	2-3	2-3	4-1	4-1	4-1	4-1	4-1
FF/FF*	5-0	X	5-0	5-0	5-0	3-2	5-0	5-0	4-1	4-1
FF/FF	3-2	5-0	X	3-2	1-4	5-0	5-0	4-1	4-1	5-0
FF/RAN	3-2	2-3	2-3	X	5-0	4-1	4-1	3-2	3-2	5-0
GCN/GCN*	2-3	2-3	4-1	1-4	X	4-1	2-3	3-2	4-1	5-0
GCN/GCN	3-2	0-5	4-1	2-3	5-0	X	5-0	1-4	3-2	2-3
GCN/RAN	1-4	1-4	4-1	3-2	2-3	3-2	X	3-2	4-1	4-1
RGCN/RGCN*	1-4	1-4	5-0	2-3	2-3	5-0	4-1	X	5-0	4-1
RGCN/RGCN	4-1	1-4	3-2	4-1	4-1	5-0	3-2	5-0	X	5-0
RAN/RAN	2-3	1-4	0-5	2-3	2-3	1-4	0-5	0-5	0-5	X

Figure 20: Results: every cell represents the result of 5 games. The first result is the number of wins of the agent on the row, the second result is the agent on the column. In this way, it is observable the results in the first position and in the second position. There is an x in the diagonal because the players did not play against themselves.

Victories resume:

Agent type	Number of victories			Winrates		
	First position	Second position	Total	First position	Second position	Total
SELF-L FF/FF*	32/45	21/45	53/90	71.1%	46.6%	58.8%
FF/FF*	41/45	27/45	68/90	91.1%	60.0%	75.5%
FF/FF	35/45	15/45	50/90	77.7%	33.3%	55.5%
FF/RAN	31/45	21/45	52/90	68.8%	46.6%	57.7%
GCN/GCN*	27/45	17/45	44/90	60.0%	37.7%	48.8%
GCN/GCN	25/45	11/45	36/90	55.5%	24.4%	40.0%
GCN/RAN	25/45	13/45	38/90	55.5%	28.8%	42.0%
RGCN/RGCN*	32/45	14/45	46/90	71.1%	31.1%	51.0%
RGCN/RGCN	31/45	17/45	48/90	68.8%	37.7%	53.3%
RAN/RAN	8/45	7/45	15/90	17.7%	15.5%	16.6%

Figure 21: Victories resume.

The results are synthesized in the table above. As expected the win rate in the second position drops consistently for two main reasons:

- The agents during the training were in the first position, and probably this fact make them learn more precisely about states and configurations in this position.
- Most probable reason: in the Settlers of Catan 1v1 version the first player has a significant advantage based on the initial phase, Section 1.4.3, it can choose the best colony on the board and this is a **significant advantage**.

Surprisingly the FF/RAN agent empirical performances were superior to all the GCN-based and RGCN-based algorithms.

8.1 Final ranking

The results of the tournament are summarized in the table below, it is just a simple aggregation of the victories in order to make a final classification. As the table shows, the best agent was without a doubt the FF/FF*. The full random agent was able to win a few games due to unavoidable random factors.

Agent type	Total winrates
FF/FF*	75.5%
SELF-L FF/FF*	58.8%
FF/RAN	57.7%
FF/FF	55.5%
RGCN/RGCN	53.3%
RGCN/RGCN*	51.0%
GCN/GCN*	48.8%
GCN/RAN	42.0%
GCN/GCN	40.0%
RAN/RAN	16.6%

Figure 22: Final ranking.

It is also interesting that the FF/RAN agent did not perform in a better way with respect to the GCN and RGCN in the first position, as expected. But it gained a lot of victories, in the second position, as shown in Figure 21.

9 Conclusions and future works

Our objective was not only to develop agents capable of leveraging the hierarchical DQN algorithm but also to gain a deeper understanding of its potential and limitations in the context of the board game Settlers of Catan. Throughout our research, we explored various network configurations and training strategies to consistently outperform the random hierarchical player.

One of the key findings of our study was the remarkable performance of the DQN feed-forward/feed-forward agent with the subgoal variant. This agent demonstrated a clear advantage in terms of decision-making and execution, highlighting the powerful exploitable feature of the hierarchical approach. By assigning different a subgoal to a specialized DQN, we were able to achieve superior performance and strategic decision-making.

Interestingly, the performance of the graph neural network variants, the GCN and RGCN agents, did not meet our initial expectations. This observation suggests that these network architectures require further investigation and fine-tuning of hyperparameters and training techniques. While they did not perform as well as the feed-forward/random agent, we believe that with the right adjustments and optimization, they hold the potential to yield competitive results.

Looking ahead, our research roadmap includes several exciting directions. Firstly, we plan to conduct extensive testing of our agents against human players. By engaging in real-world gameplay scenarios, we can gain valuable insights into the effectiveness and adaptability of our agents. Additionally, we aim to explore self-Learning approaches, enabling our agents to improve and adapt their strategies based on games against themselves.

Furthermore, we recognize the importance of investigating alternative RL algorithms beyond DQN. This exploration will allow us to uncover new insights and potentially discover algorithms that are better suited for capturing the complex dynamics and strategic decision-making required in Settlers of Catan.

To broaden the scope of our research, we intend to leverage online platforms such as "Colonist.io" to test and train our agents against a diverse range of human opponents. This provides a valuable opportunity to evaluate the performance of our agents in a more realistic and challenging setting, ultimately enhancing their capabilities and robustness.

Overall, our study has shed light on the potential of hierarchical DQN and its applications in complex board games like Settlers of Catan. By continuously refining and expanding our research, we aim to contribute to the advancement of AI in board game playing, and potentially extend our findings to other domains with similar challenges and characteristics.

References

[1] A. Bauer. Artificial intelligence with graph neural networks applied to a risk-like board game. *IEEE Transactions on Games*, 2023.

[2] J. Carr. Using graph convolutional networks and td (λ) to play the game of risk. *arXiv preprint arXiv:2009.06355*, 2020.

[3] T. Eimer, M. Lindauer, and R. Raileanu. Hyperparameters in reinforcement learning and how to tune them. *arXiv preprint arXiv:2306.01324*, 2023.

[4] R. Fox, A. Pakman, and N. Tishby. Taming the noise in reinforcement learning via soft updates. *arXiv preprint arXiv:1512.08562*, 2015.

[5] J. F. Hernandez-Garcia and R. S. Sutton. Understanding multi-step deep reinforcement learning: A systematic study of the dqn target. *arXiv preprint arXiv:1901.07510*, 2019.

[6] M. Hillebrand, M. Lakhani, and R. Dumitrescu. A design methodology for deep reinforcement learning in autonomous systems. *Procedia Manufacturing*, 52:266–271, 2020.

[7] B. Jang, M. Kim, G. Harerimana, and J. W. Kim. Q-learning algorithms: A comprehensive classification and applications. *IEEE access*, 7:133653–133667, 2019.

[8] P. Kormushev, S. Calinon, and D. G. Caldwell. Reinforcement learning in robotics: Applications and real-world challenges. *Robotics*, 2(3):122–148, 2013.

[9] T. D. Kulkarni, K. Narasimhan, A. Saeedi, and J. Tenenbaum. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. *Advances in neural information processing systems*, 29, 2016.

[10] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[11] J. Perolat, B. De Vylder, D. Hennes, E. Tarassov, F. Strub, V. de Boer, P. Muller, J. T. Connor, N. Burch, T. Anthony, et al. Mastering the game of stratego with model-free multiagent reinforcement learning. *Science*, 378(6623):990–996, 2022.

[12] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[13] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

[14] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[15] L. Veracini. Settlers of catan, 2013.

[16] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

[17] S. Zhang and R. S. Sutton. A deeper look at experience replay. *arXiv preprint arXiv:1712.01275*, 2017.