FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Deep Reinforcement Learning in Strategic Multi-Agent Games: the case of No-Press Diplomacy

Diogo Henrique Marques Cruz

DISSERTATION

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Prof. Henrique Lopes Cardoso

July 3, 2019

Deep Reinforcement Learning in Strategic Multi-Agent Games: the case of No-Press Diplomacy

Diogo Henrique Marques Cruz

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Prof. Jorge G. Barbosa
External Examiner: Prof. Brígida Mónica Faria
Supervisor: Prof. Henrique Lopes Cardoso

July 3, 2019

Abstract

Artificial Intelligence breakthroughs have a well-known and acclaimed connection with Strategic Games. Backgammon, Chess, and Go have been used to show the results of renowned *reinforcement learning* algorithms.

This work explores *strategic multi-agent games* as an environment for *deep reinforcement learning* research. *Strategic games* require large state space analysis and long term planning in order to develop a winning strategy. These skills are attractive to be researched using *deep reinforcement learning*. The multi-agent factor in a *strategic game* introduces more complexity and makes the development of a strategy harder.

A well-studied game of this category is Diplomacy. This turn-based game has compelling features to be explored in a *multi-agent system* approach. The combat in this game is a free-for-all where every player can attack or defend any other player. As the players' actions are simultaneous, the agent will have to create a long term strategy and also trust relationships with the opponents while strategically positioning its across the map to achieve success. BANDANA is a public testbed based on Diplomacy that allows the development of agents.

This work created a model on how to approach *deep reinforcement learning* in a *strategic multi-agent game* scenario. To exemplify the usage of the model, DeepDip, a no-press agent for Diplomacy, was implemented using the BANDANA testbed.

In order to create DeepDip, *gym-diplomacy*, an open-source OpenAI Gym environment was built and made publicly available. This environment provides a testbed for research of *reinforcement learning* techniques in the Diplomacy game. In an OpenAI Gym environment, the agent decides when to advance to the next state, but in the case of BANDANA and *multi-agent systems* is the environment that decides when to alternate state. The *gym-diplomacy* changes the usual OpenAI Gym environment architecture to let it decide when to move to the next state. It has compatibility with the example agents provided by OpenAI. This environment includes Diplomacy's standard map and additional variants for two and three players.

Using the Proximal Policy Optimization algorithm, DeepDip was able to win the two players variant versus a DumbBot and was starting to improve its results on the three-player variant. The agent was able understand the rules of the game and develop a strategy to win the game. This proves that environment is well-designed to develop a *deep reinforcement learning* agent in a *strategic multi-agent game* scenario. Both the standard and the three-player variant experiences needed more training time to make conclusions on the agent's final performance.

Resumo

Os avanços da Inteligência Artificial têm uma conexão bem conhecida e aclamada com *jogos de estratégia*. O Gamão, o Xadrez e o Go foram usados para mostrar os resultados dos algoritmos mais célebres de *reinforcement learning*.

Este trabalho explora *jogos multiagente de estratégia* como um ambiente para a investigação de *deep reinforcement learning*. Os *jogos de estratégia* exigem uma análise do seu grande espaço de estado e planeamento a longo prazo para desenvolver uma estratégia vencedora. Essas habilidades são atraentes para serem pesquisadas usando DRL. O fator multiagente nos *jogos de estratégia* introduz mais complexidade e dificulta o desenvolvimento da estratégia.

Um jogo bem estudado desta categoria é o Diplomacy. Este jogo por turnos tem características atraentes a serem exploradas numa abordagem de sistema multiagente. O combate neste jogo é todos-contra-todos, onde cada jogador pode atacar ou defender qualquer outro jogador. Como as ações dos jogadores são simultâneas, o agente terá de criar uma estratégia de longo prazo e também confiar nas relações com os adversários para alcançar o sucesso. BANDANA é um banco de ensaios público baseado no jogo Diplomacy que permite o desenvolvimento de novos agentes.

Este trabalho criou um modelo sobre como abordar *deep reinforcement learning* num cenário de *jogos multiagente de estratégia*. Para exemplificar o uso do modelo, DeepDip, um agente para Diplomacy sem comunicação, foi implementado usando o banco de ensaios BANDANA.

Para criar o DeepDip, *gym-diplomacy*, um ambiente OpenAI Gym de código aberto, foi construído e disponibilizado publicamente. Este ambiente fornece um banco de ensaios para pesquisa de técnicas de *reinforcement learning* no jogo Diplomacy. Num ambiente OpenAI Gym, o agente decide quando mudar para o próximo estado, mas no caso do BANDANA e de sistemas multiagente é o ambiente que decide quando mudar de estado. O ambiente *gym-diplomacy* muda o ambiente do OpenAI Gym para permitir que seja o ambiente a decidir quando passar para o próximo estado.

O ambiente tem compatibilidade com os agentes de exemplo fornecidos pelo OpenAI. Este ambiente inclui o mapa padrão do Diplomacy e variantes adicionais para dois e três jogadores.

Usando o algoritmo Proximal Policy Optimization, o DeepDip foi capaz de ganhar a variante de dois jogadores contra um DumbBot e estava a melhorar os seus resultados na variante de três jogadores, mas era necessário mais tempo de treino para tirar conclusões sobre o seu desempenho tanto no padrão quanto na variante de três jogadores.

Acknowledgements

I would like to thank my thesis supervisor Prof. Henrique Lopes Cardoso for the opportunity and support, and to my colleague José Aleixo Cruz for the mutual support.

I would also like to thank my parents, my sister, and my girlfriend, for providing me with support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis.

Diogo Henrique Marques Cruz

"As time goes on, you'll understand.
What lasts, lasts; what doesn't, doesn't.
Time solves most things.
And what time can't solve, you have to solve yourself."

Haruki Murakami, Dance Dance Dance

Contents

List of Figures

Abbreviations

A2C	Advantage Actor-Critic
A3C	Asynchronous Advantage Actor-Critic
AI	Artificial Intelligence
ANAC	Automated Negotiating Agents Competition
DDPG	Deep Deterministic Policy Gradient
DPG	Deterministic Policy Gradient
DQN	Deep Q-Network
DRL	Deep Reinforcement Learning
h-DQN	Hierarchical Deep Q-Network
MADDPG	Multi-agent Deep Deterministic Policy Gradient
MAS	Multi-Agent System
ML	Machine Learning
NN	Neural Network
PER	Prioritized Experience Replay
PPO	Proximal Policy Optimization
RL	Reinforcement Learning
RPC	Remote Procedure Call
SC	Supply Center
SG	Strategic Game
SAC	Soft Actor-Critic
TRPO	Trust Region Policy Optimization

Chapter 1

Introduction

Artificial Intelligence (AI) has always been strongly linked to games to prove its algorithms. Classical approaches to solve games include search algorithms and the use of complex heuristics designed for each particular game.

Recently, *deep reinforcement learning* (DRL) techniques have been successfully applied to several games. The best-known example is Go [SHM+16], a game believed to be out of the reach of computers because it has a large search space, the strategies do not have an immediate reward, and it would require extremely complex heuristics to develop, but it has been already beaten using DRL algorithms. Such techniques have proven to be generic enough to be applied in different scenarios, including adversarial games and environments that require cooperation between agents to solve specific tasks.

Demonstrating that an algorithm can achieve good results in a game is very important because it proves the veracity of the algorithm in an environment that is known and reproducible by the scientific community. In particular, *strategic games* are ideal environments for testing intelligent algorithms due to their characteristics, including very large state spaces, imperfect information, and simultaneous movements. In this genre of games, the player has to analyze a large scale board to make difficult decisions that make a great impact on the outcome of the current actions, and an impact on its strategy as a whole.

A popular game that has been studied given its complexity and social characteristics is *Diplomacy*. Its most interesting attributes are the huge size of its search tree that makes it difficult to approach using classical search algorithms, the difficulty in determining the true value of a position that translates into the difficulty in creating good heuristics, and the negotiation whose implementation gives a competitive advantage over the adversaries. The fact that opponents can trade throughout the game makes Diplomacy a good sandbox for multi-agent research: while players compete against each other, they also need to make deals and partnerships to increase their probabilities of winning the game.

1.1 Context

AI is deeply connected to Strategic Games (SG). Even before computers existed, Alan Turing made an algorithm and used it to play Chess [Tur53] using Tree Search algorithms. With AI already established as a scientific field, Deep Blue [CHhH02] achieved outstanding results when it defeated the then-reigning World Chess Champion Garry Kasparov proving that AI machines can surpass the human skill in specific tasks.

The Deep Q-Network (DQN) algorithm proposed by Mnih [MKS+15] became known as the first DRL algorithm. With the usage of machine learning, DQN proved its concepts by its results in Atari-2600 console games. DQN achieved results that were at the level of human players. Since then, the games of this console are used as a test environment and the results of DQN as a comparison for new algorithms.

Also with the use of machine learning, DeepMind's AlphaGo was able to beat human players at the level of "grandmaster" in the game of Go, a game that was previously thought to be unbeatable by a machine [SHS+18].

The increasing difficulty of the games that AI proposes to beat shows the evolution in its algorithms. In this way, this work will search for a strategy to approach *strategic multi-agent games* in general with the usage of DRL algorithms to modulate agents. *Strategic games* have large state-action spaces and require the player to plan a strategy that leads to victory, and the *multi-agent system* aspect makes the player have to adapt to the behavior of the opponents.

The combination of *multi-agent system* and DRL has already been proven in works such as Simões [SoLR17] with the conclusion that it is a viable strategy that achieves positive results were with only one agent the results achieved were unsatisfactory. This demonstration was done in environments created for the project, a foraging task and a predator-prey game, both in a small scale environment, the matrix used were 5x5 and 7x7.

Diplomacy is a *strategic game* with particularly interesting features for the *multi-agent system* research due to the large action space, the great branching factor, and the need for interaction between players. It is a turn-based game where all players can attack any player which creates a need to anticipate the changes on strategy of the opponents, but also allows players to support enemy units which creates a relationship of trust between players. The actions are revealed simultaneously with no random factors involved, so knowing how to predict the opponents actions will be an important skill for the players. This game allows negotiation between players to create coalitions making social interaction and interpersonal skills a part of the game's play.

Diplomacy is a proven testbed [FS11] to test the research MAS models and agent architectures. One of the frameworks that uses Diplomacy as its environment is BANDANA. BANDANA is a Diplomacy environment available online that allows the development of new bots, and provides negotiation capabilities between agents using parlance to set the game rules.

1.2 Goals

This project aims to model DRL systems in order to be applied in *strategic games* in general. In order to demonstrate the model, this work also proposes itself to create an agent capable of winning no-press Diplomacy games.

The objectives are listed below as research questions:

- Is DRL appropriate for *strategic multi-agent games*?
- What are the limitations of a DRL model in a *strategic multi-agent game*?
- Can a DRL model learn a winning strategy for no-press Diplomacy?
- What is the importance of the selection of the initial Power in Diplomacy?

1.3 Structure

After presenting an overall idea of the project in Chapter 1, important concepts will be introduced in Chapter 2 that are important for the understanding of this project.

A particular attention will be give to state-of-the-art DRL algorithms in Section 2.3. The most recent DRL algorithm papers will be split into value-based in Section 2.3.1 and policy-based in Section 2.3.2.

There will be an analysis on the available environments to develop DRL agents in Chapter 3.

In Chapter 4, a theoretical and generic model to approach *strategic game* in order to develop a DRL agent will be explained, and in Chapter 4, the model will be applied to no-press Diplomacy. In Chapter 5, the implementation of the model in Diplomacy will be detailed and the results analysed.

Everything will be wrapped up in Chapter 6 that will present a summary of all the work done, the expected contributions for the area, and future work.

Chapter 2

Background

This chapter will cover important concepts for the understanding and development of the project. It will cover the concepts of important areas of this work: DRL and *multi-agent system* .

DRL is a combination of different algorithms and techniques. To better understand it, it is important to understand the parts that when combined generate this area of research. Those areas are *reinforcement learning* (Section 2.1), an approach that learns alone from the interactions with the environment, and *deep learning* (Section 2.2), another approach that improves the learning process by the usage of complex deep artificial neural networks.

With the definitions of RL and DL set, DRL will be introduced in Section 2.3 presenting the most recent algorithms, splitting them into two categories: Value-based in Section 2.3.1 and Policy-based in Section 2.3.2.

A definition of *multi-agent system* will be present in Section 2.4. Section 2.5 will give a definition of *strategic game* and the importance that they have to research. Diplomacy will be introduced in Section 2.6 explaining its rules, listing environments where it can already be played, and previously developed agents because in this work Diplomacy will be used to create a new testbed for DRL research.

2.1 Reinforcement Learning

reinforcement learning (RL) is modeled as a Markov decision process. There is a set of states s, a set of actions a, and a reward for every action in a state. The agent interacts with the environment to learn which actions in which states give better rewards. A visualization of the model can be seen in Figure 2.1.

While trying to discover the optimal solution for the problem, there are two main approaches: value-based and policy-based. By discovering the best value for each state, an agent can choose the actions that will give him the maximum reward.

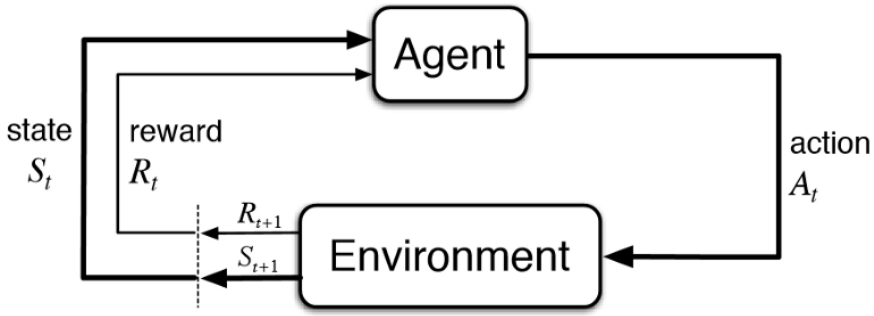

Figure 2.1: *reinforcement learning* model [SB18]

In value-based, the goal is to find the optimal value function. A value function is a prediction of the expected, accumulative, discounted, future reward, measuring the goodness of each state, or each state-action pair. So it learns the expected sum of rewards given a state and an action.

In policy-based, the objective is to optimize the policy. A policy maps a state to an action, or, a distribution over actions, and policy optimization is to find an optimal mapping. So it learns the probability of taking an action in a specific state.

There are two alternative ways of handling the conflict between exploitation and exploration inherent in learning forms of generalized policy iteration [SB18]: on-policy and off-policy methods.

On-policy methods evaluate or improve the behavior policy, e.g., SARSA fits the action-value function to the current policy [Li18]. SARSA evaluates the policy by using samples from that same policy, then greedily refines the policy of the action values.

In off-policy methods, an agent learns an optimal value function/policy, maybe following an unrelated behavior policy. For instance, Q-learning [WD92] attempts to find action values for the optimal policy directly, not necessarily fitting to the policy generating the data, i.e., the policy Q-learning obtains is usually different from the policy that generates the samples.

2.1.1 Q-Learning

In Q-Learning [WD92], a memory table of states and actions, *Q[s,a]*, is created to store Q-values for all possible combinations of s and a, similar to as it is seen in figure 2.2. This Q function represents how good it is to take the action a in state s.

By analyzing the action, it is calculated for that state if there are a reward and the new possible states *s'*. Then by consulting the table, the next action, *a'*, is determined so that the *Q[s',a']* is maximized. So determining the next step can be seen by the target reward expression (Equation 2.1), where γ represents the discount rate on future states.

$$TargetReward : R(s,a,s') + \gamma max(Q(s',a')) \tag{2.1}$$

$$
\begin{array}{cc}
 & \textbf{Action} \\
\textbf{State} & \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\
R = \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{bmatrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{bmatrix}
\end{array}
$$

Figure 2.2: Q-Learning table [McC]

The discount factor discounts future rewards if it is smaller than one. Rewards earned in the future often have a smaller current value, and this alteration may be needed for the solution to converge.

So, as the combination of states and actions increases, the Q table will also increase, which generates a computation requirement that would be too high for the current hardware. Instead of using a lookup table, another approach is to use a value function approximation which will estimate the value function [Sil15]. This function can be a Neural Network (NN), for example.

2.1.2 Policy Gradient

Policy-Gradient is an algorithm with a different approach to RL.

This algorithm does not use a Q-function, instead, it uses a policy, Equation 2.2.

$$\pi_\theta(a|s) = P[a|s] \tag{2.2}$$

The policy learns a map of state to action, and its objective is to find which actions lead to higher rewards and increase their probability.

Instead of planning it thoroughly as Q-Learning does, this algorithm observes the environment and acts upon it. Every iteration, the policy runs to generate a trajectory as represented in Equation 2.3.

$$\tau = (s_1, u_1, s_2, u_2, \ldots, s_H, u_H) \tag{2.3}$$

The algorithm takes the actions of the trajectories while observing the rewards and next states. At the end of the interaction with the environment, the end of the episode, it analyses the result and updates the policy in the direction of the steepest reward increase, favoring episodes with rewards that are greater than the average actions. The comparison of policies is made with the use of an objective function (Equation 2.4).

$$J(\theta) = E[\sum_{t=0}^{H} R(s_t, u_t); \pi_\theta] = \sum_\tau P(\tau;\theta) R(\tau) \tag{2.4}$$

The objective can be seen as searching for the trajectory that maximizes the expected reward (Equation 2.5).

$$max_{\theta} J(\theta) = max_{\theta} \sum_{\tau} P(\tau;\theta) R(\tau) \tag{2.5}$$

And it can be rewritten as a gradient (Equation 2.6) in order to perform gradient ascent on the network.

$$\nabla J(\theta) = E[\nabla_{\theta}(log(\tau;\theta)) R(\tau); \pi_{\theta}] \tag{2.6}$$

By doing several iterations, the policy converges to a maximum. This process can be represented as shown in figure 2.3

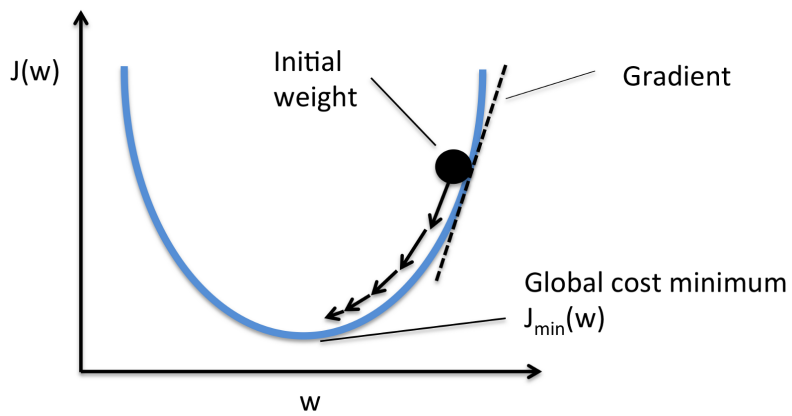

Figure 2.3: Gradient descent representation [S18]

This technique has the problem of not being accurate if the reward function has steep curvature as the steps of training cannot be able to overcome the steep which makes it stuck in a local maximum, and not being able to reach a global maximum.

2.2 Deep Learning

The premise of *deep learning* (DL) is that by increasing the number of hidden layers of a NN, a better output can be achieved for the same input. The architecture of both networks is compared in figure 2.4.

The output of a hidden layer will be the input on another hidden layer, which will generate a different result and analyze different parameters. This increases a lot the complexity of the system, but also generates better results.

As these algorithms require a big data set, and the data requires to be labeled, there's a lot of work needed to create the dataset.

As deep NN are made by combining several NN, a *neural network* is made by combining several perceptrons, so it is important to remind what is a NN and a perceptron.

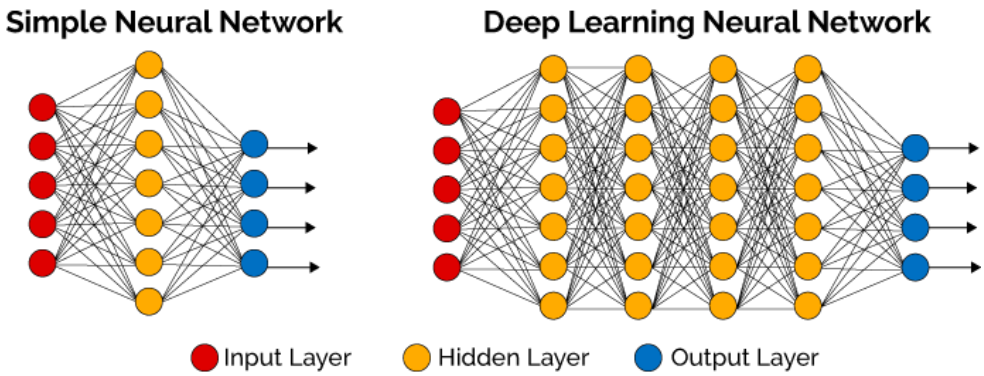

Figure 2.4: Comparison of *neural network* and deep *neural network* architecture [VÍ7]

2.2.1 Artificial Neural Network

A *neural network* (NN) can be seen as a combination of several perceptrons. A perceptron's objective is to divide two classes and to do that it learns the weights and biases of a linear function. It has a set number of inputs and outputs well defined. The inputs need to be more than one and the outputs are values that represent an action or a classification for the inputs. An example of a perceptron for the "AND" logical operation can be seen in figure 2.5.

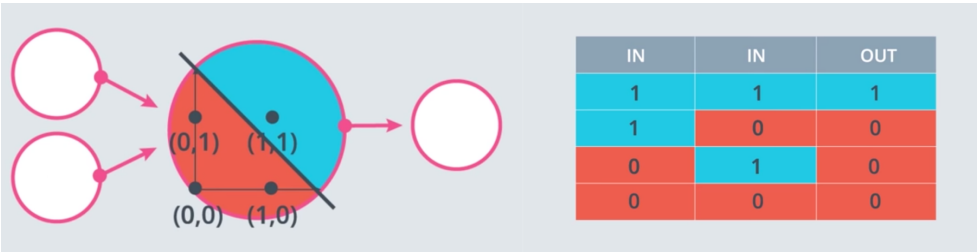

Figure 2.5: Perceptron of "AND" logical operation [Leo18]

If the two classes can only be divided using a nonlinear function, a combination of perceptrons must be used. To that complex organization is given the name of NN. A visual representation of the merging of two perceptrons can be seen in figure 2.6.

In the network, as inputs are received, they are analyzed by a hidden layer, and it creates an output, usually transformed to a percentage, that represents the value for that class. Different connections are created in the hidden layer between inputs and outputs, and each of those connections has a weight w associated for each input received, and a bias b.

The output of the NN is expected to improve by changing its weights and biases as the NN trains in the received data. The training of a NN consists of passing to it inputs that have known expected results and the network will try to adjust its weights and biases to better fit its output to the expected result. So the data that a NN handles have to be identically distributed to not overfit the entire network to a specific class.

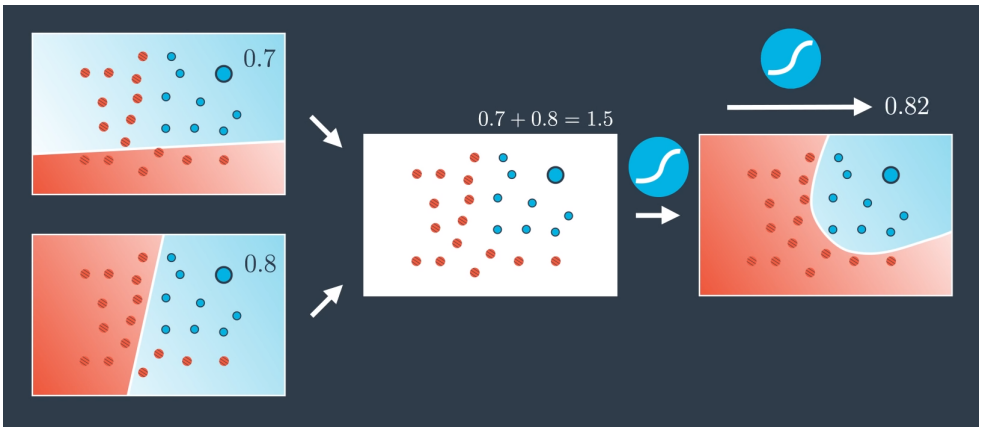

Figure 2.6: Neural Network used to solve a nonlinear function [Leo18]

2.3 Deep Reinforcement Learning

By combining RL, where data does not need to be labeled, with the supervised learning of DL, where the approximate result function has a smaller computational requirement, we obtain *deep reinforcement learning* (DRL). A representation of this architecture can be seen in figure 2.7. This idea generated the Deep Q-Network algorithm that revolutionized the research field.

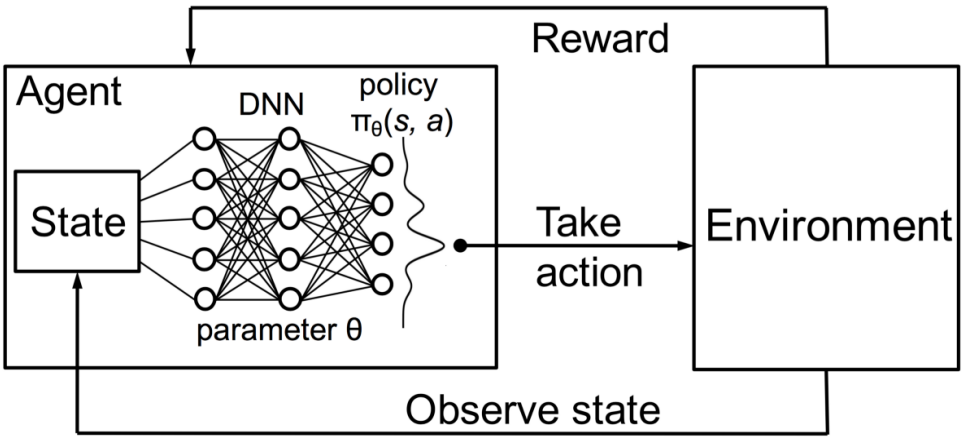

Figure 2.7: DRL architecture [MAMK16]

In this section, the current state of the art of DRL will be presented. So there will be two sections to represent the main areas of focus in the recent DRL studies and improvements being Section 2.3.1 for value-based algorithms and Section 2.3.2 for policy-based algorithms.

2.3.1 Value-based DRL algorithms

These algorithms give a free estimate of how good a particular state is. It can be used for sanity check or other algorithms that depend on this value-based approach.

These algorithms are better designed to train in off-policy. They can be trained on session sampled data from experience replay just as well as their own sessions. This increases the property

of sample efficiency, this means the algorithms require less training data, and less training to reach the optimal strategy.

2.3.1.1 Deep Q-Network

The Deep Q-Network (DQN) algorithm [MKS+15] was designed with the purpose of merging RL with DL. The global architecture would be of a Q-Learning algorithm but the value function would be changed by using a Deep *neural network* to learn its values. To prove the results, the Atari-2600 environment would be used. As every game have a score present on the screen, the reward function can be learned from the output image of the game. The architecture of the system can be seen in Figure 2.8. A Convolution *neural network* was used as it was going to be trained in the output image.

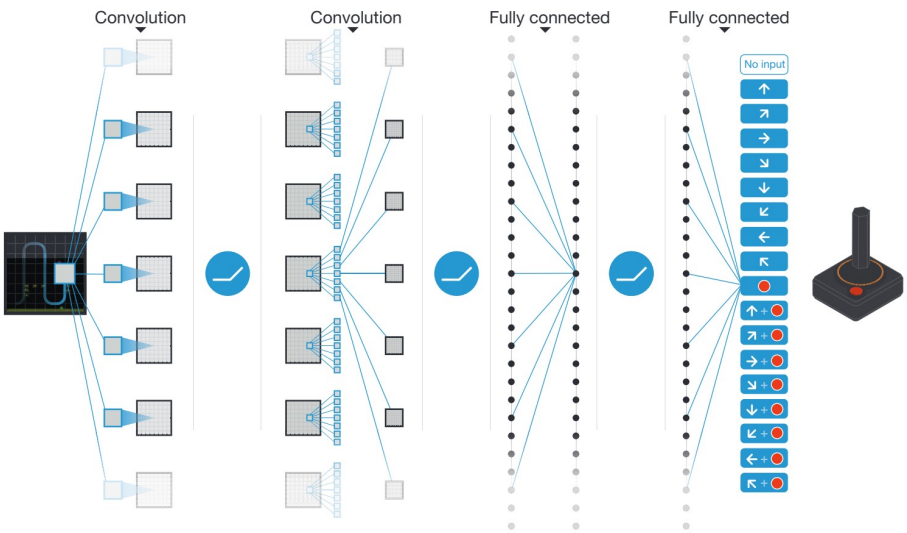

Figure 2.8: Schematic illustration of the CNN usage on the DQN [MKS+15]

The goal on deep Q-networks is to fit the Q-value function using supervised learning but there are some important differences in the algorithms that it's supposed to combine.

In DL, the input samples are randomized so the input class is quite balanced and pretty stable across training batches. In RL, the results improve as the search space becomes known. So, the input space and actions known are constantly changing. In addition, the target value of the Q function is always being updated. Both the input and output are under frequent changes which makes it very hard to learn the Q-value approximator. In order to overcome these difficulties, DQN introduces experience replay and target network to slow down the changes so it can learn Q gradually.

Experience replay stores state-action-reward data in a replay buffer, and sampled randomly, to remove correlations in the data, and to smooth data distribution changes. Then, experiences are sampled uniformly from this buffer into mini-batches to train the network. The input data set is stable and the training samples are randomized, which makes the data set behave closer to the supervised learning in DL.

A target network is also implemented to reduce the correlations between action Q-values and the target. There are two deep networks θ^- and θ. The first one, the target network, is used to retrieve Q values while the second one includes all updates in the training. The new target function (Equation 2.7) uses values from both networks for improved results, as the notion of new knowledge is important for better results.

$$TDtarget = R(s,a,s') + \gamma max_{a'} Q(s',a';\theta_i^-) \tag{2.7}$$

After, for example, an epoch, the target network is synchronized to train on the latest results. The purpose is to fix the Q-value targets temporarily, to make them less volatile, so it doesn't have a moving target to chase.

These two improvements make the idea of "making Q-learning look like supervised learning" [MKS+15] possible. There are also some interesting implementations in the algorithm. The first action made is chosen using a ε-greedy policy. This means that, at the beginning of the training, the possible actions are selected uniformly but as the training progress, the optimal action is selected more frequently. This allows maximum exploration at the beginning, which eventually switches to exploitation.

2.3.1.2 Double Deep Q-Network

In DQN, when the target is being calculated there is an upward bias in $max_{a'}(Q(s',a',\theta_i^-))$ as the current max Q-value may not be the optimal solution. The accuracy of this value depends on what actions and what neighboring states have been explored. As a consequence, at the beginning of the training, there isn't enough information about the best action to take. Therefore, taking the maximum Q value as the best action to take can lead to false positives. If non-optimal actions are regularly given a higher Q value than the optimal best action, the learning will be complicated.

The solution proposed by Double DQN is to decouple the action selected from the target Q-value generation when computing the Q target [vGS15]. The main DQN network selects what is the action with highest Q-value to take for the next state, while the target network calculates the target Q-value of taking that action at the next state (Equation 2.8).

$$TDtarget = R(s,a,s') + \gamma Q(s', arg\,max_{a'} Q(s',a';\theta_i);\theta_i^-) \tag{2.8}$$

2.3.1.3 Prioritized Experience Replay

Experience replay lets online reinforcement learning agents remember and reuse experiences from the past. In the DQN implementation, experience transitions were uniformly sampled from a replay memory. However, this approach simply replays transitions at the same frequency that they were originally experienced [SQAS15]. Replaying all transitions with equal probability, regardless of their significance, is highly sub-optimal.

Prioritized Experience Replay (PER) replays important transitions more frequently and therefore learns more efficiently. PER changes the sampling distribution by using a criterion to define

the priority of each tuple of experience. The objective is to take priority in experiences where there is a big difference between the prediction and the TD target since it means that we have a lot to learn about it. That can be achieved by using replay transitions in proportion to absolute Bellman error (Equation 2.9).

$$Priority = |R(s,a,s') + \gamma max_{a'} Q(s',a';\theta_i^-) - Q(s,a;\theta_i)| \tag{2.9}$$

But simply increasing the priority of training in these cases will lead to always train the same experiences. This greedy prioritization focuses on a small subset of the experience: errors shrink slowly, especially when using function approximation, meaning that the initially high error transitions get replayed frequently. This lack of diversity that makes the system prone to over-fitting. To overcome this issue, a stochastic sampling method that interpolates between pure greedy prioritization, when $a = 1$, and uniform random sampling, when $a = 0$, must be used (Equation 2.10).

$$P(i) = \frac{p_i^a}{\sum_k p_k^a} \tag{2.10}$$

Notice that with normal Experience Replay, a stochastic update rule is used, the experiences are selected randomly. The estimation of the expected value with stochastic updates relies on those updates corresponding to the same distribution as its expectation. As a consequence, the way the experiences are sampled must match the underlying distribution they came from. Prioritized replay introduces bias toward high-priority samples because it changes this distribution in an uncontrolled fashion, and therefore changes the solution that the estimates will converge to. In order to correct this bias, importance-sampling weights can be used to reduce the impact of the experiences seen more often (Equation 2.11).

$$w_i = (\frac{1}{N} * \frac{1}{P(i)})^b \tag{2.11}$$

With this, the weights corresponding to high-priority samples have a small adjustment because the network will see these experiences many times, whereas those corresponding to low-priority samples will have a full update.

2.3.1.4 Dueling Deep Q-Network

This proposed network architecture explicitly separates the representation of state values and (state-dependent) action advantages. [WSH+15] A visualization of the architecture can be seen in Figure 2.9.

The dueling architecture consists of two streams that represent the value and advantage functions while sharing a common convolution feature learning module. The network is changed to have two separate estimators: one for the state value function represented as $V(s)$, and one for the state-dependent action advantage function represented as $A(s,a)$. But, to use this two functions, they can't be simply added, $V(s) + A(s,a)$, that wouldn't be effective as there would be a lack of

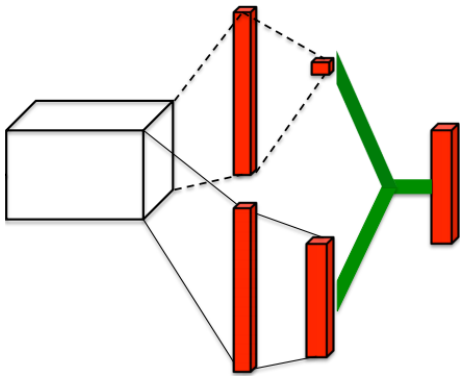

Figure 2.9: Representation of value and advantage learning [WSH+15]

capacity to identify between both functions, which would difficult the process of backpropagation and the network wouldn't be incentivized to optimize V and A independently. The solution is to force the advantage function estimator to have zero advantage at the chosen action, which can be achieved by calculating the difference between the current action, and the next best action (Equation 2.12).

$$A(s,a;\theta,\alpha) - max_{a' \in |\phi|} A(s,a';\theta,\alpha) \tag{2.12}$$

Optimization can also be placed to improve the stability by changing the max function to an average, the advantages only need to change as fast as the mean, instead of having to compensate any change to the optimal action's advantage. The final target can be represented as in Equation 2.13, where α and β represent the parameters of the advantage and value streams, respectively.

$$TDtarget = V(s;\theta,\beta) + (A(s,a;\theta,\alpha) - \frac{1}{|\phi|}\sum_{a=1}^{|\phi|} A(s,a';\theta,\alpha)) \tag{2.13}$$

Intuitively, the dueling architecture can learn which states are, or are not, valuable, without having to learn the effect of each action for each state. This is particularly useful in states where its actions do not affect the environment in any relevant way. The main benefit of this factoring is to generalize learning across actions without imposing any change to the underlying reinforcement learning algorithm.

2.3.1.5 Bootstrapped Deep Q-Network

In Bootstrapped Deep Q-Network, the network explores in a computationally and statistically efficient manner through the use of randomized value functions. Unlike dithering strategies such as ε-greedy exploration, Bootstrapped DQN carries out temporally-extended exploration or deep exploration [OBPVR16].

This network is split into a shared network and bootstrap heads, as it can be seen in Figure 2.10. Each of the heads is initialized with different weights and is going to train on random data from the experience buffer. This means that these heads start out trying random actions, but when some

head finds a good state and generalizes to it, some of the heads will learn from it, because of the bootstrapping. Eventually, other heads will either find other good states or end up learning the best good states found by the other heads. So, the architecture explores well and once ahead achieves the optimal policy, eventually, all heads achieve the policy.

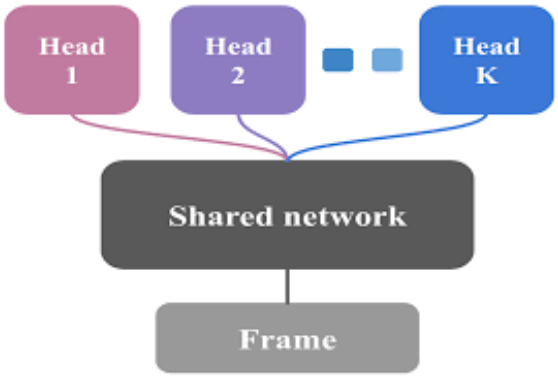

Figure 2.10: Representation of bootstrapped heads [OBPVR16]

2.3.1.6 Hierarchical Deep Q-Network

One of the major problems in RL is to deal with sparse reward channels. Without observing a non-zero reward, it is hard for an agent to learn a reasonable value function. There is a direct relationship between the amount of exploration and observed rewards. Due to the high branching factor in the action space, it can be difficult for the agent to efficiently explore the environment.

Hierarchical DQN (h-DQN) integrates hierarchical value functions, operating at different temporal scales. [KNST16] This concept splits rewards to intrinsic and extrinsic, which represent functions that are alterable and unalterable by the agent respectively. A top-level DQN, the controller, learns a policy over intrinsic goals by maximizing the expected future intrinsic reward. A lower-level DQN, the meta-controller, learns a policy over atomic actions to satisfy the given goals by maximizing the expected future extrinsic reward. This creates an efficient action space for exploration in complicated environments. A representation of this architecture can be seen in Figure 2.11.

2.3.1.7 Noisy Networks

Noisy Networks are neural networks whose weights and biases are perturbed by a parametric function of the noise. These parameters are adapted with gradient descent. [FAP+17] This changes the weights and biases of the neural networks from a value to an expression where they depend on values and that can be learned and ε which can not be learned. The weights can now be seen as $w = \mu^w + \sigma^w \odot \varepsilon^w$ and the biases as $b = \mu^b + \sigma^b \odot \varepsilon^b$. The neural network representation of this new approach is seen in Equation 2.14.

$$y = (\mu^w + \sigma^w \odot \varepsilon^w) * x + \mu^b + \sigma^b \odot \varepsilon^b \tag{2.14}$$

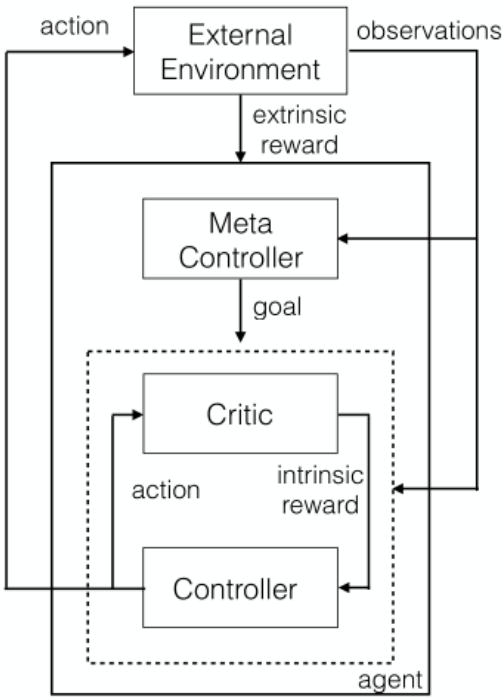

Figure 2.11: Representation of meta-controller and controller [KNST16]

This feature changes the selection of actions of the DQN algorithm as it no longer uses ε-greedy to select actions. Using ε-greedy makes the initial better actions have a higher chance of being picked which in the long run can slow down the training by hiding better actions because they will be explored with less probability. With the new approach, the exploration of the actions is better.

2.3.1.8 Rainbow

Could also be named "Noisy Network multi-step Prioritized Distributional Double Dueling Deep Q-Network" as the work made on this Rainbow algorithm [HMv+17] was of combining several previous techniques to prove that their combination is possible and analyze the influence that each of the techniques has on the final result.

A comparison of the results obtained in each of the algorithms is shown in Figure 2.12.

The conclusion was that all of the algorithms can be combined, but each of them has different weight on the final result. Prioritized replay and multi-step learning were the most crucial, while double and dueling characteristics were the least impactful.

2.3.2 Policy-based DRL algorithms

They have the innate ability to work with any kind of probability distribution, which is very useful when action space is continuous. This makes it easier to specify a multi-dimensional normal

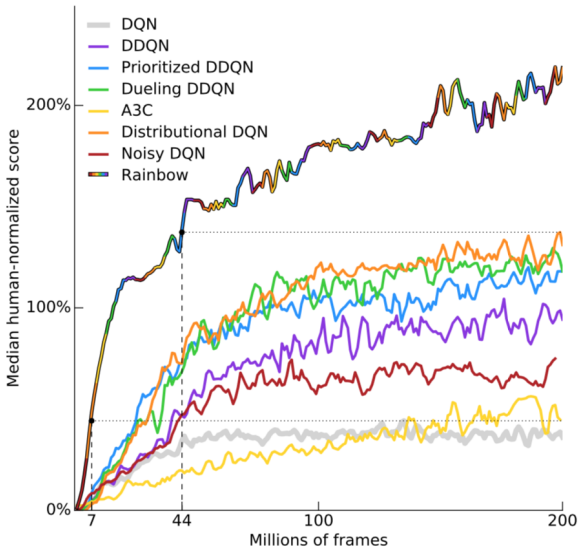

Figure 2.12: Results of several algorithm in the Atari-2600 environment, and the Rainbow combining them to achieve better results [HMv+17]

distribution or a Laplacian distribution, to a particular task.

While value-based calculates a score for every action in every state, in policy-based the action is chosen and the result affects the policy for the next state, making it lighter for big action spaces.

2.3.2.1 Deterministic Policy Gradient

Deterministic Policy Gradient (DPG) [SLH+14] was the first algorithm to implement Actor-Critic architecture similar to the one seen in Figure 2.13.

Actor-Critic combines *policy gradient* with value-learning. This structure has two main components as the name shows, the critic and the actors. There's a single critic on the algorithm that has the job of measuring how good the action taken is using value learning. The actors are the interactions with the environment and can be more than one. Their job is to control how the agent behaves using *policy gradient* .

Along the innovative architecture, there were also some improvements to it. A difference in the actors' implementation on this algorithm is that, instead of waiting for the end of the episode, they update at each step. On this approach, the objective function is rewritten to Equation 2.15.

$$J(\theta) = \int_s \rho^{\mu}(s) Q(s, \mu_\theta(s)) ds \tag{2.15}$$

This algorithm transforms the stochastic *policy gradient* to a deterministic one, which means it outputs a single action when calculating the action choices. A deterministic *policy gradient* is estimated more efficiently than stochastic *policy gradient* so this makes the algorithm more efficient.

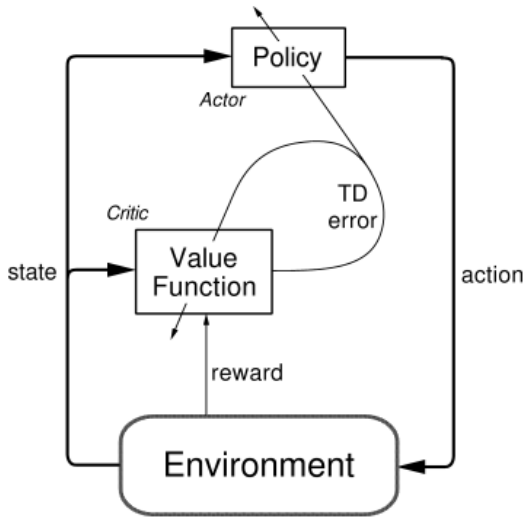

Figure 2.13: Actor-Critic Architecture [SB18]

2.3.2.2 Deep Deterministic Policy Gradient

With the objective of adapting the DQN to the continuous space [LHP+15], the Deep *Deterministic Policy Gradient* (DDPG) algorithm implements an actor-critic approach similar to DPG. To do it, it replaces the critic with a DQN and keeps the deterministic policy gradient in the actors.

2.3.2.3 Trust Region Policy Optimization

Trust Region Policy Optimization (TRPO) [SLM+15] aims at improving the policy gradient algorithm by increasing stability while training. To do this, the idea is constraining how much the policy changes in each iteration by only accepting the change if it is inside a limit of δ. In order to compare the policies, the objective function is rewritten to Equation 2.16.

$$J(\theta) = E[\frac{\pi(s,a;\theta)}{\pi_\theta(s,a;\theta)}\hat{A}(s,a;\theta_{old});\pi_{\theta_{old}}] \tag{2.16}$$

In order to maximize the function, the *Kullback–Leibler* divergence, which is used to calculate the difference between two probability distributions and also known as relative entropy, is used (Equation 2.17).

$$E[D_{KL}(\pi(s,.;\theta_{old})||\pi(s,.;\theta);\pi_{\theta_{old}})] \leq \delta \tag{2.17}$$

2.3.2.4 Asynchronous Advantage Actor-Critic

Better know by "A3C", this actor-critic architecture improves the actors to be multiple and working in parallel while keeping the critic as a shared knowledge base. [MBM+16]

A representation of this architecture can be seen in Figure 2.14.

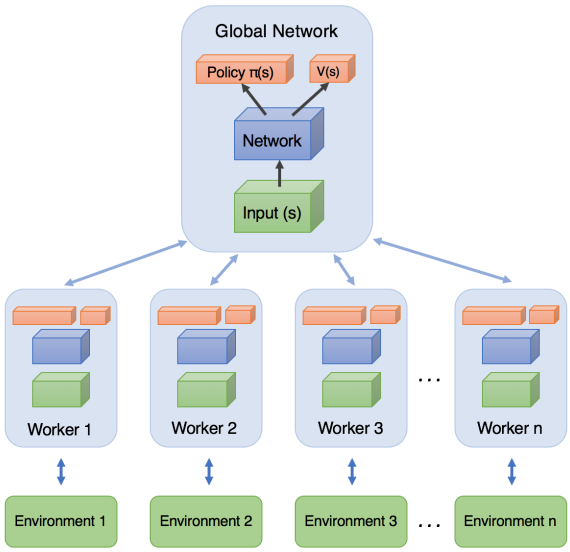

Figure 2.14: In A3C, workers interact independently with different instances of the environment [Jul16]

The actor job is to explore the environment and, in this approach, at the end of each episode, the actor will update the critic with its values. At the beginning of the episode the actor has the updated knowledge from the other actors coming from the critic.

2.3.2.5 Synchronous Advantage Actor-Critic

Even if the A3C made big improvements on the state-of-the-art results, the parallelization didn't handle correctly the cases where the actor's network was outdated compared to the critic network. To overcome this issue, "A2C" proposes that the actors should be synchronous. [WKT+16] This means that the actors should only update the critic when all of them have finished the episode. This guarantees that all of the actors are synced with the latest knowledge, and there aren't conflicts or loss of information at the critic. A representation of this architecture can be seen in Figure 2.15.

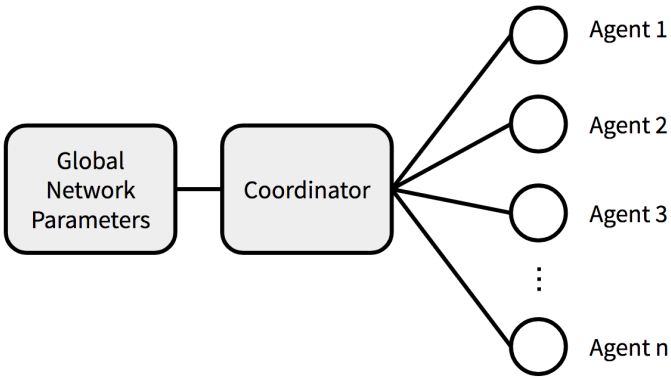

Figure 2.15: A2C implements a coordinator to synchronize the actors with the critic [Wen18]

This improvement made this algorithm more efficient with single-GPU architectures and is faster than a CPU-only A3C implementation when using larger policies.

2.3.2.6 Multi-agent Deep Deterministic Policy Gradient

The Multi-agent Deep Deterministic Policy Gradient (MADDPG) [LWT+17] improves DDPG to be able to learn from multi-agent environments.

The critic learns a centralized action-value function. Multiple distributed parallel actors gather experience and feed data to the same replay buffer.

Multiple agents can have arbitrary reward structures, including conflicting rewards in a competitive setting. So, there are multiple actors, one for each agent, that explore and upgrade the policy parameters θ_i on their own.

A representation of this architecture can be seen in Figure 2.16.

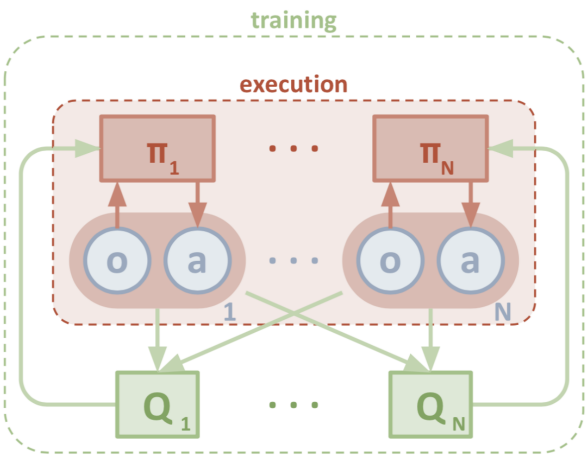

Figure 2.16: Architecture used to support multiple agents [LWT+17]

2.3.2.7 Proximal Policy Optimization

Even if TRPO presented a great improvement with its approach, the implementation was complicated. Calculating the relative entropy and other alterations to the original algorithm made the algorithm less appealing to use. Proximal Policy Optimization (PPO) [SWD+17] uses the same approach but reducing complexity.

The major change was of removing the relative entropy from the objective function and replacing it with a *clip* function. This clip limits the reward to an interval $[1-\varepsilon, 1+\varepsilon]$. The new objective function can be seen in Equation 2.18.

$$J(\theta) = E[clip(\frac{\pi(s,a;\theta)}{\pi_{old}(s,a;\theta)}, 1-\varepsilon, 1+\varepsilon)\hat{A}(s,a;\theta_{old}); \pi_{\theta_{old}}] \tag{2.18}$$

2.3.2.8 Soft Actor-Critic

Soft Actor-Critic (SAC) combines off-policy updates with the stable stochastic actor-critic formulation. The objective of this algorithm is to reduce hyperparamter tuning. To do this, SAC makes the network act randomly and maximizes the expected reward and the entropy H at the same time [HZAL18]. The new objective function is seen in Equation 2.19.

$$J(\theta) = \sum_{t=1}^{T} E[R(s_t, a_t) + \alpha H(\pi_\theta(.|s_t)); \rho_{\pi_\theta}] \tag{2.19}$$

The entropy maximization leads to policies that can explore more and capture multiple modes of near-optimal strategies.

2.4 Multi-Agent System

Multi-Agent Systems (MAS) are complex systems defined by their environment and their agents. The environment is considered the world where the agents live and it manages the outputs. The agents generate the inputs for the environment. A representation of these systems can be seen in figure 2.17.

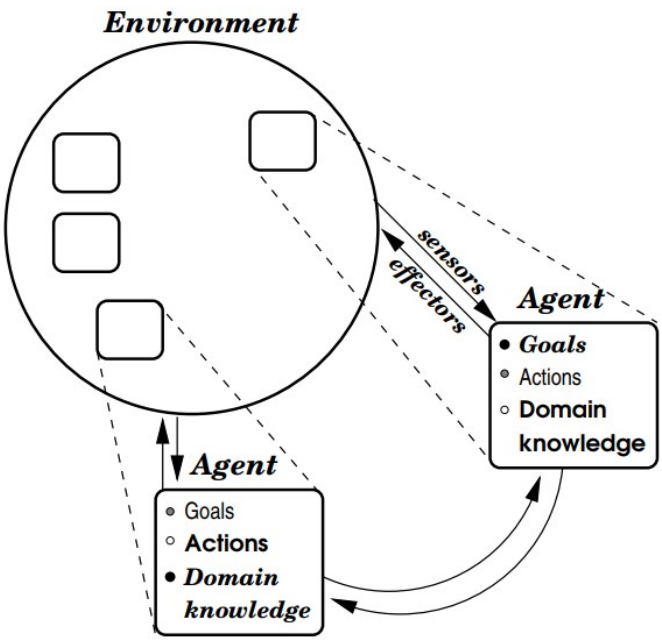

Figure 2.17: *multi-agent system* architecture [BB01]

The environment can be defined by how its state space and the available actions interact with the agents. According to Russell and Norvig [RN09], the environment can be classified using seven parameters.

- **Deterministicness**: The result of an action can be deterministic or stochastic, as it can always change the state to the same new one, or it has a probability of changing it which can lead to different results at each iteration.
- **Staticness**: The environment can be dynamic or static, meaning that it can change while waiting for an agent input or not.
- **Observability**: The environment can be full or partially observable depending on the knowledge that an agent has on it for its task.
- **Agency**: In the situation of more than one agent, the groups of agents should be categorized on their interactions as apathetic, cooperative, or competitive.
- **Knowledge**: The agents can know the environment from a set of rules, or in the case of an unknown environment they must learn the rules of the environment.
- **Episodicness**: The information needed to generate the next action can either be episodic or sequential. In episodic, the agent has all the needed information in the current state, not requiring additional prior knowledge. In sequential, some knowledge of previous states is needed to calculate the current best action.
- **Discreteness**: Its action space can either be discrete when there is a limited amount of spaces and actions or continuous when there is a set of precision to the action space.

The agents of these systems can be a program, a robot, a human, or a team. The teams of these systems may consist of a combination of any type of agent. The agents can be categorized as passive when for a set of conditions that it senses from the environment it has a given action predefined, as active that can choose between actions when the conditions generate a conflict, or as cognitive that improves its behavior as it learns from the environment.

2.5 Strategic Games

The definition of strategy is a plan of action designed to achieve a long-term or overall aim. In a strategic game, the players have to devise a plan that will lead them to the victory. That winning plan might include increasing its own power or undermining the adversary.

In order to create such a plan, the players have to be capable of autonomous decision-making skills. The player must evaluate the current and future state of the game, considering both its pros and its cons of a given action. An usual approach is to use a decision tree to modulate the decision of best action for the given game state.

An essential skill for a player in this family of games is to be unpredictable, with more impact when the players' actions are simultaneous. If the moves of a player are easy to anticipate, it leads to outcomes beneficial for the adversary, creating the need on the player to be creative. In order to implement that creativity, the player must be able to give up on its current strategy and adopt a new one, further implying that it should be open-minded to new strategies in the middle of a game.

Important breakthroughs of AI are associated to SG.

In the early 50's, Alan Turing made an algorithm and used it to play Chess [Tur53] using Tree Search algorithms. In a Tree Search algorithm, the agent calculates a value for each possible next state and chooses the best action possible.

In the 90's, Gerald Tesauro presented TD-Gammon [Tes95] that played Backgammon at the level of expert human players using Temporal Difference (TD) learning. In TD learning, the agent calculates the next moves similar to a Tree Search algorithm, but the evaluation of the game state is changed to include a NN. Every turn, the agent calculates the value of the next actions and calculates the difference to the value it gave in previous games for the same state. Using the NN, the agent then tries to minimize the difference between the values by changing the weights of the NN.

In 1996, Deep Blue [CHhH02] achieved outstanding results when it defeated the then-reigning World Chess Champion Garry Kasparov proving that AI machines can surpass the human skill in specific tasks. Deep Blue used a variation of the alpha-beta search algorithm which is a Tree Search algorithm.

DeepMind's AlphaGo [SHS+18] was able to beat human players at the level of "grandmaster" in the game of Go without handicaps and on the full-sized board. Go was previously thought to be unbeatable by a machine due to its branching factor of 350 which would require massive computational power with the algorithms available at the time. In order to surpass that problem, AlphaGo uses several algorithms. It uses Supervised Learning to predict the best next action a human player would do based on human players data. It then uses RL to self-train in order to increase the focus on winning the game and reduce the focus on predicting the human action. Then, AlphaGo uses the trained NN in the Monte Carlo Tree Search (MCTS) expansion phase. In order to calculate the outcome of an action, the algorithm uses the value generated by the MCTS predition of the end game result, and the value predicted by the NN.

DeepMind researched the game Starcraft 2 and created AlphaStar[1]. AlphaStar was able to win against professional players in the 1vs1 scenario.

All these games were designed for 2 players. Using these algorithms, it is now interesting to approach games with higher player counts and to do this a MAS approach could be used.

OpenAI Five [Ope18] studied the game Dota 2. Their research developed a team of 5 bots capable of playing Dota 2 with a limited pool of characters. In 2019, the team of bots was able to win against the previous world champions of Dota 2. To achieve that win, OpenAI developed Rapid, a new PPO variant, to solve competitive self-play.

OpenAI Five's team achieved great results in the MAS scenario, but needed to use an amount of hardware that is not feasible for a smaller research team, 128.000 CPU cores and 256 GPUs. DeepMind's AlphaStar also used a lot of hardware to train in "many thousands of parallel instances". Diplomacy is proposed as a DRL environment as it provides stimulating challenges but at a smaller requirement of computational power.

[1] https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/

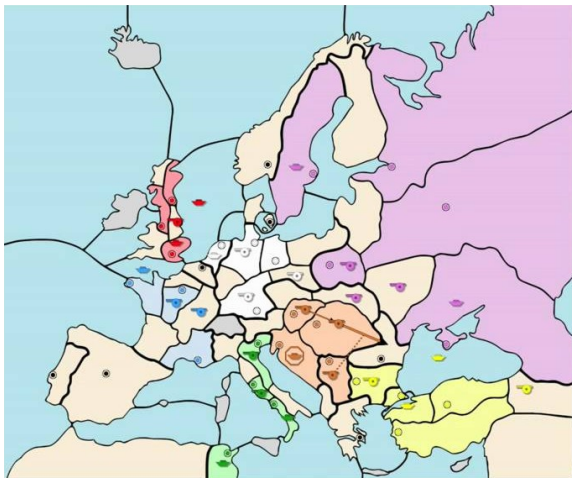

Figure 2.18: Diplomacy map where each color represents a player [dS17].

2.6 Diplomacy

Diplomacy is a *strategic game* where players can attack each others without limitations. This makes it an interesting environment to be analyzed because the players need establish who are their adversaries at every turn since there is no limitation on who a player can attack. A good player will know how to attack and defend, and also more difficult concepts such as trust and betrayal. For example, the agent can make an attack that leads to an adversary to gain an advantage but increasing its trust in the player. Long term planning is fundamental in this game since the strategy revolves around placing the units in strategic positions that allow both to attack and defend.

In this work, Diplomacy was used to create an environment to train DRL agents (see Appendix A [CCLC19]).

In subsection 2.6.1, the rules of the game will be present as well as its interesting features that differentiate this game from other games. Subsection 2.6.2 analyses BANDANA and its predecessor frameworks which allows humans to play Diplomacy, as well as the creation of new bots. Subsection 2.6.3 will present implementations of Diplomacy agents to demonstrate the relevance of this game.

2.6.1 Rules

In Diplomacy, seven players try to conquer Europe. The players represent one of the "Great Powers of Europe" in the years prior to World War I. The game starts in 1901 and the players can choose Great Britain, France, Austria-Hungary, Germany, Italy, Russia or Turkey.

The map has 75 Provinces, and 34 of the spaces are considered as Supply Centers (SC). Each of these Provinces can have more than one Region. There are a total of 121 Regions on the standard Diplomacy map. Each Region has a capacity of 0 or 1 unit. An example of a map for this game can be seen in figure 2.18, where each color represents a player, and several units are represented.

At the beginning of the game, each player controls 3 SC, except for Russia that starts with 4. The player can place as many units as SC that it controls, so everyone starts with 3 units, except for Russia which starts with 4 units.

A player becomes the owner of a SC if he moves one of his units into that space. If that player then moves that unit out of that SC, he will remain the owner until another player moves one of his units into that space. This means that after every SC is captured at least one time, the game can be seen as a zero-sum game, in order to acquire a new SC another player as to lose one.

If the owner of a SC changes, the new owner will receive an extra unit in the next round, and the previous owner will lose one unit. A player is eliminated when he loses all units. The game ends when a player has 18 or more Supplies Centers, or when all players that have not yet been eliminated agree to end the game in a tie.

Each round has two phases: "Spring" and "Autumn". Each of these phases has a negotiation phase followed by an action phase. At the end of "Autumn", the players must update their units to match the number of SC they control:

- *Number of units* > *Number of owned SC*: The player has to disband units to match the number of owned SC;
- *Number of units* < *Number of owned SC*: The player can place units to match the number of owned SC which do not have units there.

2.6.1.1 Negotiation Phase

During the negotiation phase, players negotiate on the commands they will send during the action phase. Usually, players agree not to attack or agree that a player will use some of their units to support a unit of the other player. But more complex plans can be made, a player can say that it will not attack and then not fulfill the agreement and betray the opponent. In this game is important for each player to know who can they trust and whom they should distrust.

2.6.1.2 Action Phase

In the action phase, each player must send a order to each of their units. A unit can have one of 3 different orders: hold, support, move-to.

- **Hold Order**: The unit remains in its current Region to defend it. The hold action is an action always available to a unit.
- **Move-to Order**: Moves the unit from his current location to an adjacent province in order to capture it. The move-to action has a parameter to set the province the unit is going to move to. In Diplomacy there are no set coalitions from the start, so all the players can attack each other without limitations.
- **Support Order**: The unit will not move, but will give extra strength to another unit. The supported unit can be an opponent's unit. The support order targets another unit on the

current turn, it cannot target future orders. The supported order destination must be in range of movement for the supporting unit.

Usually, in turn-based games, the turn is defined by which player can send actions to the units, but in Diplomacy, all players submit their commands simultaneously.

In an attack, the player sends a "move to" command, supported by another unit that received the "support" command, to move the unit to an occupied province. If the defender doesn't have its unit supported it will be forced to retreat, which that the unit has to move to an adjacent province that can not be the one where the attacker comes from. If there are no adjacent provinces unoccupied, the unit is disbanded and is removed from the game.

Only one unit may occupy each province. If multiple units are ordered to "move to" the same region, only the unit with the most support moves there. If two or more units have the same highest support, a standoff occurs, and no units ordered to that region move. A unit ordered to give support that is attacked has those orders canceled and is forced to "hold", except in the case that support is being given to a unit invading the region from which the attack originated.

2.6.1.3 Model Specification

In order to model this game as a *multi-agent system* , some assumptions must be made beforehand on the environment and action space.

- **Deterministic**: The action of an agent in two equal states will always lead to the same result as there is no randomness factor in this game.

- **Static**: The actions of the players are stored and are executed simultaneously so the environment doesn't change without the agent's input.

- **Partially Observable**: Even if the board is fully observable, the result of a taken action depends on the other players' actions.

- **Agency**: The game can be played by multiple agents but their interactions will be limited as they won't cooperate. This mode without cooperation is called "no-press". The environment can be seen as competitive as the objective is to win alone the game and to do that the chances improve if the opponents get worse results.

- **Sequential**: There is a strategy underneath each player action that is being planned and executed in the course of the game. As the player has to adapt to the adversary by assessing its past actions, the game is considered a sequential game.

- **Discrete**: There are 34 spaces, 7 players, and a unit can't be split, meaning that the action space is finite and the environment is discrete.

2.6.2 BANDANA

BANDANA[2] (BAsic eNvironment for Diplomacy playing Automated Negotiating Agents) [dS17] is a Java framework developed to facilitate the implementation of Diplomacy playing agents capable of negotiation and was released with D-Brane, an agent capable of negotiation, to demonstrate the use and capabilities of the framework. Since its release, a Diplomacy tournament is made to develop agents for this framework within the Automated Negotiating Agents Competition (ANAC) [dBA+18] that has been held annually.

BANDANA extends the DipGame[3] [FS11] framework, providing an improved negotiation server that allows players to make binding agreements with each other. The DipGame environment was released with the purpose of playing and creating agents for Diplomacy, providing both online and locally tools for the propose. It follows the guidelines of DAIDE and uses a client-server structure.

DAIDE[4] (Diplomacy Artificial Intelligence Development Environment) is a Diplomacy environment with negotiation capabilities. They created their own communications model, communications protocol, and language in which diplomatic negotiations and instructions can be expressed. They have also created an arbitrator, a set of libraries to help develop new agents, and new agents to test the environment. It uses a client-server structure which was developed in order to foster the development of artificial agents to play Diplomacy.

In BANDANA, two types of Diplomacy players can be created, one can build a player that only makes tactical decisions, or a player that also negotiates with its opponents. Tactical choices concern the orders to be given to each unit controlled by the player. Negotiations involve making agreements with other players about future tactical decisions. In the original Diplomacy game, these negotiations are non-binding, meaning that a player may not respect a deal it has reached. However, in BANDANA deals are binding: a player may not disobey an agreement it has established during the game. The removal of the trust issue that non-binding agreements bear simplifies the action space of mediation.

Tactics and negotiations in a BANDANA player are handled by two different modules. They may communicate with each other, but that is not mandatory. A complete BANDANA player consists of these two modules, that should obey to a defined interface.

To play a game of Diplomacy, BANDANA has a dedicated Java class which launches a game server and initializes each player. The game server is responsible for communicating the state of the game to the players and for receiving their respective actions. In the case of negotiation, BANDANA uses a separate server with a predefined message protocol that allows mediation. Players do not communicate directly with each other. The game continues until someone wins, or a draw is proposed and accepted by all surviving players.

[2] http://www.iiia.csic.es/ davedejonge/bandana/
[3] http://www.dipgame.org/
[4] http://www.daide.org.uk/

2.6.3 Agents

Diplomacy is an environment interesting to develop agents [FS09], and as such there already some implementations for the game and environment that help the creation of new agents.

DumbBot [Nor05] is a simple bot but with great results that can win against humans. It works in two stages: it calculates a value for each province/coast and then creates an order for each unit, based on those values. It has been used as a comparison for the other bots, and will also be used to train DeepDip.

DarkBlade [RMSL09] is a multi-agent system with agents organised in a 2-layer hierarchy designed for DAIDE. It improves the strategy of DumbBot with values on unit threat and a threat history. It also uses personality traits to change the behavior of the agent.

DipBlue [FCR15] is another agent for the Diplomacy environment. Using a more modulated approach, it splits its architecture into: *Agreement Executor*, *Word Keeper*, *Map Tactician*, *Fortune Teller*, and *Team Builder*. All of the submodules are used on a main module *Adviser*, while *Agreement Executor* and *Work Keeper* are analysed together in the *Negotiator*. The *Negotiator* handles the relations with the other agents, while the *Adviser* handles the strategy of the agent. *Map Tactician* is based on DumbBot and evaluates the map in player power, amount of enemy units in each position, and a value for the provinces. *Fortune Teller* analyses the success of action in an optimistic view, it disregards chain actions caused by other players actions. *Team Builder* handles support moves to help a neighbor do its action with success.

The agent created for BANDANA was D-Brane [dS17], and it has an architecture split into two main components: the strategic module where the orders to the board are made and a negotiation module. The strategy proposed is to divide the game into mini-games of conquering each SC and then combine the strategies of each SC to form the strongest final strategy.

Tagus Bot [de 17] was designed for the DAIDE environment and uses the strategy of DumbBot but improves it with negotiation skills and "Opening Libraries" that control the first rounds of the game depending on which country the agent is controlling.

AlphaDip [MLC18] uses a strategy based on D-Brane and the NB^3 algorithm [dS15] to search for the best moves. The strategy was improved by using the concept of hierarchy, similar to DipBlue, and it implements a *President*, a *Strategy Office*, a *Foreign Office*, and a *Intelligence Office*. The *President* coordinates the other sub-modules and has the task of making the final decision on which action to take and send it to the environment. The *Strategy Office* tries to maximize the player number of controlled SC with the usage of the NB^3 algorithm. The *Foreign Office* tries to create coalitions with the opponents, and commitments for the current round. The *Intelligence Office* studies the trustworthiness of the opponents by giving them a trust value that increases over time but decreases when the opponent attacks the player. The *Intelligence Office* also tries to predict the goal of the opponent by predicting which SC wants more based on direct attacks to the SC.

Chapter 3

Deep Reinforcement Learning Environments

There are already other environments prepared to develop DRL agents and some are also prepared to develop DRL in a MAS but, as will be seen, there is a lack of a testbed that supports DRL for MAS with the need to negotiate as the BANDANA environment can provide.

Well-known environments, such as OpenAI in Section 3.1 and DeepMind in Section 3.2, will be presented in this Chapter.

3.1 OpenAI

The most famous environment to test DRL agents is Atari-2600. It was the testbed used for DQN using ALE (Arcade Learning Environment) [BNVB13] and since then it established itself as the go-to environment for DRL agents and it was incorporated by the team of OpenAI on Gym [BCP+16] for an easier time to set up. Examples of Gym environments can be seen in figure 3.1. Gym also offers other environments from simple text based games or algorithms, to 2D and 3D robots. The 3D robots use the MuJoCo physics engine. There is also the Debate Game [ICA18] that lets two agents try to persuade a human judgment about the content of an image by argumentation.

The team of OpenAI also has OpenAI Five [Ope18] which is Dota 2 game environment that handles multiple agents that have to coordinate themselves and they are trying to accomplish that with the use of DRL algorithms, but it isn't open-source so it's not available to the public.

3.2 DeepMind

DeepMind's most famous environment is Go, a game they have achieved super-human levels that were previously thought to be unbeatable by a machine [SHS+18]. They also provide Chess and Shogi replays for training, DeepMind Lab [BLT+16] which is an 3D environment of a single-player game for the agent to explore, AI Safety Gridworlds [LMK+17] to train agents that need to

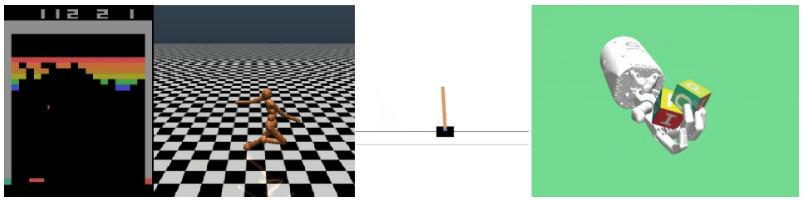

Figure 3.1: Examples of OpenAI Gym environments. From left to right: Atari-2600's Breakout-v0, MuJoCo's Humanoid-v2, CartPole-v1, and HandManipulateBlock-v0.

explore without endanger themselves, and a 3D environment called Control Suite [TDM+18] that is similar to the 3D environment of OpenAI's Gym.

Figure 3.2: Deepmind studied Chess, Shogi, and Go.

DeepMind also created an environment to play Starcraft 2. StarCraft II Learning Environment (SC2LE) [VEB+17] is an environment also available to develop a DRL in a *multi-agent system* scenario but without negotiation between competitors. It has a large action space involving the selection and control of a large amount of units. In this game, a professional player will do more than 500 actions per minute. This environment includes sub-environments where the agent can train specific actions as seen in figure 3.3.

Figure 3.3: Sub-environments present in SC2LE.

3.3 Rogueinabox

Rogueinabox is an environment that allows an interaction with the Rogue game [APM+17] which creates the possibility of developing DRL agents [ACS18]. In this game, in a grid map, the agent has to find the stairs to delve deeper into a dungeon while collecting coins and fighting monsters. A representation of the map can be seen in figure 3.4 The Rogue game is different in each start as the agent does not know what it will find in each floor of the dungeon and that makes it an interesting environment as the agent must adapt to each floor.

Figure 3.4: An image of the representation of Rogue in the rogueinabox environment. The player is represented by a "@" and has to find the stairs "%".

3.4 OpenSim RL

OpenSim RL [KMO+18] is an environment created by a team at Stanford University. Here the agent will try to learn how to move and walk around. To do this, it will have to control a physiologically plausible 3D human model in OpenSim, a physics-based simulation environment, that can be seen in figure 3.5.

This environment had a competition at NIPS 2017 where the goal was only of learning how to move in a 2D environment.

In the NIPS 2019 competition is measured the capacity to walk around in a 3D environment.

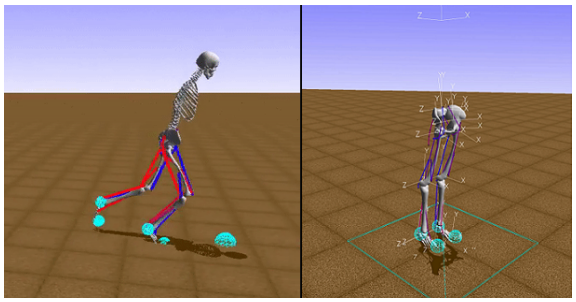

Figure 3.5: OpenSim is a physics-based simulation environment with 3D rendering. The environment where the goal was to learn how to move is represented in the left of the figure. The environment with the goal of learning to walk around is represented in the right of the figure.

3.5 PyGame Learning Environment

PyGame Learning Environment (PLE) [Tas16] is an environment where the agent will interact with small arcade games. It provides 9 arcade games such as Pong, Snake, FlappyBird, "Monster Kong" which is a spinoff of the original Donkey Kong game, and "RaycastMaze" where the agent must exit a labyrinth in a 3D environment. An example of the rendering of 6 environments of PLE can be seen in figure 3.6.

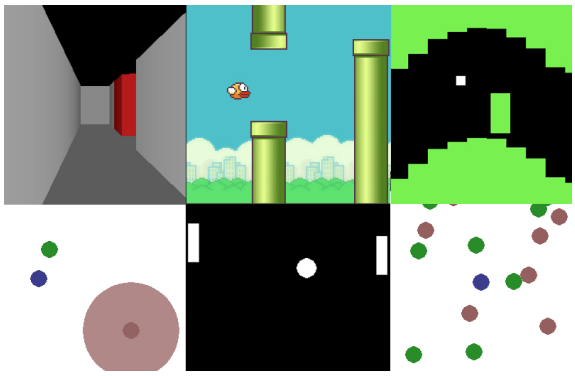

Figure 3.6: Example of the environments that PLE provides. In the figure can be seen, from the left to the right, RaycastMaze, FlappyBird, Pixelcopter, PuckWorld, Pong, and WaterWorld.

3.6 Unity Machine Learning Agents Toolkit

Unity Machine Learning Agents Toolkit (ML-Agents) [JBV+18] is an Unity plugin to create and use Unity environments for training agents. It provides a set of 2D, 3D and VR/AR games ready for use that can be seen in figure 3.7 and environments created by the community.

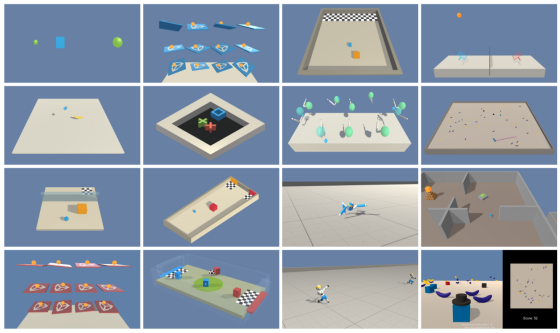

Figure 3.7: ML-Agents provides several environments made in the Unity engine.

Chapter 4

Model to apply DRL in Strategic Games

An efficient solution to simple games such as Atari-2600 [MKS+15], and to complex games such as Go using human knowledge [SHS+18], is proven to be DRL. Along with the game of Go that requires a solid strategy to win the game, *strategic games* in general also require this capability.

In *strategic games* the conditions of the environment are complex and include additional challenges such as imperfect information due to the multiple agents that make simultaneous actions creating entropy in the environment. In multi-agent *strategic games* there is also social skills, such as negotiation, that an agent can use to improve its results.

Each game has its characteristics that should be analyzed to properly model the DRL algorithm so there is not an absolute procedure that will always work, but some guidelines can be made and an example using the Diplomacy game will be presented.

In this chapter will be developed a general model of how to handle correctly a *strategic game* in a DRL approach with which the agent can achieve the success of winning the game. In Section 4.1, a needed definition of "smallest unit" will be presented that will be used in other sections. In Section 4.2, the state of the environment will be analyzed on how to adapt it to the algorithm, and the case of Diplomacy is analyzed in Section 4.2.1. In Section 4.3, some details about how to create the reward function for the environment will be thought about, and the reward functions studied for Diplomacy will be discussed in Section 4.3.1. In Section 4.4, the action space will be analyzed with special attention to the multiple units case and to the Diplomacy case in Section 4.4.3.

4.1 Smallest Unit

In the case of the Atari games the player was a single unit, and as such, the actions of the output of the model represent the actions that the player will take. For more complex games, as in the case of *strategic games* , the game might have more than one unit that the agent has to control, and so there is a need to define the "smallest unit" that the agent controls.

This concept of "smallest unit" represents what the agent is controlling when interacting with the environment. The "smallest unit" can be a single character or a complex group of characters. In every action that interacts with the environment, the agent will have to send its commands to its "smallest unit".

For example, in the case of a robot in a grid map that can move ‘up’ or ‘down’. If the agent is only controlling a single robot, its action would be ‘up’ or ‘down’. In this case, the "smallest unit" of this agent would be a single robot.

If it is controlling a group of these robots, the agent would have to generate an action for each robot, and it could be represented as an array where each element is the action for a robot. If it was controlling 4 robots, an action could be [up, up, down, down]. In this case, the "smallest unit" of this agent would be the group of 4 robots.

If the agent is controlling an area where there could be robots in it, the area would have to be divided into segments. Each of those segments would control the robot when it is in its area. If it was controlling 4 vertical areas, an action could be again [up, up, down, down]. This system is more complex than optimal for less units but it allows the agent to control in a *multi-agent system* an area where there might not always be a robot present.

In the case of Diplomacy, the status of each of its Regions gives the representation of the game at a given time. This representation can be seen as the areas representation. Each Region can have a single Unit so the player controlling the unit can send its order to the Region and it will be applied to its Unit. The order can be represented as an integer in the interval $[0, 2 * Regions]$.

Notice that a more complex "smallest unit" will lead to a bigger action space.

4.2 Input

A DRL algorithm needs a stable input to represent its state. This means that the format of the input must be the same across different steps of the algorithm. For example, in the case of models that handle images, usually the images are resized so that the model can learn from every image that the users passes to it.

In *strategic games* , the game will have to be transformed into a standard format. There could be the need to create an interpretation of the game and define on what is that the state represents, but a simple interpretation can be that a state is what the agent can access. If the environment has a graphic representation, the state could be the information of the screen, a image. If it is a board game, the state can be a representation of the board as a grid where each of its elements gives information about the state of the board.

The information of the state will have to be created according to the "smallest unit". The state can contain information about the position and status of each "smallest unit". The complete state will show details on each of the smallest units, having in the empty spaces some placeholder information to have a valid format.

4.2.1 Diplomacy State Representation

The status of each Region gives the representation of the game at a given time.

This information will be transformed from the Java Object into an array of integers to allow a smaller message to be sent in the communication between frameworks, and for the DRL algorithm that needs a numeric representation of the state.

Each region can have a Supply Center, a Owner, and a Unit. So, each region will be represented by a set of arrays of dimensions $[2, number\ Powers + 1, number\ Powers + 1]$.

- **First Parameter**: Represents if the Region has a Supply Center. Works similar to a Boolean in an integer representation.
- **Second Parameter**: Represents the Power that owns that Region. Owning a Region is an important aspect of the game as it reveals the movements the Power had in previous turns, and if a Region has a Supply Center, owning it gives one more Unit to the Power which is very important to capture more Regions and Supply Centers. At the start of the game most of the Regions do not have a owner, so there is a need to have the representation of not owned Region which is given by the value 0.
- **Third Parameter**: Represents the current Unit placed in the Region. Only a single Unit can be at a given time in a Region, so its representation is important for the agent to understand the current movement of the adversary and its strategy. Most of the Regions will not have Units during most of the game so there is also the need to represent the absence of a Unit in the Region using the 0 value.

Every power has the same rights over an owned province, and the unit's movement is the same independently of the power that owns it. In this way, to represent the orders as actions that the algorithm can learn, the actions will be power agnostic, the power will not interfere directly on the action representation, and so the agent's Power will always be represented by value 1. The agent will not know if it is playing for example as England or Russia, it will just know that his units and the Regions it owns are the ones represented by a 1.

4.3 Reward Function

As the agent interacts with the environment it will receive a reward to represent the impact it had on the agent's goal.

A simple reward function is to give a reward of value 1 when the agent wins and -1 when it loses. The best feature of this function is that the agent will learn its main goal of winning and not be distracted with some side effects of the training such as preserving every of its units or try to make short games. This function would work on simple environments since in a few actions the agent would get a feedback of their impact, but in larger environments it can make the training time unbearable. So, even though it is a simple function, it may be too scarce for the agent to learn how to win the game.

There are some games with a system of positive score or victory points that can give the agent an immediate impact of the actions it took will have on its goal. These score systems, where points cannot be lost, are a helping tool since the state should contain the information on what happened to trigger the increase of the score. Such a feature should be represented on the reward function as it will help the training of the agent. For example in the case of Atari, the environment used for the DQN paper, the agent has access to an immediate score on the screen that increases in a good action. Similar systems can be found in *strategic games* particularly in Eurogames where actions generate points and at the end of the game the player with the biggest score wins making the player choose to win less points in the early actions to prepare for big points in later actions or get an early advantage that the opponents won't be able to reach.

Important to notice that *time* should not impact the value of an action. Getting a good score in the early game might not mean that it is a good strategy in the long term, and giving a reward for being alive for longer might not mean that it is getting closer to a winning condition.

So the reward function must focus on the goals, win the game, and not on a specific task.

4.3.1 Diplomacy Reward Function

DeepDip's objective is to win the game. To achieve this, it is required to conquer a total of SC_{toWin} as given by equation 4.1.

$$SC_{toWin} = SC_{total}/2 + 1 \tag{4.1}$$

A straightforward approach to defining a reward function is to give a positive reward for a win, a neutral reward for a draw and a negative reward for a loss. If the game does not end in a draw, the agent will receive a reward equal to its number of Supply Centers plus a bonus or penalty depending on the end game result. If it wins the game, the agent receives an extra positive reward of $+SC_{toWin}$, while when losing it accumulates a penalty of $-SC_{toWin}$.

- **Eliminated**: When the player loses every SC it owned, it gets eliminated and receives a reward of $-SC_{toWin}$.
- **Lost**: When another player owns enough SC to win the game, the agent receives a reward of $[-SC_{toWin} - 1, -1]$.
- **Draw**: When DeepDip can establishes a Draw agreement with another player, the agent receives a reward of $[1, SC_{toWin} - 1]$.
- **Win**: If it wins the game, the agent will receive a reward of $[2 * SC_{toWin}, 2 * (SC_{toWin} - 1) + SC_{toWin}]$.

There were experiments with other reward functions but the results were always worse than the end game result function.

- **Current Owned SC**: In addition to the end game reward, this reward function gave a score each turn equal to the number of SC the player had at the moment. The agent started to learn

but would only try to have a long game with lots of captured SC and did not care for a win or loss.

- **Fixed reward on Capture**: Another trial was made using a reward of value 1 when the agent captured a SC. This did not properly work as the agent did not understand its goal because it did not have any downside and would also lead a reward dependent on time so the agent would try to enter a $loseSC - recapture$ loop.

- **Exponential on Capture**: In this reward function the agent would get a reward of value equal to $3^{CurrentSC}$. As the best agent is calculated based on the mean reward of a set number of steps, this reward system was too volatile as in a single good game would have a decisive impact on the decision of the best agent.

4.4 Output

The agent will output an action to interact with the environment. This action will be defined by the environment's action space.

4.4.1 Multiple Units

In the game of chess, each agent on its turn has one action of moving a piece, but, in more complex *strategic games* , the agent can have to create actions to multiple units simultaneously. This is a big increase in complexity since the model will have to output an action more complex that includes in itself actions for each unit.

So, if the agent is controlling one unit in its turn it will have to output an action of complexity n, where n is the different possible actions that unit is capable of, as can be seen in Equation 4.2.

$$Action_n = A1, A2, ..., An \tag{4.2}$$

On the other hand, if the agent is controlling m units with the same action range, it will have to output an action of complexity n^m as represented in Equation 4.3.

$$Action_{n*m} = A1_1, A2_1, ..., An_1, A1_2, A2_2, ..., An_2, ..., A1_m, A2_m, ..., An_m \tag{4.3}$$

4.4.2 Unit Action Complexity

In a SG, the units do not have to be all the same, each unit can have its own properties and capabilities. This complexity, if reflected in the possible actions the unit can take, will matter when designing the action space of the agent because, in this model, every unit must have the same number of actions. The action space of a DRL agent must be consistent throughout its process. This sets the action space to be defined by the most complex unit. In the case of a unit with a smaller action space, its action space will be increased to the maximum size and filled with

"do nothing" actions. This need of the action space to be a square matrix makes it important to analyze the "smallest unit" and simplify the units.

For example in chess, a pawn can only move forward, so its action will be to move or not to move, meanwhile the bishop can move diagonally in any direction and any amount of spaces to a maximum of 7 spaces. The action space of the pawn would be $[move]$ for a total of 2 possible actions, while the bishop would be $'none' or [direction, amount]$ for a total of $1+4*7=29$ possible positions where it could move to. But, in this model, each unit has to have the same number of actions, so the bishop action space would impact the pawn action space.

Care in mind that the action space of the environment affects the choice of the algorithm. Value-based algorithms do not work on big action space environments as the memory needed would increase exponentially, so a policy-based algorithm must be chosen.

Also, as there can be actions that are invalid in a given state, the actions to be sent should be analyzed and modified to send to the environment. The environment can receive an invalid action, discard it, and ask for a new one, making the process of reaching the end game slower than expected, and this would degrade the training process as the agent would have low feedback on its actions. In order to not stop the training process, the agent needs to always have a valid action that can be sent and is not heuristically calculated. After the model outputs its predicted action, evaluate it and check if it is valid. If it is, then it can be used, or else use the always valid default action. This makes the model learn the value of each of its actions without the impact of human-made heuristics.

4.4.3 Diplomacy Action Space

DeepDip will give direct orders to control its units. There are 3 different orders that can be sent to a unit. These orders have to be adapted from the BANDANA representation to a numeric representation so that the algorithm can train to know what orders to send. The representation will be the same across all the maps, but they will be appropriately resized to match the map specification.

In the case of the "standard" map, the transformation of representations is presented:

- **Hold Order**: (Power Region) HLD

  This order does not have a parameter so there is just 1 possible order for each Region.

  Example: (FRA BREAMY) HLD $\rightarrow$ 0

- **Move-to Order**: (Power Region) MTO Destination-Region

  The Move-to order has one parameter to the destination Region. So, for each Region there could be 121 different Regions to be passed as the parameter. In the map there is not so many borders to any Region, but that will be easy for the agent to learn that most of those Regions do not produce any good result different than what the hold order does.

  Every Move-to Order that is invalid will be transformed in a Hold Order.

The values in $[1, 121]$ will represent the possible Move-to Orders.

Example: (AUS BUDAMY) MTO RUMAMY $\rightarrow$ 54

- **Support Order**: (Power Region) SUP Move-to Order

  Only one Move-to Order can be given to Region, all the other units can not have a Move-to Order to that same Region, but they can support to increase the Power of that order. In such way, the parameter of the Support Order can be represented as the destination Region of the Move-to Order - e.g., the Move-to Order "(AUS TRIAMY) MTO VENAMY" will be simplified to just "VENAMY".

  Every Support Order that is invalid will be transformed in a Hold Order.

  The values in $[122, 242]$ will represent the possible Support Orders.

  Example: (AUS ADRFLT) SUP (AUS TRIAMY) MTO VENAMY $\rightarrow$ 191

The agent has to create an order for each Region of the map. There might not be any of its Units in the Region but the algorithm needs a fixed size action space to train. Even if there are a lot of possible combinations that will not have impact, as most of them will produce invalid Order and as such will be transformed into Hold Order, with training DeepDip can understand the lack of importance of those Orders and opt for better ones.

Chapter 5

Gym's Diplomacy Environment: Setup, Experiments, Analysis

In order to implement the model introduced in Section 4 and to prove its concepts, an OpenAI Gym [BCP+16] environment was created that allows the development of BANDANA (see Section 2.6.2) agents capable of learning how to play Diplomacy (see Section 2.6) in the no-press variant.

The BANDANA framework is a game engine that allows the development of agents for Diplomacy. It has a tournament feature that allows a big number of games to be continuously played which is relevant for the training of a DRL agent since there is reduced down time in restarting the game when it finishes.

DeepDip is an agent created for the BANDANA framework that uses the structure defined by OpenAI Gym to create the orders that will be sent to the game. OpenAI Gym is a framework that defines a standard structure that a DRL agent should be constructed with, which makes the structure of the agent easy to recognize by any developer and makes the agent compatible with any environment.

As OpenAI Gym is built on Python, it is easy to connect to the current state-of-the-art DRL frameworks, such as Tensorflow [ABC+16] and PyTorch [PGC+17], with Gym agents and make use of the DRL techniques that those frameworks provide. OpenAI Gym also provides the developers with a set of example algorithms which simplifies the test of a new environment.

With all this in mind, creating a Diplomacy environment for Gym will make it easier to implement RL or DRL agents that could play this game and analyze their behavior.

The design proposed is represented in Figure 5.1. It consists of abstracting the Diplomacy game information provided by BANDANA to match the OpenAI Gym environment specification. This custom environment encapsulates an adapted implementation of a BANDANA player and the communication between all the necessary processes.

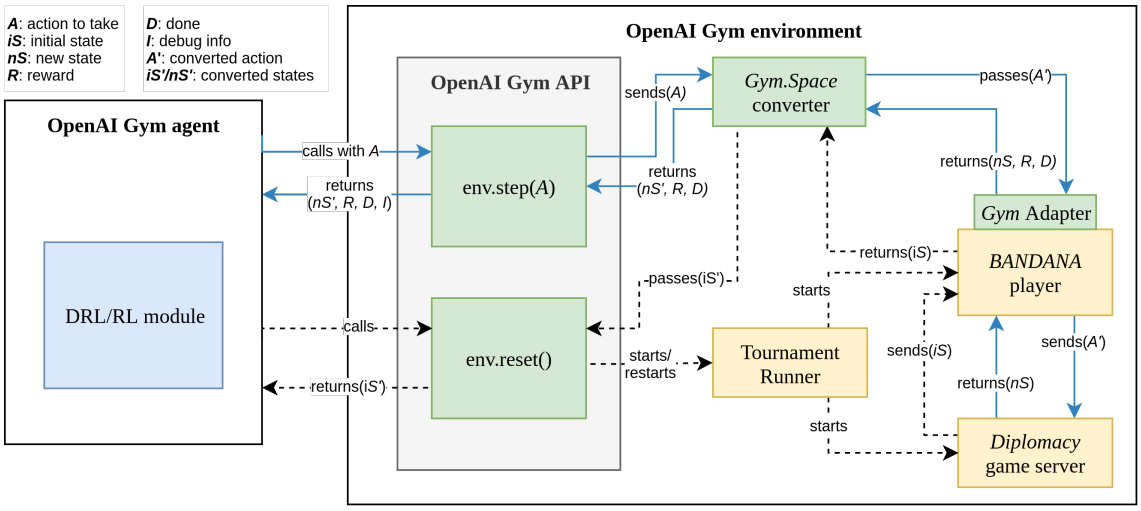

Figure 5.1: Conceptual model of the Open AI Gym Diplomacy environment and agent.

Section 5.1 describes the setup that was created to adapt BANDANA to allow the training of DRL agents. In Section 5.2, the experiments will be detailed. In Section 5.3 the results will be analyzed.

5.1 Setup

This Section will present the process of creation of the environment and the logic behind the decisions made.

Section 5.1.1 describes what is the goal of the agent, what it will train to do, the complexity of Diplomacy, and introduces the variant maps.

In Section 5.1.2, the Gym framework will be detailed and the motif on why it was chosen.

BANDANA is made in Java and OpenAI Gym is made in Python, so it was needed to create a system to send the messages between both language. In Section 5.1.3, the needed adaptation of messages from BANDANA's Java to Gym's Python will be stated.

The OpenAI architecture is designed to advance to the next state when the agent decides to, but BANDANA and MAS architectures are set to be the environment to decide when to advance to the next state. The gym-diplomacy changes the OpenAI Gym environment to let the environment decide when to move to the next state. Section 5.1.4 explains the code details of the environment execution and this alteration.

5.1.1 Diplomacy Environment

A turn in the Diplomacy game is made by 5 seasons: SPR (Spring Moves), SUM (Spring retreats), FAL (Fall Moves), AUT (Fall retreats), WIN (Adjustments).

- **SPR & FAL**: the agent has to send orders to move its units.

- **SUM & AUT**: the agent send commands to resolve the orders sent in the case that a unit lost a fight and has to retreat.
- **WIN**: the agent sends build orders to create new units in the case that it has captured a new Supply Center, or it sends disband orders in the case of losing a Supply Center.

In this environment, the agent will focus on the SPR and FAL phases of the turn since they are the ones with most impact on the game strategy, the other phases will use orders generated by a DumbBot. "Hold" orders will replace any empty order sent by the agent, therefore, there are no risks of occurring timeouts. "Hold" orders will also substitute received invalid orders so that the environment is not stuck while the agent is learning the borders because, in the early stages of training, the agent sends orders of Regions that are not adjacent.

Diplomacy has a branching factor of 450 [FS11] in the standard seven-player map. Consequently, in order to test the environment, more accessible variant maps provide faster feedback. Two variant maps were created, the "Small" variant uses a map with fewer Regions to fasten the training process and "Three" which is a three-player map to study the impact of increasing the number of players. The maps main specifications can be seen in table 5.1. The representation of the "Small" variant map can be seen in figure 5.2.

Map Name	Players	Provinces	Regions	Supply Centers	SC_{toWin}
Standard	7	75	121	34	18
Three	3	37	37	15	8
Small	2	19	19	9	5

Table 5.1: Diplomacy maps specifications

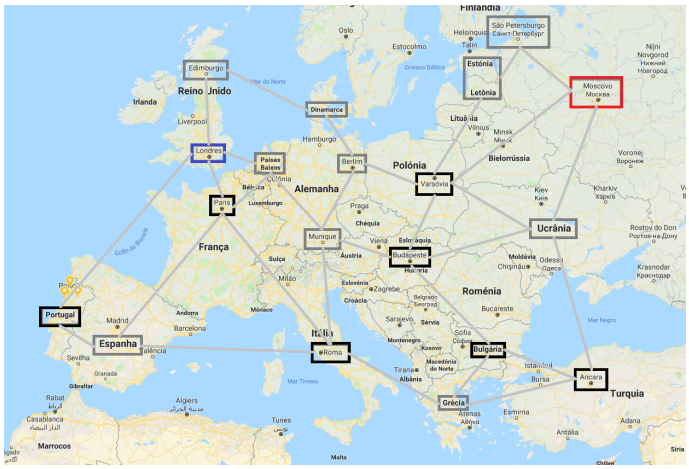

Figure 5.2: Representation of the "Small" map variant.

5.1.2 OpenAI Gym

OpenAI Gym is a Python toolkit to develop *reinforcement learning* agents that operate on user defined environments.

OpenAI Gym defines a standard architecture that a *reinforcement learning* agent and the environment should have. By using this defined interface, the agents are capable of interacting with different environments, and the environments can be used to compare the performance of different *reinforcement learning* approaches in the same conditions. Given that *reinforcement learning* algorithms are very general and can be applied to a multitude of situations, being able to generate a model in different scenarios with good results is very beneficial, as it proves the algorithm usefulness.

OpenAI maintains a repository, Baselines [DHK+17], containing examples of implementations of state-of-the-art DRL methods. These implementations can be used to validate the created environment. Applying these agents can lead to a better understanding of which algorithms perform better under the specific circumstances of Diplomacy and on other multi-agent cooperative scenarios.

The defined Gym interface is made by two methods that the agent will use to interact with the environment:

- `reset`: A function that resets the environment to a new initial *state* and returns its initial *observation*. It is used to initiate a new episode.
- `step`: A function that receives the *action* that the agent wishes to use to interact with the environment as the argument and returns *observation*, *reward*, *done*, and *info*.
    1. *observation*: The state of the environment.
    2. *reward*: The value of the state-action pair.
    3. *done*: The status of the episode.
    4. *info*: An optional information value.

In OpenAI Gym, an environment must define the "action space" and the "observation space" fields in order to abstract the environment to generic code.

- `action space`: The space of possible actions that will be used to generate the actions.
- `observation space`: The space that defines the dimensions of the environment's state.

Following the definitions, created for Diplomacy, of state in 4.2.1 and action space in 4.4.3, from the existing spaces available in OpenAI Gym, the class most appropriate to represent both of them is the "MultiDiscrete" class.

The MultiDiscrete space consists of a series of `Discrete` spaces with different number of cases in each. It is parametrized by passing an array of positive integers specifying the number of possible cases for each of its child spaces. A Discrete space with dimension n is a set of integers $\{0, 1, \ldots, n-1\}$.

Depending on the map that is being used, the dimension of both MultiDiscrete spaces will change accordingly to the map's number of Regions, but the Discrete spaces inside them will remain the same.

- Each State row: $[N_{Players}, 2, N_{Players}]$ (3 Discrete spaces)
- Each Action Space row: $[1 + (N_{Actions} - 1) * N_{Regions}]$ (1 Discrete space)

5.1.3 Communication between Python and Java

In this environment, the Python agent will have to send a list of actions to the Java player. `OpenAIAdapter` is the class created to make the connection between the Python classes and the Java classes. Google's protocol buffers[1] and gRPC[2] were used to do the communication between the different languages.

Protocol buffers is an open-source mechanism to serialize structured data, similar to XML, that is independent of language and platform. The data used is smaller than a typical XML which will make the communication faster and not interfere in the training process. In this case, protocol buffers will be used to generate methods, in both Python and Java, that can use the interpretation of the state, as seen in Listing 5.1, and action space data, present in the Listing 5.2. In order to make the data of the communication smaller, a representation using integer was used, so in BANDANA is necessary to convert the game state to that representation.

```
message RegionData {
  int32 id = 1;
  int32 owner = 2;
  int32 sc = 3;
  int32 unit = 4;
}
```

Listing 5.1: Protocol buffer representation of the Regions data. The values represent the order that parameter takes on the message.

```
message OrderData {
  int32 start = 1;
  int32 action = 2;
  int32 destination = 3;
}
```

Listing 5.2: Protocol buffer representation of the Orders data. The values represent the order that parameter takes on the message.

gRPC is remote procedure call (RPC) that facilitates the creation of distributed applications and services. This service uses by default protocol buffers, which makes it a natural selection to the chosen serialization.

[1] https://developers.google.com/protocol-buffers/
[2] https://grpc.io/

The Gym environment is configured as the RPC server and the BANDANA player as the RPC client. Using this RPC service, the Java player will not need to know that it is interacting with a Python agent as everything is hidden in the RPC, which will facilitate the development of future agents.

The implemented remote methods are present in Listing 5.3, and that method sends from Java the state of the board which the Python agent will answer with its intended action.

```
service DiplomacyGymService {
  rpc GetStrategyAction (BandanaRequest) returns (DiplomacyGymOrdersResponse) {}
}
```

Listing 5.3: gRPC implemented procedures

5.1.4 gym-diplomacy implementation

The OpenAI Gym interface has been built having in mind environments where there is *only one* controllable agent that can choose *when* to act. The agent should be able to call the `reset` and `step` function at any time. However, in a board game such as Diplomacy, the players must wait for their turn to play so the environment would not react immediately to the agent's step call. To circumvent this issue, the flag `waiting_action` indicates when the `step` function should proceed and when it should be blocked. This way, the agent can always call the step function, whenever it wants, but the function may make it wait for the result.

When the agent calls the `reset` function, shown in Algorithm 1, it expects the initial state of the Diplomacy board in return. To obtain it, the BANDANA process and the gRPC server starts. Initially, the observation state is set to a null value, then the flag is set to block to wait for the received state.

After the game and the players processes start, the first round of the game begins. Every Spring and Fall, DeepDip will send a request for action, with the current game observation attached. The `handle_request` function, that can be seen in Algorithm 3, takes the request, extracting the state information from it, and setting the relevant variables. It then sets the `wait_action` flag as ready and hangs, waiting for the agent's action. The `reset` function is now allowed to continue and returns the initial observation to the agent.

With the observation, the Gym agent calls the `step` function, that can be seen in Algorithm 2), providing an action as the argument. This function sets the `action` global variable and the `wait_action` flag, meanwhile the handler sends the action to BANDANA through the `handle_request` function.

The `handle_request` function will return again a new observation of the game state as the result of the action that the agent took. The agent will call again the `step` function and everything is repeated, until the game ends.

When the game ends, BANDANA saves the result of the game and the logs of the agents, and sends the `done` variable as true to indicate the agent that the game has ended and that it can

finish its current episode. Immediately after, BANDANA starts a new game. The agent receives the `done`, saves its reward score for the episode, and calls the `reset` function to start the new episode. This process will continue until the desired step is reached.

Algorithm 1: `reset` implementation

Data: bandana_subprocess: the process corresponding to the BANDANA game manager;
server: the gRPC server;
wait_action: a Boolean that determines whether the BANDANA player is waiting to be given an action or not;
action: the global variable holding the action to take in the environment;
observation: the current observation of the game state;
Result: Starts BANDANA and the gRPC server. When BANDANA is ready, returns the first state of the game;
observation: the observation corresponding to the initial game state;

```
action ← None;
observation ← None;
wait_action ← False;
if bandana_subrocess is None then
    bandana_subrocess ← init_bandana();
end
if server is None then
    server ← init_grpc_server();
end
while observation is None do observation is set by the handle_request function of the
  gRPC server.
    pass;
end
return observation
```

5.2 Experiments

In order to test if the environment is viable to study RL algorithms, simplified versions of the game were created with fewer powers, provinces, and units. 3 scenarios were created, the "Small" map, the "Three" map, and the "Standard" map, which specifications can be seen in table 5.1. These smaller maps are meant to reduce the observation space and the action space which will facilitate and accelerate the learning process.

To create the maps, configuration files were made to set up the maps. Two files were created: `small.cfg` for the "Small" variant, and `three.cfg` for the three-players variant.

In order to use these new maps, some alterations to BANDANA and parlance were made.

In BANDANA, a simple change was made to receive the name of the map as a parameter on the function `ParlanceRunner.runParlanceServer()`.

Algorithm 2: `step` implementation

Data: wait_action: a Boolean that determines whether the BANDANA player is waiting to be given an action or not;
new_action: the global variable holding the action to take in the environment;
Result: Interacts with environment sending the action and retrieving the new state of the game;
observation: the new observation of the game state;
reward: the float value of the reward of the action;
done: informs if the game has ended or not;
info: additional and optional information;

```
stored_action ← new_action;
while wait_action is not true do
    pass;
end
return observation, reward, done;
```

Algorithm 3: `handle_request` implementation

Data: new_action: the global variable holding the action to take in the environment;
request: the request of the BANDANA player;
Result: When the Diplomacy player sends a request to get an action, the handler sets the wait_action flag, and returns the new_action;
wait_action: a Boolean that determines whether the BANDANA player is waiting to be given an action or not;
observation: the new observation of the game state;
reward: the float value of the reward of the action;
done: informs if the game has ended or not;
info: additional and optional information;
clean_action: returns to the Java agent only the valid Orders;

```
observation, reward, done, info ← parse_data(request);
wait_action ← True;
if done is True then
    return ;
end
while wait_action do
    pass;
end
clean_action ← remove_invalid_orders(new_action);
return clean_action
```

In parlance there were more alterations. The code is not prepared to set new variants of the map dynamically. It was necessary to include in the file `xtended.py` the initialization of the new maps, and in `entrypoints.txt` the new maps were added to the variants block.

All of the tests were executed using the same reward function as described in 4.3.1.

The PPO algorithm, as introduced in section 2.3.2.7, was used to train in all the variants of the map. The PPO algorithm was provided by Stable-Baselines repository [HRE+18], which is an implementation of OpenAI's Baselines with some modifications that were used to facilitate the process of analysing, saving, and loading the agent. A graph with the structure of the model can be seen in figure 5.3. The default parameters of the algorithm were used.

- Discount factor: 0.99;
- Number of steps per update: 128;
- Entropy coefficient: 0.01;
- Value function coefficient: 0.5;
- Clipping parameter: 0.2;

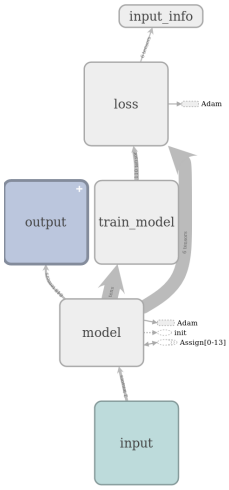

Figure 5.3: Representation of the graph from the PPO model. The image was generated using Tensorboard.

5.2.1 Small Map Experiment

In this map variant, named 'small', there are only 2 Players and 19 Regions, of which 9 are Supply Centers. DeepDip will train against 1 DumbBot. Both players start the game owning a single supply center. In this smaller board, a player must own 5 SC to win.

The reward function is calculated at the end of each episode, where an episode is equivalent to a game. If the game does not end in a draw, the agent will receive a reward equal to its number of Supply Centers plus a bonus or penalty depending on the end game result. If it wins the game,

the agent receives an extra positive reward of $+5$ (the total reward will be at least 10), while when losing it accumulates a penalty of -5 (the total reward will be within $[-5, -1]$). Figure 5.4 contains the result of an execution learning from scratch.

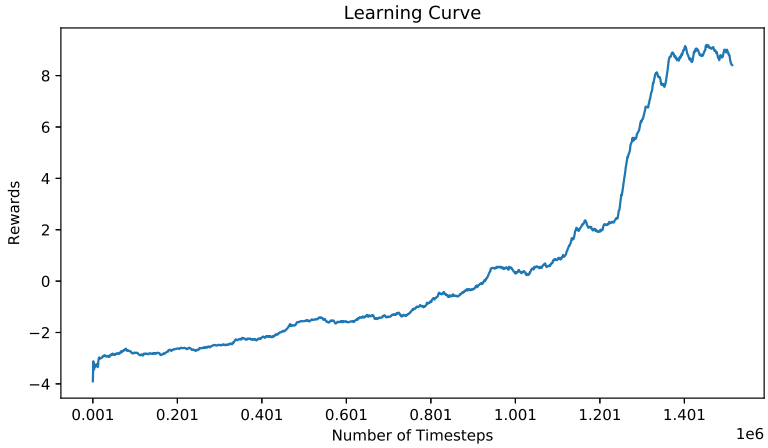

Figure 5.4: Rewards per episode of a PPO agent in the 'small' board. A positive reward indicates that the agent was not eliminated from the game. A reward is higher than 10 when the agent has won the game.

A run of 10^4 steps was used to make a final evaluation of the trained agent. It has won 745 out of 796 games, which translates to 93.6% of victories (combination of solo victories and draws where the agent has more Supply Centers than the opponent). The mean reward was of 9.21, corresponding to 732 solo victories.

5.2.2 Three Map Experiment

The 'three' map variant is made for 3 Players and has 37 Regions, of which 15 are SC. DeepDip will train against 2 DumbBots. Again, all players start the game owning one SC. As the there are more SC, now the players must capture 8 SC to win.

The reward function is the same as in the previous experiment. In this scenario, the bonus and penalty is adapted to the number of SC, so if it wins the game, the agent receives an extra positive reward of $+8$ (the total reward will be at least 16), while when losing it accumulates a penalty of -8 (the total reward will be within $[-8, -1]$). Figure 5.5 contains the result of an execution learning from scratch.

A run of 10^4 steps was used to make a final evaluation of the trained agent. The mean reward of the evaluation was of -3.1, which means that the agent was not capable of winning any game but was starting to understand how to not get eliminated. When the reward is -8, the agent is eliminated, so, because the mean reward is getting closer to 0, it means that agents is getting eliminated in less games. In 70 games, DumbBot 1 got an average rank of 1.629, DumbBot 2 got 1.957 and lastly DeepDip got 2.414.

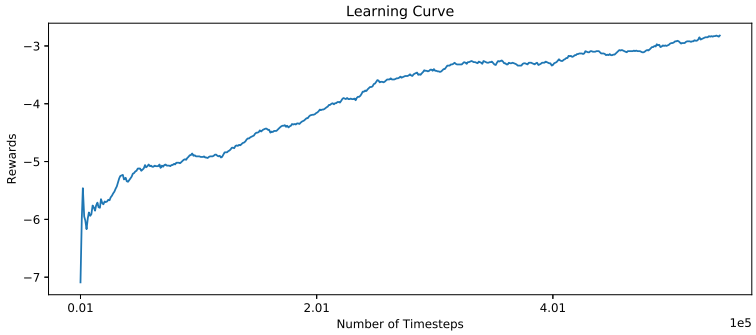

Figure 5.5: Rewards per episode of a PPO agent in the 'three' board. The agent did not achieve good results as it was not capable of getting wins in this variant, but the results were improving so with more training the agent maybe could achieve better results.

5.2.3 Standard Map Experiment

The 'standard' map variant is made for 7 Players and has 121 Regions, of which 34 are SC. DeepDip will train against 6 DumbBots. The players start the game owning 3 SC, except for the Russia player which starts with 4 SC. The players must now capture 18 SC to win.

The reward function is the same as in the previous experiment. In this scenario, the bonus and penalty is adapted to the number of SC, so if it wins the game, the agent receives an extra positive reward of $+18$ (the total reward will be at least 36), while when losing it accumulates a penalty of -18 (the total reward will be within $[-18, -1]$). Figure 5.6 contains the result of an execution learning from scratch.

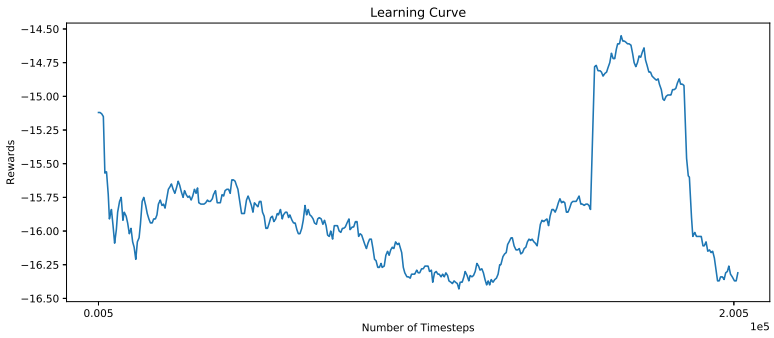

Figure 5.6: Rewards per episode of a PPO agent in the 'standard' board.

A run of 10^4 steps was used to make a final evaluation of the trained agent. The mean reward of the evaluation was of -16.93, which means that the agent was not capable of winning any game and was eliminated almost every time. In 168 games, DeepDip got an average rank of 5.494 with 0 victories and all the games that it did not finish as position 7 were because other players were eliminated.

5.3 Analysis

The results on the "small" map were better than the other two variants. As predicted in the curse of dimensionality [Bel66], the problem increases in difficulty for the agent as the maps get bigger and with more players.

In the "small" map, the results were quite good as the agent proves that it is capable of winning the game by understanding the rules of the game. Most of the actions made in this map were "Move-to Orders" because they are the orders that faster lead to success, which was predictable. The length of the games is short, as to not allow the opponent to get more units, the agent has an established strategy of doing quick captures. The opponent, DumbBot, does not have the most complex strategy, and that strategy was designed with the standard map in mind, so more trials with other bots would be interesting to analyze, but all of the other bots are designed for the standard map and incapable of playing a smaller variant.

In the "three" map, the results are improving but still unsatisfactory. The agent was only starting to understand how to avoid losing every SC it owned. More training was needed to understand if it had potential to get wins in this map. The increase in map size showed not to be the main factor in the poor results of the agent since it was capable of avoid being eliminated because its mean reward was increasing and getting closer to 0, so it is still losing but ends the game with more SC. The introduction of a third player was the principal factor of complexity as it introduces more entropy in the environment that makes the necessity in the agent to protect more its SC as it might have two attacks simultaneously in different Regions. The usage of "Hold Orders" would be a requirement to get good results in this map, but at best result achieved in training the agent was still not capable of consistently making them. The length of the games was high, in the "small" map there were 745 games, meanwhile, in "three" map, there were only 70 which is anticipated, because the agent has increased focus on defending its positions trying to not get eliminated than capturing more SC to win the game and the opponents created in the standard map don't have a proper strategy for this map. In long games, since there happen more rounds in each match, the training is slowed down.

In the "standard" map, the results are inconclusive due to the slow training. As there are 7 agents running in the same computer, the demand for computational power increases, which slows down the training. It is needed more training to understand if it had potential to get wins in this map. There are more games because the DumbBot strategy was made for this map, making the games short in length.

With these experiments, it is possible to conclude that DeepDip was able to understand the basic rules of Diplomacy but was still not able to achieve a human level of skill in the game. The results were consistent and independent of the Power that DeepDip was playing as.

The gym-diplomacy environment that was created provides an easy setup for developers to research Diplomacy using Python frameworks. It can also be used as an example of how to adapt an OpenAI Gym environment from agent-centered to environment-centered. The gRPC communication between Java and Python is also a feature that proved itself important when compared to

the initial implementation that used plain sockets. Using plain sockets there were a lot of messages that were lost, and the server crashed after hours of training, while in the gRPC implementation, there were no messages lost, and the server was able to train for an indefinitely period of time.

Chapter 6

Conclusions

To properly prepare this project, Chapter 2 revisited the concepts that make the foundation of DRL with particular attention to the current state of DRL research in Section 2.3 in the form of a review, and the well-known connection of AI and SG in Section 2.5.

Chapter 3 analyzes the existing environments prepared to develop DRL agents.

Chapter 4 introduces a theoretic model of how to apply DRL in strategic games was created and implemented to Diplomacy.

Then, the Diplomacy model was implemented using the OpenAI Gym architecture in Chapter 5, and the new environment was named gym-diplomacy. The connection between BANDANA and OpenAI Gym required the development of a communication channel between Java and Python, which was successfully managed by gRPC. This new environment includes the standard and smaller Diplomacy variants, but not all bots provided by BANDANA are dynamic to play in maps that are not the standard. DumbBot (Section 2.6.3) was chosen to be the default opponent on the environment because it can play and win in the smaller variants and it was the one that increased less the computational requirement in the standard map.

The environment is compatible with OpenAI's Baselines, which is a set of high-quality implementations of RL algorithms, and Stable Baselines that extends the original providing more customization. Using Stable Baselines' PPO algorithm, the process of creating DeepDip for gym-diplomacy was made more manageable. DeepDip achieved outstanding results in the two-player variant, promising results three-player variant, and inconclusive results in the standard seven-player variant.

With this work and its results, Diplomacy was proven to be appropriate to study DRL agents. The multi-agent factor of Diplomacy proved to be a challenge for the agent. With the results in the two-player variant, DRL is again validated as a valid approach to SG, further, with the background study and the results in the three-player variant, it is also an established approach to *strategic multi-agent games* . Every player has the same probabilities of winning, independently of its starting Power, as DeepDip did not register a significant difference in results when playing

a particular Power. Current state-of-the-art algorithms are capable of developing a strategy to win a game of Diplomacy since the agent was able to win. One of the main difficulties in the area is still the curse of dimensionality because it increases the hardware requirement as proven with the worse results on bigger maps that expand the state and action space.

The source code for this project was made available in https://github.com/BlueDi/DeepDip to provide a framework for future works.

6.1 Future Work

Reproducing the experiments in a system with better computational power would provide faster training to analyze the "three" and "standard" map.

A different approach to the model would also be interesting to analyze. Instead of placing the areas as the smallest unit, the player units could be the smallest unit. An idea would be to create a NN for each unit making it possible to create and destroy them as the number of units of the player changes.

Especially in the case of the smaller maps, the existing agents did not perform as good as expected. Introducing self-play would allow the agent to learn quicker and develop better strategies. Converting the trained model from Python into Java would allow the agent to play against itself introducing self-play, and to include DeepDip in BANDANA's example of agents. Adapting an existing agent to play independently of the size of the board would be an alternative to create a stronger opponent.

Introducing a hierarchical modular structure in the DRL algorithm would be interesting as it would provide additional strategies for the agent. It would help the agent to train the capabilities of defending via support orders.

Combining the strategic capacities with negotiation capacities would also be interesting.

The game engine that BANDANA uses is parlance. Parlance is written in Python 2 which is outdated and will not be maintained past 2020. Converting parlance to Python 3 would be helpful to create a system where the agent did not train using bandana because it would not be needed to convert the messages between Java and Python which would reduce the communication time.

BANDANA at the moment is hard-coded for the standard map. This makes statistic features, that BANDANA provides, not accessible to developers in the smaller variants. Adapting BANDANA to be dynamic as of the map being used would provide additional statistics for developing agents in smaller maps to better analyze their performance.

References

[ABC+16] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016. Cited on page 41.

[ACS18] Andrea Asperti, Daniele Cortesi, and Francesco Sovrano. Crawling in Rogue's dungeons with (partitioned) A3C. *arXiv:1804.08685 [cs, stat]*, April 2018. Cited on page 31.

[APM+17] Andrea Asperti, Carlo De Pieri, Mattia Maldini, Gianmaria Pedrini, and Francesco Sovrano. A Modular Deep-learning Environment for Rogue. *WSEAS Transactions on Systems and Control*, 12, 2017. Cited on page 31.

[BB01] Joseph P. Bigus and Jennifer Bigus. *Constructing Intelligent Agents Using Java: Professional Developer's Guide, 2nd Edition*. Wiley, New York, edição: 2 edition, 13 de março de 2001. Cited on pages xi and 21.

[BCP+16] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *arXiv:1606.01540 [cs]*, June 2016. Cited on pages 29 and 41.

[Bel66] Richard Bellman. Dynamic programming. *Science*, 153(3731):34–37, 1966. Cited on page 52.

[BLT+16] Charles Beattie, Joel Z. Leibo, Denis Teplyashin, Tom Ward, Marcus Wainwright, Heinrich Küttler, Andrew Lefrancq, Simon Green, Víctor Valdés, Amir Sadik, Julian Schrittwieser, Keith Anderson, Sarah York, Max Cant, Adam Cain, Adrian Bolton, Stephen Gaffney, Helen King, Demis Hassabis, Shane Legg, and Stig Petersen. DeepMind Lab. *arXiv:1612.03801 [cs]*, December 2016. Cited on page 29.

[BNVB13] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research*, 47:253–279, June 2013. Cited on page 29.

[CCLC19] Diogo Cruz, José Aleixo Cruz, and Henrique Lopes Cardoso. Reinforcement Learning in Multi-Agent games: OpenAI Gym Diplomacy environment. In Paulo de Moura Oliveira, Paulo Novais, and Luís Paulo Reis, editors, *Progress in Artificial Intelligence: 19th EPIA Conference on Artificial Intelligence*. Springer, September 2019. Cited on page 24.

[CHhH02] Murray Campbell, A.Joseph Hoane, and Feng hsiung Hsu. Deep blue. *Artificial Intelligence*, 134(1):57 – 83, 2002. Cited on pages 2 and 23.

[dBA+18] Dave de Jonge, Tim Baarslag, Reyhan Aydogan, Catholijn Jonker, Katsuhide Fujita, and Takayuki Ito. The Challenge of Negotiation in the Game of Diplomacy. In *The 6th International Conference on Agreement Technologies*, 2018. Cited on page 27.

[de 17] Sancho Fernandes de Mascarenhas. *AI Player for Board Game Diplomacy*. Master's Thesis, Instituto Superior Técnico, June 2017. Cited on page 28.

[DHK+17] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. *OpenAI Baselines*. GitHub, 2017. Cited on page 44.

[dS15] Dave de Jonge and Carles Sierra. NB$\hat{3}$: A multilateral negotiation algorithm for large, non-linear agreement spaces with limited time. *Autonomous Agents and Multi-Agent Systems*, 29(5):896–942, September 2015. Cited on page 28.

[dS17] Dave de Jonge and Carles Sierra. D-Brane: A diplomacy playing agent for automated negotiations research. *Applied Intelligence*, 47(1):158–177, July 2017. Cited on pages xi, 24, 27, and 28.

[FAP+17] Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, and Shane Legg. Noisy Networks for Exploration. *arXiv:1706.10295 [cs, stat]*, June 2017. Cited on page 15.

[FCR15] André Ferreira, Henrique Lopes Cardoso, and Luís Paulo Reis. Strategic Negotiation and Trust in Diplomacy - The DipBlue Approach. *Trans. Computational Collective Intelligence*, 20:179–200, 2015. Cited on page 28.

[FS09] Angela Fabregues and Carles Sierra. Diplomacy game: The test bed. *PerAda Magazine, towards persuasive adaptation*, pages 5–6, 2009. Cited on page 28.

[FS11] Angela Fabregues and Carles Sierra. DipGame: A challenging negotiation testbed. *Engineering Applications of Artificial Intelligence*, 24(7):1137–1146, October 2011. Cited on pages 2, 27, and 43.

[HMv+17] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining Improvements in Deep Reinforcement Learning. *arXiv:1710.02298 [cs]*, October 2017. Cited on pages xi, 16, and 17.

[HRE+18] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable baselines. https://github.com/hill-a/stable-baselines, 2018. Cited on page 49.

[HZAL18] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. *arXiv:1801.01290 [cs, stat]*, January 2018. Cited on page 21.

[ICA18] Geoffrey Irving, Paul Christiano, and Dario Amodei. AI safety via debate. *arXiv:1805.00899 [cs, stat]*, May 2018. Cited on page 29.

[JBV+18] Arthur Juliani, Vincent-Pierre Berges, Esh Vckay, Yuan Gao, Hunter Henry, Marwan Mattar, and Danny Lange. Unity: A General Platform for Intelligent Agents. *arXiv:1809.02627 [cs, stat]*, September 2018. Cited on page 32.

[Jul16] Arthur Juliani. Simple Reinforcement Learning with Tensorflow Part 8: Asynchronous Actor-Critic Agents (A3C), December 2016. Cited on pages xi and 19.

[KMO+18] Łukasz Kidziński, Sharada P Mohanty, Carmichael Ong, Jennifer Hicks, Sean Francis, Sergey Levine, Marcel Salathé, and Scott Delp. Learning to run challenge: Synthesizing physiologically accurate motion using deep reinforcement learning. In Sergio Escalera and Markus Weimer, editors, *NIPS 2017 Competition Book*. Springer, Springer, 2018. Cited on page 31.

[KNST16] Tejas D. Kulkarni, Karthik R. Narasimhan, Ardavan Saeedi, and Joshua B. Tenenbaum. Hierarchical Deep Reinforcement Learning: Integrating Temporal Abstraction and Intrinsic Motivation. *arXiv:1604.06057 [cs, stat]*, April 2016. Cited on pages xi, 15, and 16.

[Leo18] Mat Leonard. Intro to Deep Learning with PyTorch - Udacity. https://classroom.udacity.com/courses/ud188, 2018. Cited on pages xi, 9, and 10.

[LHP+15] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv:1509.02971 [cs, stat]*, September 2015. Cited on page 18.

[Li18] Yuxi Li. Deep Reinforcement Learning. *arXiv:1810.06339 [cs, stat]*, October 2018. Cited on page 6.

[LMK+17] Jan Leike, Miljan Martic, Victoria Krakovna, Pedro A. Ortega, Tom Everitt, Andrew Lefrancq, Laurent Orseau, and Shane Legg. AI Safety Gridworlds. *arXiv:1711.09883 [cs]*, November 2017. Cited on page 29.

[LWT+17] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments. *arXiv:1706.02275 [cs]*, June 2017. Cited on pages xi and 20.

[MAMK16] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource Management with Deep Reinforcement Learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, HotNets '16, pages 50–56, Atlanta, GA, USA, 2016. ACM. Cited on pages xi and 10.

[MBM+16] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. *arXiv:1602.01783 [cs]*, February 2016. Cited on page 18.

[McC] John McCullock. A Painless Q-Learning Tutorial. http://mnemstudio.org/path-finding-q-learning-tutorial.htm. Cited on pages xi and 7.

[MKS+15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg

Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015. Cited on pages xi, 2, 11, 12, and 33.

[MLC18] João Marinheiro and Henrique Lopes Cardoso. Towards General Cooperative Game Playing. In Ngoc Thanh Nguyen, Ryszard Kowalczyk, Jaap van den Herik, Ana Paula Rocha, and Joaquim Filipe, editors, *Transactions on Computational Collective Intelligence XXVIII*, Lecture Notes in Computer Science, pages 164–192. Springer International Publishing, 2018. Cited on page 28.

[Nor05] David Norman. DAIDE - DumBot Algorithm. http://www.daide.org.uk/s0003.html, October 2005. Cited on page 28.

[OBPVR16] Ian Osband, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. Deep Exploration via Bootstrapped DQN. *arXiv:1602.04621 [cs, stat]*, February 2016. Cited on pages xi, 14, and 15.

[Ope18] OpenAI. OpenAI Five. https://openai.com/five/, 2018. Cited on pages 23 and 29.

[PGC+17] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS-W*, 2017. Cited on page 41.

[RMSL09] João Ribeiro, Pedro Mariano, and Luís Seabra Lopes. DarkBlade: A Program That Plays Diplomacy. In Luís Seabra Lopes, Nuno Lau, Pedro Mariano, and Luís M. Rocha, editors, *Progress in Artificial Intelligence*, Lecture Notes in Computer Science, pages 485–496. Springer Berlin Heidelberg, 2009. Cited on page 28.

[RN09] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009. Cited on page 21.

[S18] Suryansh S. Gradient Descent: All You Need to Know. https://hackernoon.com/gradient-descent-aynk-7cbe95a778da, March 2018. Cited on pages xi and 8.

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT press, 2018. Cited on pages xi, 6, and 18.

[SHM+16] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016. Cited on page 1.

[SHS+18] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144, December 2018. Cited on pages 2, 23, 29, and 33.

[Sil15] David Silver. UCL Course on RL. http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html, 2015. Cited on page 7.

[SLH+14] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic Policy Gradient Algorithms. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, pages I–387–I–395, Beijing, China, 2014. JMLR.org. Cited on page 17.

[SLM+15] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust Region Policy Optimization. *arXiv:1502.05477 [cs]*, February 2015. Cited on page 18.

[SoLR17] David Simões, Nuno Lau, and Luís Paulo Reis. Multi-agent Double Deep Q-Networks. In Eugénio Oliveira, João Gama, Zita Vale, and Henrique Lopes Cardoso, editors, *Progress in Artificial Intelligence*, Lecture Notes in Computer Science, pages 123–134. Springer International Publishing, 2017. Cited on page 2.

[SQAS15] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized Experience Replay. *arXiv:1511.05952 [cs]*, November 2015. Cited on page 12.

[SWD+17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *arXiv:1707.06347 [cs]*, July 2017. Cited on page 20.

[Tas16] Norman Tasfi. Pygame learning environment. https://github.com/ntasfi/PyGame-Learning-Environment, 2016. Cited on page 32.

[TDM+18] Yuval Tassa, Yotam Doron, Alistair Muldal, Tom Erez, Yazhe Li, Diego de Las Casas, David Budden, Abbas Abdolmaleki, Josh Merel, Andrew Lefrancq, Timothy Lillicrap, and Martin Riedmiller. DeepMind Control Suite. *arXiv:1801.00690 [cs]*, January 2018. Cited on page 30.

[Tes95] Gerald Tesauro. Temporal difference learning and td-gammon. *Commun. ACM*, 38(3):58–68, March 1995. Cited on page 23.

[Tur53] Alan Mathison Turing. Digital computers applied to games. *Faster than Thought*, 25, 1953. Cited on pages 2 and 23.

[VÍ7] Favio Vázquez. Deep Learning made easy with Deep Cognition. https://becominghuman.ai/deep-learning-made-easy-with-deep-cognition-403fbe445351, December 2017. Cited on pages xi and 9.

[VEB+17] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, John Quan, Stephen Gaffney, Stig Petersen, Karen Simonyan, Tom Schaul, Hado van Hasselt, David Silver, Timothy Lillicrap, Kevin Calderone, Paul Keet, Anthony Brunasso, David Lawrence, Anders Ekermo, Jacob Repp, and Rodney Tsing. StarCraft II: A New Challenge for Reinforcement Learning. *arXiv:1708.04782 [cs]*, August 2017. Cited on page 30.

[vGS15] Hado van Hasselt, Arthur Guez, and David Silver. Deep Reinforcement Learning with Double Q-learning. *arXiv:1509.06461 [cs]*, September 2015. Cited on page 12.

REFERENCES

[WD92] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, May 1992. Cited on page 6.

[Wen18] Lilian Weng. Policy Gradient Algorithms, April 2018. Cited on pages xi and 19.

[WKT+16] Jane X. Wang, Zeb Kurth-Nelson, Dhruva Tirumala, Hubert Soyer, Joel Z. Leibo, Remi Munos, Charles Blundell, Dharshan Kumaran, and Matt Botvinick. Learning to reinforcement learn. *arXiv:1611.05763 [cs, stat]*, November 2016. Cited on page 19.

[WSH+15] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling Network Architectures for Deep Reinforcement Learning. *arXiv:1511.06581 [cs]*, November 2015. Cited on pages xi, 13, and 14.

Appendix A

EPIA 2019 Paper

Reinforcement Learning in Multi-Agent Games: OpenAI Gym Diplomacy Environment

Diogo Cruz[1], José Aleixo Cruz[1], and Henrique Lopes Cardoso[1,2]

[1] Faculdade de Engenharia, Universidade do Porto, Portugal
{up201105483,up201403526,hlc}@fe.up.pt
[2] Laboratório de Inteligência Artificial e Ciências dos Computadores (LIACC), Porto, Portugal

Abstract. Reinforcement learning has been successfully applied to adversarial games, exhibiting its potential. However, most real-life scenarios also involve cooperation, in addition to competition. Using reinforcement learning in multi-agent cooperative games is, however, still mostly unexplored. In this paper, a reinforcement learning environment for the Diplomacy board game is presented, using the standard interface adopted by OpenAI Gym environments. Our main purpose is to enable straightforward comparison and reuse of existing reinforcement learning implementations when applied to cooperative games. As a proof-of-concept, we show preliminary results of reinforcement learning agents exploiting this environment.

Keywords: reinforcement learning · multi-agent games · Diplomacy · OpenAI Gym

1 Introduction

Artificial intelligence has grown to become one of the most notable fields of computer science during the past decade. The increase in computational power that current processors provide allows computers to process vast amounts of information and perform complex calculations quickly and cheaply, which in turn has renovated the interest of the scientific community in machine learning (ML). ML software can produce knowledge from data. Reinforcement learning (RL) [16] is an ML paradigm that studies algorithms that give a software agent the capability of learning and evolving by trial and error. The knowledge an RL agent acquires comes from interactions with the environment, from understanding what actions lead to what outcomes. While computers are getting better at overcoming obstacles using reinforcement learning, they still have great difficulty with acting in and adjusting to real-life scenarios.

Games have always been an essential test-bed for AI research. Researchers have focused mostly on adversarial games between two individual opponents, such as Chess [5]. Reinforcement learning, in particular, has been successfully applied in this type of games, with increasing efficiency over the past years. One of the first games for which RL techniques have been applied to develop software

playing agents was backgammon [17], while recently more complex games like Go [15], Dota 2 [12] and a variety of Atari games [11] have been the main center of attention.

Games where negotiation and cooperation between players are encouraged but also allow changes in the relationships over time, have not been given the same amount of attention. Generally, these kinds of multi-agent games have a higher level of complexity: agents need not only to be concerned with winning the game, but they also need to coordinate their strategies with allies or opponents, either by competing or by cooperating, while considering the possibility of an opponent not fulfilling its part of the deal.

Experimenting with this type of games is important because they mimic the social interactions that occur in a society. Negotiating, reaching an agreement and deciding whether or not to break that agreement is all part of the daily life. Achieving cooperative solutions allows us to derive answers for real-life problems, for example, in the area of social science.

With this paper, we provide a tool that facilitates future research by making it easier and faster to build agents for this type of games. More specifically, we introduce an open-source OpenAI Gym environment which allows agents to play a board game called Diplomacy and evaluate the performance of state-of-the-art RL algorithms in that environment.

The rest of the paper is structured as follows. Section 2 introduces background information regarding Diplomacy, the BANDANA program (a game engine for Diplomacy) and the OpenAI Gym framework. Section 3 describes how the environment was developed and implemented. Section 4 contains experimental data from trials using the proposed environment. Section 5 contains the main conclusions of this work and considerations about future improvements.

2 Background

2.1 Diplomacy

Diplomacy [3] is a complex board game. This competitive game can be played with up to 7 players, each having the objective of capturing 18 *Supply Centers* that are placed over 75 possible *Provinces*, by moving the player's owned units across the board. Diplomacy is a game that involves adversarial as well as cooperative decisions. Players can communicate with each other to create deals. A deal can be an agreement or an alliance that the player uses in order to defend itself or attack a stronger opponent. Yet, the deals agents make are not binding and players may betray alliances. The social aspect of Diplomacy makes it a perfect test-bed for cooperation strategies in adversarial environments. Because the search-tree of Diplomacy is very large, the time and storage requirements of tabular methods are prohibitive. As such, approximate RL methods must be employed. Together with its social component, this makes Diplomacy a fit domain to explore using reinforcement learning techniques.

Several bots have been developed for Diplomacy. Up until recently, most approaches limited themselves to the no-press variant of the game (i.e., without

negotiation). For a fairly recent list of works on both no-press and press variants, see Ferreira et al. [7]. De Jonge and Sierra [10] developed a bot called D-Brane, which encompasses both tactical and negotiation modules. D-Brane analyzes which agreements would result in a better tactical battle plan using Branch and Bound and is prepared to support an opponent, in the hopes of having the favor returned later in the game. D-Brane, however, was implemented in a variant of Diplomacy with binding agreements, explained in Section 2.2.

2.2 BANDANA

BANDANA [10] is a Java framework developed to facilitate the implementation of Diplomacy playing agents. It extends the DipGame [6] framework, providing an improved negotiation server that allows players to make agreements with each other. The Diplomacy league of the Automated Negotiating Agents Competition [9] asks for participants to conceive their submissions using the BANDANA framework.

Two types of Diplomacy players can be created using BANDANA – one can build a player that only makes tactical decisions or a player that also negotiates with its opponents. Tactical choices concern the orders to be given to each unit controlled by the player. Negotiations involve making agreements with other players about future tactical decisions. In the original Diplomacy game, these negotiations are non-binding, meaning that a player may not respect a deal it has reached. However, in BANDANA deals are binding: a player may not disobey an agreement it has established during the game. The removal of the trust issue that non-binding agreements bear simplifies the action space of mediation.

Tactics and negotiations in a BANDANA player are handled by two different modules. They may communicate with each other, but that is not mandatory. A complete BANDANA player consists of these two modules, that should obey to a defined interface.

To play a game of Diplomacy, BANDANA has a dedicated Java class which launches a game server and initializes each player. The game server is responsible for communicating the state of the game to the players and for receiving their respective actions. In the case of negotiation, BANDANA uses a separate server with a predefined message protocol that allows mediation. Players do not communicate directly with each other. The game continues until someone wins, or a draw is proposed and accepted by all surviving players.

Despite the fact that BANDANA facilitates the creation of a Diplomacy player, it is a Java-based platform, which makes it hard to connect with the most popular machine learning tools, often written in Python, such as Tensorflow [1] and PyTorch [13].

2.3 OpenAI Gym

OpenAI Gym [2] is a Python toolkit for executing reinforcement learning agents that operate on given environments. The great advantage that Gym carries is that it defines an interface to which all the agents and environments must obey.

Therefore, the implementation of an agent is independent of the environment and vice-versa. An agent does not need to be drastically changed in order to act on different environments, as the uniform interface will make sure the structure of the information the agent receives is almost the same for each environment. This consistency promotes performance comparison of one agent in different conditions, and of different agents in the same conditions. Two of the methods defined by the Gym interface are:

- `reset`: A function that resets the environment to a new initial *state* and returns its initial *observation*. It is used to initiate a new episode after the previous is done.
- `step`: A function that receives an *action* as an argument and returns the consequent *observation* (the state of the environment) and *reward* (the value of the state-action pair), whether the episode has ended (*done*) and additional information that the environment can provide (*info*).

Each environment must also define the following fields:

- `action space`: The object that sets the space used to generate an action.
- `observation space`: The object that sets the space used to generate the state of the environment.
- `reward range`: A tuple used to set the minimum and maximum possible rewards for a step.

This specification represents an abstraction that encompasses most reinforcement learning problems. Given that RL algorithms are very general and can be applied to a multitude of situations, being able to generate a model in different scenarios with good results is very beneficial, as it proves the algorithm usefulness. Also, as OpenAI Gym is built on Python, it is easier to connect Tensorflow and PyTorch with Gym agents and make use of the RL techniques that those frameworks provide. With this in mind, creating a Diplomacy environment for Gym would make it easier to implement RL agents that could play this game, and analyze their behavior. By taking the BANDANA framework and adapting it to the OpenAI Gym specification, a standard Diplomacy environment is created and can be explored by already developed agents, particularly RL agents. For instance, OpenAI maintains a repository containing the implementation of several RL methods [4] which are compatible with Gym environments. Employing these can lead to a better understanding of which methods perform better under the specific circumstances of Diplomacy and on other multi-agent cooperative scenarios. Also, if the model used to abstract Diplomacy is successful, it can be recycled to create environments for similar problems.

3 An OpenAI Gym Environment for Diplomacy

In this section we describe the proposed OpenAI Gym environment that enables Diplomacy agents to learn how to play the game. The main objective of the

environment is to take advantage of the features that both OpenAI Gym and BANDANA offer. We also intend to allow different configurations of a Diplomacy board to be used in the environment, besides the standard one. We try to achieve this by making a bridge between both frameworks, permitting intercommunication. The OpenAI Gym environment created will be referred to as `gym-diplomacy` throughout the paper.

Because BANDANA offers the choice of creating a strategic or a negotiation agent, we built an environment for each case. The created environments are similar but with different scopes of action spaces and reward functions. The strategic environment allows the use of custom maps, however the negotiation environment does not.

The architecture of `gym-diplomacy`[3] is detailed in Section 3.1. The definition of the observation space and its set up will be described in Section 3.2. The *action space* for both strategic and negotiation scenarios are described in Sections 3.3 and 3.4, respectively. The conversion of observation and action objects to a special OpenAI Gym class called `Spaces` is detailed in Section 3.5. The *reward function* that defines the reward that the agent will receive is described in Section 3.6.

3.1 `gym-diplomacy` architecture

The design proposed is represented in Figure 1. It consists of abstracting the Diplomacy game information provided by BANDANA to match the OpenAI Gym environment specification. We implement the methods required for a Gym environment, `reset` and `step`.

The BANDANA's features are inside the Gym environment. However, as a BANDANA player is written in Java and a Gym environment in Python, to exchange information we need to connect both using inter-process communication. For that, the server-client model was adopted using sockets as endpoints and Google's Protocol Buffers for data serialization.

When `reset` is called, the environment should return to its initial state, which means that it creates a new game. To do so, we make use of the BANDANA's `TournamentRunner` class to manage both the players and the game server. In the first reset call, the players and the game server are initialized, but in after calls the game server starts a new game without restarting the process. We then connect to our custom BANDANA player, retrieving the game's initial state iS. We created a Java class with the role of an adapter, which we attach to our BANDANA player, to convert the representation of the game state from the BANDANA format to OpenAI `Spaces` format so that the agent can interpret it, as explained in Section 3.5.

The OpenAI agent will analyze the received state and decide what its action A will be. When A is ready, the agent calls the `step` function, providing the action A it wants to execute as an argument. This action is also a `Spaces` object, so we need to convert it to a valid BANDANA action A'. We then pass

[3] Available at https://github.com/jazzchipc/gym-diplomacy.

the resulting action A' through our environment to the connected BANDANA player.

The BANDANA player executes A', which generates a new game state nS. The reward R of the action A' is calculated by the adapter, using BANDANA functions. A binary value D, which informs if the current game has ended, is also determined. Then, nS is converted to a `Space` object nS' and the environment sends nS', R, and D back to the OpenAI agent, which makes use of the information in its learning module. An optional parameter I, corresponding to the optional debug information, may be passed to the agent.

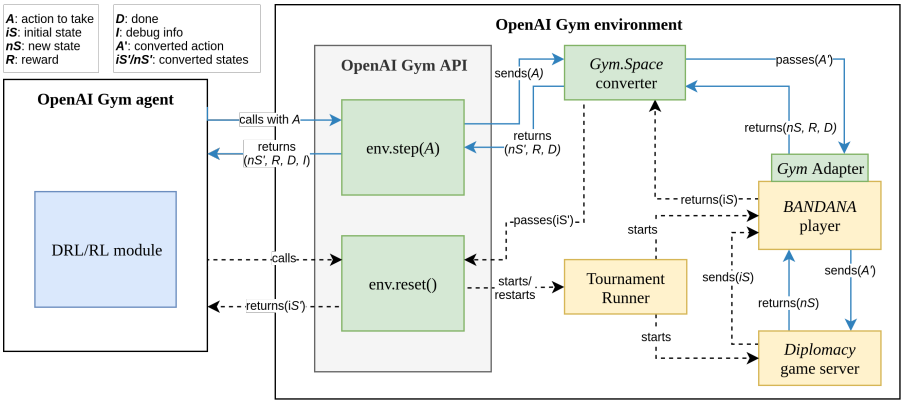

Fig. 1. Conceptual model of the Open AI Gym Diplomacy environment and agent. The solid and dashed arrows represent the interactions between the components when the agent calls the `step` and `reset` functions, respectively.

3.2 `gym-diplomacy` observation space

An observation of the Diplomacy game state should contain the most relevant information available to the player. In this case, the board gives that information. The information about all the *Provinces* is one possible representation of the current game state. Each *Province* may only be owned by one player at a time, it may have a structure called *Supply Center* that players must capture to win the game, and it can only have at most one *Unit* placed in it. Therefore, for a standard board of Diplomacy, a list with the information of the 75 *Provinces* can be used to represent the board. Each element of this list is a tuple containing the *Province* owner, whether it has a *Supply Center*, and the owner of the unit if it has one.

3.3 `gym-diplomacy` strategy action space

From the strategic point of view, for each turn, a player needs to give an order to each unit it has on the board. The number of units a player has corresponds to the number of *Supply Centers* it controls during a certain point of the game. There are 34 *Supply Centers* in a standard Diplomacy board. However, the maximum number of units a player can have at any given time is 17, because once a player holds 18 or more supply centers, it wins the game.

An order to an unit can be one of three possible actions: `hold`, `move to`, or `support`. The `hold` order directs the unit to defend its current position, while the `move to` order makes the unit attack the destination province; the `support` order tells the unit to support another order from the current turn.

For any player, Equation 1 gives an upbound on the possible number of orders for each unit n_{orders}, where P is the number of *Provinces* in the board. If we consider only adjacent *Provinces*, the number of possible actions would be more precise, but this information is not part of the state representation. The BANDANA framework will examine invalid orders, such as moving a unit to a non-adjacent province, and will replace them with `hold` orders.

$$n_{orders} = 1 + 2P \tag{1}$$

3.4 `gym-diplomacy` negotiation action space

From the negotiation point of view, in each turn a player needs to evaluate the current state of the board and decide if it is going to propose an agreement to its opponents. In the original version of Diplomacy, players talk freely, either privately or publicly. In BANDANA, however, to facilitate mediation between agents, there is an established negotiation protocol. According to it, a *Deal* is composed of two parts: a set of *Order Commitments* and a set of *Demilitarized Zones*. Any of these sets can be empty. An order commitment represents a promise that a power will submit a certain order o during a certain phase σ and a year y, represented by the tuple $oc = (y, \sigma, o)$. A demilitarized zone represents a promise that none of the specified Powers in the set A will invade or stay inside any of the specified provinces in set B during a given phase and year, represented by the tuple $dmz = (y, \sigma, A, B)$. Because a deal may contain any number of order commitments and demilitarized zones, and the year parameter can go up to infinity, the action space of negotiation is infinite as well. However, creating agreements several years in advance may not be advantageous, as the state of the board will certainly change with time. Therefore, a limit (y_{max}) can be considered for the number of years that should be planned ahead. Given the number of phases H, the number of units our player owns u_{own}, the number of units an opponent controls u_{op}, and the number of players L, the maximum number of deals becomes the value described in Equation 4, where n_{oc} is the number of possible oc and n_{dmz} is the number of possible dmz.

$$n_{oc} = y_{max} * H * (u_{own} + u_{op}) * n_{orders} \tag{2}$$

$$n_{dmz} = y_{max} * H * L * 2^P \tag{3}$$

$$n_{deals} = 2^{(n_{oc}+n_{dmz})} \tag{4}$$

Because for each deal we may or may not select a possible oc and dmz, the number of possible arrangements, and therefore the negotiation action space grows exponentially with base 2 for each oc and dmz available. Equations 2 and 3 express the upper bound for the value of n_{oc} and n_{dmz}, respectively, where P is the number of provinces in the board. While we can shrink the action space by only allowing actions which are valid for a given state, the search tree is still extremely immense.

3.5 OpenAI Gym Spaces

In OpenAI Gym, the action and observation spaces are objects that belong to a subclass of the `Space` class. The one we found most appropriate to represent the Diplomacy action and observation space is the `MultiDiscrete` class. In a `MultiDiscrete` space, the range of elements is defined by one or more `Discrete` spaces that may have different dimensions. A simple `Discrete` space with dimension n is a set of integers $\{0, 1, \ldots, n-1\}$. To encode the observation space, we characterize each province i with a tuple of integers (o_i, sc_i, u_i), where o represents the player that owns province (0 if none), sc is 0 if the province does not have a supply center or 1 otherwise, and u represents the owner of the unit currently standing in the province (0 if none). We use a `MultiDiscrete` space with $3n_p$ `Discrete` spaces, where n_p is the number of provinces. An observation for 75 provinces then becomes:

$$\text{observation: } [(o_1, sc_1, u_1), (o_2, sc_2, u_2), ..., (o_{75}, sc_{75}, u_{75})]$$

For tactical actions, the translation to a `MultiDiscrete` space is done by associating an integer to each type of order and to each province. Let sp denote the order's starting province, o the type of order and dp the destination province. Then a tactic action is described by:

$$\text{tactic action: } (sp, o, dp)$$

When the action type is `hold`, the value of dp is disregarded.

Given the immense complexity of the negotiation action space, we reduced the scope of action of `gym-diplomacy`. Instead of deciding over the whole action space, we limit the possible actions to one oc per deal that consists of two `move to` orders: one for a player's own unit and the other for an opponent's unit. We currently represent the negotiation action space with a `MultiDiscrete` space with five `Discrete` spaces. Let sp_{own} and dp_{own} represent the starting and destination provinces, respectively, of the move order for the agent's own unit. Let op be the opponent we are proposing the deal to. Let sp_{op} and dp_{op} be the starting and destination provinces of the opponent's units. Then a negotiation action in our limited scope is given by:

negotiation action: $(sp_{own}, dp_{own}, op, sp_{op}, dp_{op})$

3.6 `gym-diplomacy` reward function

The objective of the agent is to win the game and, to achieve this, it is required to conquer a certain number of supply centers, that depends on the board configuration. A straightforward approach to defining a reward function is to give a positive reward for a win, a neutral reward for a draw, and a negative reward for a loss. While this approach is appropriate for a small board layout, for a standard board, this results in a sparse reward space, as the agent is only able to learn after the end of an episode. To foster the learning process, we also study a reward function that considers the supply centers that the agent conquers at each turn. Therefore, in the negotiation environment, the agent learns with each action, instead of each episode, while leading to the same global objective. The reward function $R_a(s, s')$ is described in Equation 5, where r is a constant defining the reward for conquering one supply center and $SC(a)$ is the number of supply centers controlled in state a. It represents the reward of transitioning from state s to state s' after taking action a.

$$R_a(s, s') = r * (SC(s') - SC(s)) \tag{5}$$

4 Experimental Evaluation

Diplomacy presents an environment that is interesting to be used as a testbed for RL algorithms in a multi-agent perspective in two different approaches: strategic thinking and negotiation skills. In this section, we provide evidence that the strategic thinking needed for this game is still challenging for state-of-the-art RL algorithms. In the strategy experiment, we used an already implemented version of the Proximal Policy Optimization (PPO) [14] algorithm, from the stable-baselines repository [8]. In the negotiation experiment, we used an already implemented version of the Actor-Critic using Kronecker-Factored Trust Region (ACKTR) [18] algorithm, from the OpenAI baselines repository [4].

4.1 Strategic environment experiments

In order to test if the environment is viable to study RL algorithms, a simplified version of the game was created with fewer powers, provinces, and units. This is meant to reduce the observation space and the action space. This reduction will facilitate and accelerate the learning process which allows experimenting with different algorithms and developing a proper reward function.

In this version, named 'small', there are only 2 players and 19 provinces, of which 9 are supply centers. Both players start the game owning a single supply center. In this smaller board, a player must own 5 supply centers to win.

The PPO algorithm was used to train the agent. Figure 2 contains the result of an execution learning from scratch. The reward function is calculated at the

end of each game. If the game does not end in a draw, the agent will receive a reward equal to its number of Supply Centers plus a bonus or penalty depending on the end game result. If it wins the game, the agent receives an extra positive reward of $+5$ (the total reward will be at least 10), while when losing it accumulates a penalty of -5 (the total reward will be within $[-5, -1]$).

A run of 10^4 steps was used to make a final evaluation of the trained agent. It has won 745 out of 796 games, which translates to 93.6% of victories (combination of solo victories and draws where the agent has more Supply Centers than the opponent). The mean reward was of 9.21, corresponding to 732 solo victories.

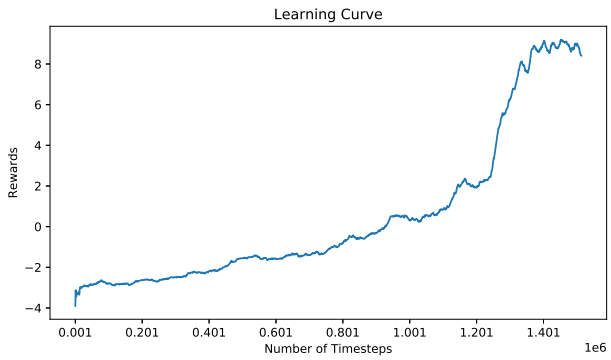

Fig. 2. Rewards per episode of a PPO agent in the 'small' board. A positive reward indicates that the agent was not eliminated from the game. A reward is higher than 10 when the agent has won the game.

4.2 Negotiation environment experiments

For negotiation scenarios, BANDANA does not allow a smaller map to be used. Therefore, we have used the standard 75 regions Diplomacy map for the negotiation experiments with all the 7 players. Because of the size of the action space, as mentioned in Section 3.4, we have started with a simple range of decision: the agent may only propose one deal per turn, to a single opponent, with only one order commitment. The order commitment is for the immediately following phase of the game and contains just two `move` orders.

Since negotiation does not directly affect the number of conquered supply centers, using the reward function in Equation 5 could lead to inconsistent learning. For that reason, we use a different reward function to train the agent for negotiation. The agent receives a positive reward for each valid deal and a negative reward for each invalid deal it proposes. A deal is invalid if the player proposes to itself or if the orders inside the deal do not match the current state

of the game. While this reward function does not directly lead to victory, it helps the agent to become better at negotiating. Because there is a time limit during the negotiation phase, it is important not to waste time by proposing invalid deals.

Figure 3 contains the average results of three different executions, all learning from scratch. Because each game may have a different number of turns, instead of showing the episode reward over the number of steps we show the average reward over the number of episodes.

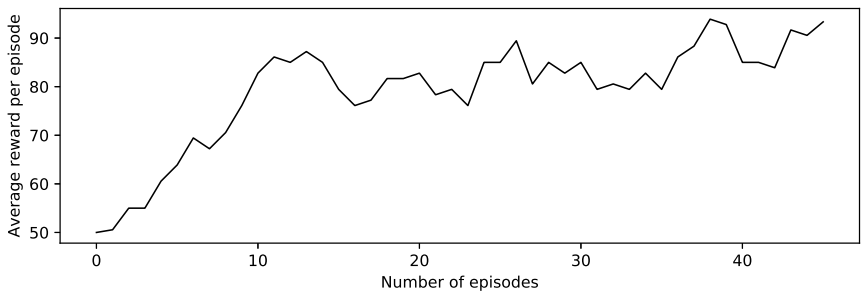

Fig. 3. Average rewards per episode (game) of an agent learning from scratch with ACKTR in the negotiation environment (3 executions over 46 episodes). The values have been smoothed using a window of size 3. Each game has a variable number of steps. A valid deal gets a positive reward $+5$, each invalid deal gets -5.

Because negotiation only takes place every two phases and is a rather long stage, running negotiation steps takes quite a bit of time, which limits the amount of training a player can have. However, the learning progress is evident, as the agent learns to propose more valid actions.

5 Conclusions

By combining the standardization of OpenAI Gym with the complexity of BANDANA, we have succeeded in facilitating the implementation of reinforcement learning agents for the Diplomacy game, both in the strategic and in the negotiation scenarios. We were able to create agents and to use already implemented algorithms, with little code adaptation. This achievement enables us to continue testing reinforcement learning techniques to improve Diplomacy players performance.

Some future enhancements include improving the representation of the action and observation space, as these are determinant in the performance of the techniques used. Diplomacy's environment execution is computationally heavy and determines the learning pace of our agents. Optimizing the environment execution is thus a relevant enhancement. Another improvement would be to let the developer define the reward function through a parameter of the environment.

References

1. Abadi, M., Agarwal, A., Barham, P., et al.: TensorFlow: Large-scale machine learning on heterogeneous systems (2015), https://www.tensorflow.org/
2. Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., Zaremba, W.: Openai gym. arXiv preprint arXiv:1606.01540 (2016)
3. Calhamer, A.B.: The Rules of Diplomacy. Avalon Hill, 4 edn. (2000)
4. Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S., Wu, Y., Zhokhov, P.: Openai baselines (2017)
5. Drogoul, A.: When ants play chess (or can strategies emerge from tactical behaviours?). In: European Workshop on Modelling Autonomous Agents in a Multi-Agent World. pp. 11–27. Springer (1993)
6. Fabregues, A., Sierra, C.: Dipgame: a challenging negotiation testbed. Engineering Applications of Artificial Intelligence **24**(7), 1137–1146 (2011)
7. Ferreira, A., Lopes Cardoso, H., Reis, L.P.: Strategic negotiation and trust in diplomacy–the dipblue approach. In: Transactions on Computational Collective Intelligence XX, pp. 179–200. Springer (2015)
8. Hill, A., Raffin, A., Ernestus, M., Gleave, A., Traore, R., Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S., Wu, Y.: Stable baselines. https://github.com/hill-a/stable-baselines (2018)
9. de Jonge, D., Baarslag, T., Aydoğan, R., Jonker, C., Fujita, K., Ito, T.: The challenge of negotiation in the game of diplomacy. In: Lujak, M. (ed.) Agreement Technologies 2018, Revised Selected Papers. pp. 100–114. Springer International Publishing, Cham (2019)
10. Jonge, D.d., Sierra, C.: D-brane: a diplomacy playing agent for automated negotiations research. Applied Intelligence **47**(1), 158–177 (2017)
11. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M.: Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602 (2013)
12. OpenAI: Openai five, https://blog.openai.com/openai-five/
13. Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., Lerer, A.: Automatic differentiation in pytorch. In: NIPS-W (2017)
14. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms (2017)
15. Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al.: Mastering the game of go with deep neural networks and tree search. nature **529**(7587), 484 (2016)
16. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. The MIT Press, second edn. (2018)
17. Tesauro, G.: Td-gammon, a self-teaching backgammon program, achieves master-level play. Neural computation **6**(2), 215–219 (1994)
18. Wu, Y., Mansimov, E., Liao, S., Grosse, R.B., Ba, J.: Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation. CoRR **abs/1708.05144** (2017)