FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

A Reinforcement Learning Environment for Cooperative Multi-Agent Games: Enhancing Negotiation Skills

José Aleixo Peralta da Cruz

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Prof. Henrique Lopes Cardoso

July 19, 2019

A Reinforcement Learning Environment for Cooperative Multi-Agent Games: Enhancing Negotiation Skills

José Aleixo Peralta da Cruz

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

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

July 19, 2019

Abstract

In artificial intelligence, an *agent* is an autonomous program that interacts with its environment over time and decides its actions according to a given objective function. In a multi-agent system (MAS), multiple agents interact with each other. A MAS is decentralized, meaning that no agent controls other agents. Certain scenarios are inherently decentralized because the entities that inhabit it may need to act independently, with different or conflicting goals and information. Such situations can be naturally modeled as a MAS.

There are situations where an agent must cooperate with another agent in the same environment to achieve its objective. However, there are cases where alternating between cooperative and adversarial behavior is the only way to achieve the final goal. Diplomacy is a multi-player board game, where the central aim of a player is to conquer Europe. To do so, a player must negotiate alliances and coordinate strategies with other players. The catch is that these agreements may either be public or private, and they are non-binding. No player has full information regarding its opponents' intentions, and any player may break its established agreements. Creating a computer player (bot) capable of taking these factors into account is a difficult task because of the huge amount of possible plays and outcomes, and because it is challenging to measure the true strength of a player's position.

The solution we propose is a custom Diplomacy environment that enables reinforcement learning agents to submit agreements to other players. Agents can use this environment to learn how to negotiate using reinforcement learning methods. We created three different scenarios based on the same environment, which were designed to focus on different aspects of negotiation in Diplomacy. We tested each case with state-of-the-art reinforcement learning methods.

We analyzed the success of our approach by comparing agents trained in our environment with other bots built for the Diplomacy game. Because Diplomacy is a game heavily reliable on negotiation tactics, a bot that has a strong negotiation capability is more likely to win the game. However, there are few implemented Diplomacy bots that take advantage of being able to make agreements. From the existing bots, we compare our solution to bots that simplify the game's rules by assuming that the deals made are binding, eliminating the problem of trust.

We found that while our environment is functional, and allows an agent to learn to act in Diplomacy scenarios with a reduced scope of action, the learning process for more complex situations is extremely slow, even with an adjusted negotiation scope. Software and hardware limitations translate into a prohibitively long time for the training process in our environment, which allied to the extensive observation and action spaces results in an agent that is not proficient in playing Diplomacy. However, with the environment we developed, we hope to enable further research in this game and its particular challenges.

Continuing to tackle problems that have a social aspect using reinforcement learning techniques, and improving current solutions, could lead to systems capable of making better decisions in scenarios where negotiating and compromising is necessary. These systems could be used to

study situations with real-life applications in areas such as social science, politics, and economics, helping humans make better decisions in complicated conditions.

Resumo

Em inteligência artificial, um *agente* é um programa autónomo que interage com o ambiente ao longo do tempo e decide as suas ações de acordo com uma função objetivo. Num sistema multi-agente (SMA), vários agentes interagem uns com os outros. Um SMA é descentralizado, isto é, nenhum agente controla os outros agentes. Certos cenários são inerentemente descentralizados porque as entidades que lhes pertencem agem de forma independente, com informação e objetivos diferentes ou conflituosos. Tais situações podem ser modeladas como um SMA.

Há situações em que um agente tem de cooperar com outro agente no mesmo ambiente para atingir a sua meta. No entanto, há casos onde alternar entre comportamento cooperativo e competitivo é a única forma de atingir o objetivo final. O Diplomacy é um jogo de tabuleiro multi-jogador, cujo principal objetivo é conquistar a Europa. Para tal, um jogador deve negociar alianças e coordenar estratégias com outros jogadores. A peculiaridade do jogo é que estes acordos podem ser públicos ou privados, e não-vinculativos. Nenhum jogador conhece as intenções do seu oponente e qualquer jogador pode quebrar qualquer acordo. Criar um jogador virtual (bot) capaz de ter em conta estes fatores é uma tarefa complexa, devido ao elevado número de jogadas e resultados possíveis, e porque é muito difícil um jogador avaliar a sua verdadeira posição e força no jogo.

A solução que propomos é um ambiente de Diplomacy personalizado, que permite que agentes que usem *reinforcement learning* possam submeter acordos aos outros jogadores. Criámos três cenários diferentes baseados no mesmo ambiente, que foram desenhados para se focarem em diferentes aspetos da negociação no Diplomacy. Testámos cada caso com métodos do estado da arte de *reinforcement learning*.

Analizámos o sucesso da nossa abordagem comparando os agentes treinados no nosso ambiente com outros bots construídos para o Diplomacy. Como o Diplomacy é um jogo que depende bastante da tática de negociação, um bot que saiba negociar bem tem mais hipóteses de ganhar o jogo. No entanto, há poucos bots de Diplomacy que tirem vantagem de poderem fazer acordos. Daqueles que existem, comparamos a nossa solução a bots que simplificam as regras do jogo, assumindo que os acordos feitos não podem ser quebrados, eliminando assim o problema da confiança.

Concluímos que apesar de o nosso ambiente ser funcional e permitir que um agente aprenda a agir em cenários do Diplomacy com um âmbito de ação reduzido, o processo de aprendizagem para situações mais complexas é extremamente lento, mesmo com um âmbito de negociação ajustado. Limitações de *software* e *hardware* traduzem-se num ciclo de aprendizagem muito demoroso, que aliado ao extenso espaço de observação e ação resulta num agente que não é proficiente no jogo de Diplomacy. Ainda assim, com o ambiente que desenvolvemos, esperamos facilitar pesquisas futuras sobre jogo e os seus desafios.

Continuar a resolver problemas que têm um aspeto social usando *reinforcement learning* e continuar a melhorar soluções atuais poderá levar a sistemas capazes de tomar melhores decisões em cenários onde negociação e compromisso sejam necessários. Estes sistemas podem ser usados

para estudar situações com aplicações na vida real, em áreas como a ciência social, a política e a economia, ajudando humanos a tomar melhores decisões em situações complicadas.

Acknowledgements

I would like to thank my supervisor Professor Henrique Lopes Cardoso for the guidance and support. I would also like to thank my family for always being encouraging and making it possible for me to do what I enjoy the most.

José Aleixo Cruz

"A man provided with paper, pencil, and rubber, and subject to strict discipline, is in effect a universal machine."

Alan Turing

Contents

List of Figures

LIST OF FIGURES

List of Tables

Abbreviations

A2C	Advantage Actor-Critic
A3C	Asynchronous Advantage Actor-Critic
ACKTR	Actor Critic using Kronecker-Factored Trust Region
AI	Artificial Intelligence
ANAC	Automated Negotiating Agents Competition
CNN	Convolutional Neural Network
DRL	Deep Reinforcement Learning
MAS	Multi-Agent System
MDP	Markov Decision Process
ML	Machine Learning
NN	Neural Network
PPO	Proximal Policy Optimization
RL	Reinforcement Learning
RPC	Remote Procedure Call
TRPO	Trust Region Policy Optimization

Chapter 1

Introduction

Artificial intelligence (AI) has become one of the most prominent fields of computer science during the past decade. The evolution of electronic components allowed computers to process vast amounts of information quickly and cheaply, which in turn revamped the interest of the community in machine learning (ML). The ability to learn from experience used to be associated uniquely with humans, however, modern software is also able to produce knowledge. Reinforcement learning gives a software agent the capability of adaptation by training. While computers are getting better at overcoming obstacles, they still have great difficulty with acting and adjusting to highly complex scenarios.

Humans live in society, meaning that we must often negotiate and cooperate with other people to achieve our goals. We make decisions on our daily interactions based on previous experience and intuition. For example, our trust in a person depends on past interactions and affects the way we communicate and act with that individual. AI is generally employed to help humans make decisions regarding a particular problem. However, current AI programs are not capable of deciding on socially complicated scenarios, where negotiation and cooperation are very intricate, as well as humans.

Games have always been an essential test-bed for AI research. There has been extensive research aimed mostly at adversarial games. However, games where negotiation and cooperation are encouraged or necessary have not been the target of such a thorough investigation. Experimenting with this type of games is important because they mimic the actual interactions that occur in a society. Negotiating, reaching an agreement and deciding whether or not to respect that agreement is all part of a person's daily life.

The work presented in this document focuses on the cooperation aspects of games. Instead of taking strategy into account, we will try to improve a software agent's performance on a decision-making scenario by negotiating with other agents.

In the remaining of this chapter, we expose the motivations, objectives, and structure of this dissertation.

1.1 Motivation

Stochastic scenarios are everywhere in the natural world. Decision-making in these circumstances is not as straightforward as in deterministic situations. Even humans struggle to make choices when faced with uncertainty. Game theory models have come to facilitate this task, allowing one solution or strategy to be used for multiple problems of the same type. These models arise from tests conducted across a diversity of games.

Researchers have focused mostly on adversarial games with one opponent, such as Chess [Dro93] and Go [SHM+16]. However, in many real-life situations, different entities can be cooperative, instead of strictly competitive. Such scenarios can be naturally modeled using a multi-agent system (MAS), where independent entities interact with each other. To address this type of situations, some attention has slowly shifted to cooperative games ([NNN18] and [Ope18]). In cooperative games, different players cooperate with each other to surpass a certain obstacle or reach a determined goal. Still, few MAS-related works include the social aspect of cooperation.

Diplomacy [Cal00] is a board game which is incredibly complex. It involves adversarial as well as cooperative decisions. Players can communicate with each other and reach agreements or create alliances. But the deals they make are not binding and the players may betray allies. 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, achieving the best performance in the game using a deterministic method is very complicated. Instead, we will explore the usage of reinforcement learning methods, which have been very successful in making an agent learn to play a game. However, the time and storage requirements of tabular methods are prohibitive. As such, approximate RL methods must be employed.

1.2 Objectives

The objective of this work is to design and implement an architecture for a single agent capable of learning how to incorporate social aspects on its decision-making process. This capability is crucial to winning a game of Diplomacy. This aspiration encompasses smaller goals, such as modeling a reinforcement learning approach to cooperative negotiation games, implementing the model and then training it.

In the end, we will evaluate the model in experimental scenarios and try to answer the following research questions:

1. How to create an environment with negotiation for reinforcement learning agents?
2. Is a reinforcement learning model appropriate for negotiation scenarios with a large search space?
3. How should a reinforcement learning model be structured to learn how to negotiate in games with cooperation and competition?

With the answers to these questions, we wish to understand if a program could adapt itself to both known and unknown situations where social aspects influence the final outcome. Associating cooperative and adversarial tactics is extremely useful in numerous real-world game theory scenarios, such as in politics and economics.

1.3 Document Structure

The current document comprises six more chapters. In Chapter 2, we provide an overview of essential concepts related to machine learning and reinforcement learning, as well as state-of-the-art deep reinforcement learning techniques, to provide background for what we mention in the subsequent chapters. We also give information about multi-agent systems and negotiation games and introduce the game of Diplomacy. Chapter 3 exhibits relevant work in reinforcement learning, current environments aimed at testing deep reinforcement learning methods and negotiation techniques, and the analysis of the gap between the scope of the current work and other approaches. Chapter 4 contains the framing of Diplomacy in game theory and the formalization of the problem we propose to solve. Specifically, we formalize negotiation in Diplomacy as a Markov decision process. Chapter 5 addresses the implementation of our proposal, namely the system structure, the tools utilized, and the implementation details. Chapter 6 includes the details about the performed experiments, respective results, and discussion. Chapter 7 presents the conclusions of the current project, which include an overview of all the work done, the answers to our initial research questions, and eventual improvements to our solution.

Chapter 2

Background

In this chapter we present an overview of concepts that provide the basis for the rest of the document. We will include notions related to machine learning, reinforcement learning and deep reinforcement learning. These notions are essential to understanding how and why approaches differ from one to another. We base ourselves in established definitions and current literature and try to present all terms with a consistent interpretation. In particular, we gather information from a survey by Li [Li17] and a book by Sutton [SB18] and complement it with papers about the latest state of the art.

2.1 Machine Learning

Machine learning is the study of computer algorithms that learn from data and improve their performance in making predictions and decisions. It is a subarea of artificial intelligence. Its principles are based on probability theory, statistics, and optimization. A machine learning algorithm has an *input data-set*, a *loss function*, and a *model*. The input data-set contains the information that the model should use to generate its output. The loss function is the penalty for a bad output. The model tries to compute a result to get the lowest loss possible. In mathematical terms, it tries to minimize the loss function. This process of optimization is called *learning* or *training*.

There are three big branches of machine learning: *supervised learning*, *unsupervised learning* and *reinforcement learning*. In supervised learning, the input data is labeled, which means that an example - a particular piece of data - contains the values of the features and of one or more labels. A *feature* is an independent variable and a *label* is a dependent variable whose value is influenced by a set of features. A supervised learning model tries to predict the label value from the given set of feature values. Because the training data-set is labeled, the feedback is explicit, that is, the model knows whether it got the output correctly or not by comparing it to the values of the labels. In unsupervised learning, the input data contains features but is not labeled. No feedback is given, so the model tries to find patterns in the data on its own. In reinforcement learning, there

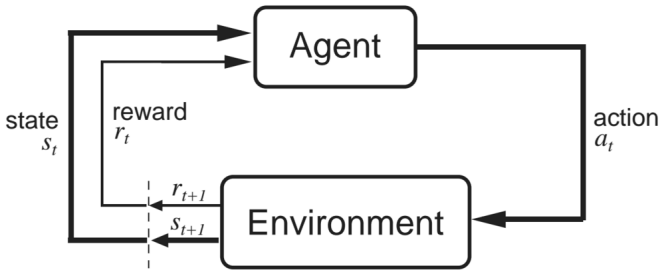

Figure 2.1: Agent-environment interface [SB18]

is *evaluative*, not explicit, feedback. The feedback tells the model how well it is doing, but not if it has actually done its task properly or not. It is important to understand that reinforcement learning is not a subset of unsupervised learning as there is feedback. It is not supervised learning either because the feedback is not explicit.

2.2 Reinforcement Learning

An intelligent *agent* is a program that perceives the environment through sensors and acts upon it using actuators. In reinforcement learning, an agent interacts with its surroundings over time, which we call the *environment*. At each time step t, the agent perceives a set of environmental conditions, which make up a *state* s_t, and selects an *action* a_t to take. Depending on the resulting state s_{t+1}, the agent gets a *reward* r_{t+1}. This interaction is illustrated in Figure 2.1.

For instance, if an agent is learning how to drive a car according to traffic laws, the reward it receives should depend on whether or not it obeys the law. If it accelerates when the traffic light is red, it should get a low reward. If it stops on red lights, it should get a high reward. Eventually, the agent will learn that the best thing to do is stopping on red lights, which is how you respect the law. The objective of a reinforcement learning agent is to maximize the reward function $\mathscr{R}$. Previously, we referred that a machine learning algorithm has a loss or cost function that it should minimize. In RL, the cost function is the inverse of $\mathscr{R}$.

According to Sutton [SB18], reinforcement learning algorithms have the following elements:

- a *policy* π, which defines the learning agent's way of behaving at a given time;
- a *reward function* $\mathscr{R}$, which defines the goal of a reinforcement learning problem;
- a *value function* $\mathscr{V}$, which defines the amount of reward an agent may accumulate over time from a given state.

If the transitions of the environment are known, there may be an additional element that is the *model of the environment*, which is used to plan actions, by predicting their consequences before actually committing to them.

To better understand each element, we should formalize a reinforcement learning problem. We can describe one as a Markov Decision Process (MDP), where:

- S is a finite set of states;
- A is a finite set of actions;
- $P_a(s_t, s_{t+1})$ is the probability that an action a on state s_t at a given time t will lead to state s_{t+1} at time $t+1$;
- $R_a(s_t, s_{t+1})$ is the immediate reward r_{t+1} of transitioning from state s_t to state s_{t+1} after taking action a.

In a MDP, the probability of each possible value for s_{t+1} and r_{t+1} depends only on the immediately preceding state and action - it does not depend on the history of previous states and actions. Because of this, a state in a MDP must include all information about the past interactions that makes a difference in future decisions. In mathematical terms, this translates to $P(s_{t+1}|s_t) = P(s_{t+1}|s_1, s_2, ..., s_t)$. A state s_t that obeys this constraint possesses the *Markov property*. $P_a(s_t, s_{t+1})$ is the *transition function*, which represents the environment's dynamics. If the transition function is well defined, we may create a model of the environment. *Model-based* reinforcement learning methods simulate interactions with the environment using the created model, as opposed to *model-free* methods that learn by interacting directly with the environment. For the remaining of this document, we will only be considering model-free methods, as most studied scenarios are too complex to be adequately used with model-based approaches.

A **policy** is a probability distribution that defines the likelihood of an agent choosing a certain action given the state of the environment. In a MDP, a policy is a mapping from S to probabilities of selecting each element of A. If an agent is following a policy π at time step t, then $\pi(a|s)$ is the probability of $a_t = a$ if $s_t = s$. For example, a greedy policy would make the agent always choose the action that gives the best reward. That is, if a is the action that leads to the highest reward in state s, then $\pi(a|s) = 1$, while all other actions would have an associated probability equal to 0. However, this would mean that no action besides the best one was tried by the agent. Executing an action with the best **immediate** reward may not lead to the best total reward, meaning the agent would get stuck in a local minimum. Effectively, one of the greatest challenges in reinforcement learning is maintaining a balance between **exploitation** and **exploration**.

Exploiting involves using what an agent already knows to make a decision. *Exploring* involves learning new things. A simple example of a policy that allows an agent to explore is the ε-greedy policy, where the agent chooses greedily most of the time but, with probability ε, it selects among all the actions with equal likelihood. This allows the agent to explore new options and get closer to the global optimum solution.

The **reward function** defines the values of the rewards for each possible transition in a MDP, that is, the values of $R_a(s_t, s_{t+1})$ for every $a \in A$ and $(s_t, s_{t+1}) \in S \times S$. A reinforcement learning agent will maximize the reward function by selecting the set of actions that triggers the transitions that provide the highest combined reward.

The **value function** is an element that helps the agent to achieve the best global solution, by allowing it to estimate how promising a certain state (or a state-action pair) is. For instance, an agent may choose an action with lower immediate reward in favor for a state that is more promising, so that it receives more reward in the long-run. At a certain time step t, we can define the *expected return* or *expected reward* of a state, denoted G_t, as Equation 2.1,

$$G_t \doteq \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \tag{2.1}$$

where γ is a *discount factor* with value equal or less than 1. The number of time steps T is set depending on whether the task is *episodic* or *continuous*. In an episodic task, the number of time steps T is finite and all future rewards may or may not have the same weight in the final result. In a continuous task, the number of time steps is infinite, therefore, in order to avoid an infinite sum and ensure convergence, the condition $\gamma < 1$ has to be true.

The value function of a state s under a policy π, denoted $v_\pi(s)$, can then be defined as the expected return when starting in s and following π (Equation 2.2).

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s], \text{for all } s \in S \tag{2.2}$$

The v_π function is the *state-value function for policy* π. From Equation 2.2, we may define the value of taking action a in state s under a policy π as the expected return when starting from s, taking the action a and from that time forward following policy π as Equation 2.3.

$$q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = t] \tag{2.3}$$

The q_π function is the *action-value function for policy* π. We can also write the state-value function in terms of the action-value function, as shown in Equation 2.4, which is the weighted sum of the action-value for every action a available in state s, where the weight is the probability of the action a being chosen according to policy π.

$$v_\pi(s) = \sum_a \pi(a|s) q_\pi(s,a) \tag{2.4}$$

However, the agent does not know the true values of either of these two functions. If it did, the problem would be easily solvable. So the agent **estimates** these values.

One of the most common methods for estimating *action-value*, which is often called the Q-value, is doing the average of n observed rewards for a given action a in a state s (Equation 2.5). With each new observation, a new value must be stored in memory, representing the reward of taking the action on a given time step.

$$Q_n \doteq \frac{R_1 + R_2 + \ldots + R_{n-1}}{n-1} \tag{2.5}$$

The memory requirements grow linearly with this method of value update. However, there is another way of calculating the average keeping the memory requirements constant, described in Equation 2.6. This update rule is the basis of most reinforcement learning methods. The general form is described in Equation 2.7.

$$Q_{n+1} \doteq \frac{1}{n}\sum_{i=1}^{n} R_i = Q_n + \frac{1}{n}[R_n - Q_n] \tag{2.6}$$

$$NewEstimate \leftarrow OldEstimate + StepSize[Target - OldEstimate] \tag{2.7}$$

The expression $[Target - OldEstimate]$ is an *error* or *loss* in the estimate. It is reduced by taking a step toward the *target*. The *target* is presumed to indicate a desirable direction in which to "move", though it may be noisy. Therefore the main objective of a RL algorithm is to reduce the difference between the old estimates and the target value, until the current estimate converges. The *StepSize* parameter determines how much weight the newly found information has on the old value update. The impact of the *target* value shows how important it is to define a suitable reward function for a problem.

2.2.1 Q-Learning

Q-learning [WD92] is a tabular solution reinforcement learning method. In tabular solution methods, the state and action spaces are small enough for the value functions to be represented as arrays or tables. In Q-learning, the value function takes a pair state-action and is updated with the rule in Equation 2.8.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a [Q(S_{t+1}, a)] - Q(S_t, A_t)] \tag{2.8}$$

Q-learning is called an *off-policy method* because it does not follow the current policy π to choose the action a from the next state S_{t+1} in order to update the current state value. Instead, it chooses the greedy (i.e. max reward) action for the update ($\max_a[Q(S_{t+1}, a)]$). In other words, Q-learning assumes that it is always following a greedy policy when it updates its estimates, even if the current policy π is not greedy. An advantage of this separation is that the estimation policy, used to update Q-values, is independent of the behaviour policy, used to choose which states and actions should be updated. It has been proven that Q-learning, provided infinite computation time and a partly-random policy, can always identify an optimal policy for a finite Markov decision process [Mel01].

However, in problems where the search space is too large, tabular methods such as Q-Learning are not appropriate, because of hardware limitations and costs. Instead, we choose to use approximate solution methods. In these methods, the agent makes decisions on states it has not seen before based on previous encounters with different states. The key issue becomes *generalization*. Neural networks are models capable of doing good function approximations and helping on the

task of generalization. This naturally led to the use of deep learning to regress functions for use in reinforcement learning agents, which is called *deep reinforcement learning*.

2.3 Deep Reinforcement Learning

In deep reinforcement learning (DRL), deep neural networks are used to approximate the value function or find the optimal policy. In tabular methods, all values in tables are stored in memory. However, large search trees make storing and evaluating all these values unfeasible. By using a neural network, we can avoid the space and time problem, and calculate values for new and unfamiliar states. Neural networks are used as a function approximator, vastly reducing memory requirements, as shown in Figure 2.2. Most DRL techniques are relatively recent, since they are enabled by the vast power of modern computer processors.

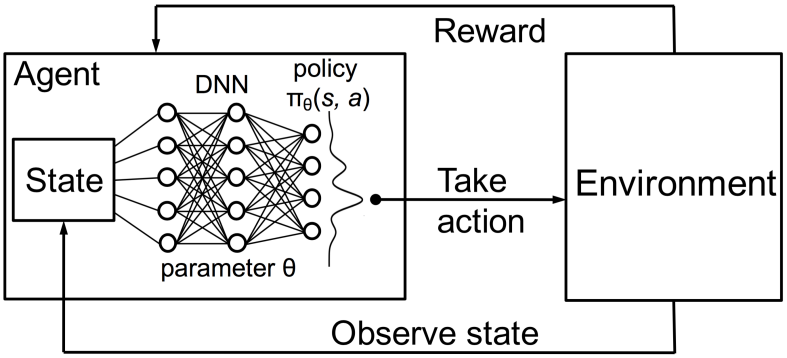

Figure 2.2: Reinforcement learning with policy represented via a deep neural network [MAMK16].

2.3.1 Value-Based Methods

Value-based deep reinforcement learning methods use a *value network*. A value network is a neural network that assigns a value to every state s of the environment by calculating the expected cumulative reward for that state. Every state the agent observes goes through the value network. The states which lead to higher rewards obviously get higher values in the network. Instead of states, the network may assign scores to state-action pairs (s,a), which is the case of the Deep Q-Network.

2.3.1.1 Deep Q-Network

The Deep Q-Network (DQN) method [MKS+13] is a deep reinforcement model that uses a variant of Q-learning. While the traditional Q-learning method requires storing the value function Q for each state-action pair, the DQN *approximates* $Q(s,a)$. In this approach, the Q-value function is generalized, instead of "remembered" by the agent.

Using a deep neural network allows generalization, but carries some challenges. If we take the target $[R_{t+1} + \gamma \max_a [Q(S_{t+1}, a)]$ of the Q-learning method, we observe that it depends on Q itself. When a function approximator such as a neural network is used to represent Q, the target may become unstable, which provokes the neural network to diverge or never stabilize. This instability is due to a couple of causes: the possible correlations in sequential observations and the correlation between action-values and target values.

When employing neural networks, it is important to make sure that samples are independent of each other and follow similar data distributions. To address this, instead of feeding the DQN with a mini-batch of sequential observations, such as the last 100 observations, a mechanism called *experience replay* is used. To perform experience replay, we store a considerable number of observations in a buffer and, instead of feeding them sequentially to the neural network, we extract a mini-batch randomly. This reduces the correlation between observations, because we are using both recent as well as old observations, so they are more independent of each other and the agent doesn't "forget" what it had previously learned.

Because the Q-function is updated with values of itself, changing a Q-value for a single state-action pair also alters the value for other pairs. This leads to a moving target that the neural network keeps chasing, never converging. The solution applied by DQN is to create two neural networks: a Q-network, with a parameter vector θ, and a target-network, with a parameter vector (θ^-). The Q-network is updated as it would normally be, while the target-network is only updated and synchronized periodically with the Q-network. To calculate the target we use the more stable value of the target-network, therefore the target will be less correlated with the Q-value. The loss function for the update rule on time step t effectively becomes what is described in Equation 2.9, which is the function the neural network tries to minimize.

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')}[(r + \gamma \max_a [Q(s', a, \theta_t^-)] - Q(s, a, \theta_t))^2] \tag{2.9}$$

2.3.2 Policy Gradient Methods

In *policy gradient* or *policy-based* methods a policy depends on a vector of parameters θ, in contrast to *value-based* methods such as DQN, which depend on value functions. In a policy-based method, the policy is defined as $\pi(a|s, \theta)$, expressing that choosing an action a depends not only on the current state s, but also on the values of the θ vector. To determine how good a policy with parameters θ is, we use a performance measure, which is a function $J(\theta)$ that takes these parameters and determines their benefit. The learning process tries to maximize the performance measure, which means the updates made to the values of the parameters approximate gradient ascent in J (Equation 2.10).

$$\theta_{t+1} = \theta_t + \alpha \widehat{\Delta J(\theta_t)} \tag{2.10}$$

The performance measure is calculated differently for episodic and continuous cases. For the sake of relevancy, we will only refer to the episodic cases performance. In these cases, the

performance measure is the expected reward of following the policy π with parameters θ starting on the initial state s_0 (Equation 2.11).

$$J(\theta) \doteq v_{\pi_\theta}(s_0) \doteq \sum_a \pi_\theta(a|s_0) q_{\pi_\theta}(s_0, a) \tag{2.11}$$

The gradient of the performance measure is then calculated as shown in Equation 2.12, according to the **Policy Gradient theorem**, where $\mu_{\pi_\theta}(s)$ is the probability of being at state s when following policy π_θ and $\propto$ means proportional to.

$$\begin{aligned} \nabla_\theta J(\theta) = \nabla_\theta v_{\pi_\theta}(s_0) &\propto \sum_{s \in S} \mu_{\pi_\theta}(s) \sum_{a \in A} q_{\pi_\theta}(s,a) \nabla_\theta \pi_\theta(a|s) \\ &= \sum_{s \in S} \mu_{\pi_\theta}(s) \sum_{a \in A} \pi_\theta(a|s) q_{\pi_\theta} \pi(s,a) \frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} \\ &= \mathbb{E}_{\pi_\theta}[q_{\pi_\theta}(s,a) \nabla_\theta \ln \pi_\theta(a|s)] \end{aligned} \tag{2.12}$$

A policy's parameters can be chosen arbitrarily, as long as $\pi(a|s,\theta)$ is differentiable with respect to its parameters. One common way of defining θ is as the vector of all the weights of a deep neural network that tries to choose the best action for a given state. Then, a numerical preference $h(s,a,\theta)$ is associated to each state-action pair. Actions with higher h value have higher probabilities of being selected.

One of the advantages of policy-based methods over value-based methods is that the policy may be a simpler function to approximate than the value function. For extensive spaces, the number of state-action values to learn can be quite large. Policy-based methods avoid this because they only need to learn a number of parameters that is far less than the space count. Another advantage is that policy-based methods can learn stochastic policies, while action-value methods have no natural way of finding stochastic policies, as they converge to a deterministic policy. In scenarios that require selecting actions with arbitrary probabilities, a stochastic policy leads to the best outcome. For instance, when playing a game of Rock-Paper-Scissors against an opponent, having a deterministic policy allows the adversary to counter every move we do. Hence, the best solution is to select an action with a certain probability.

A major problem with the "vanilla" policy gradient loss referred in Equation 2.12 is that the diversity of the set of states visited in an episode (also called trajectory), can result in a high discrepancy between the cumulative reward values of different episodes. Consequently, generated gradients are too noisy and the variance is too high. One way of reducing the variance and increasing learning stability is to subtract the cumulative reward by a baseline $b(s_t)$.

For example, let's say the values of $\nabla_\theta \ln \pi_\theta(a|s)$ are $[0.5, 0.3, 0.2]$ and the values of $q_{\pi_\theta}(s,a)$ are $[1000, 1001, 1002]$ for three different trajectories. The variance of the product of the values (which originates from Equation 2.12) is equal to $Var(0.5*1000, 0.3*1001, 0.2*1002) = 232700$. By using a simple constant baseline $b(s_t) = 1000$, the variance then becomes $Var(0.5*0, 0.3*1, 0.2*2) = 0.0433333$, which is a much smaller value.

2.3.2.1 Advantage Actor-Critic (A2C)

Some methods explore the incorporation of a baseline in order to stabilize gradients and accelerate learning. *Actor-critic* methods are policy-based methods that also learn to approximate a value function. These are considered hybrid methods, as they combine approaches used in value-based methods and policy-based methods. The purpose of learning the value function is precisely to use it as a baseline during the learning process and assist on the policy update.

An *actor-critic* method has a learning model split in two components, depicted in Figure 2.3. One component, labeled the *actor*, determines the action to take on a given state. The actor is a reference to the learned policy, which dictates how the agent acts. The other component is called the *critic* and evaluates the impact of that action, which is a reference to the value function. *TD error* (*temporal-difference error*) is the difference between the estimates of a value at different time steps (Equation 2.7). The actor will update its values, and therefore the choices it makes, according to the TD-error calculated by the critic. The critic will also make adjustments to its values based on the TD-error.

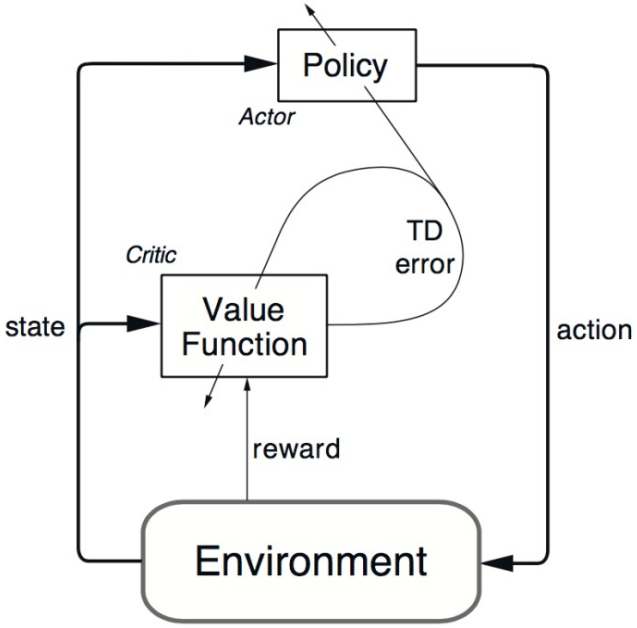

Figure 2.3: The actor-critic method architecture. [Yoo19]

Another common baseline to use when learning a policy is the advantage of taking an action a on a state s, given by $A(s,a) = Q(s,a) - V(s)$. The advantage function captures how good an action is compared to other actions that may be taken in the same state.

The **Advantage Actor-Critic (A2C)** [WML+17] method combines both notions in its name. In A2C, the *critic* learns the advantage function, instead of the Q-values, thus reducing the high

variance of policy networks and stabilizing the model. It is based on the **Asynchronous Advantage Actor-Critic (A3C)** [MBM+16]. In A3C, different workers run parallel environments and asynchronously update a global network. However, experimental results determined that there was no gain with the asynchronous feature of A3C and removing it simplified implementations, resulting in the A2C.

An algorithm that improves on the concept of A2C is the **Actor Critic using Kronecker-factored Trust Region (ACKTR)** [WML+17]. ACKTR uses distributed Kronecker factorization [MG15] to optimize the calculation of the gradient update. This update can be 25% more expensive than A2C, but it leads to better results.

2.3.2.2 Proximal Policy Optimization (PPO)

Policy-gradient methods are generally too sensitive to the choice of the *StepSize* parameter (Equation 2.7). If the steps are too small, the learning progression is extremely slow. If the steps are too large, the model may overshoot an optimal region and end up with a noisy learning signal.

The **Trust Region Policy Optimization (TRPO)** [SLM+15] tries to fix this issue by setting a constraint (Equation 2.13) on how much a new policy can differ from the old one, using the Kullback–Leibler divergence (KL) [KL51], which measures the difference between two probability distributions.

$$\begin{aligned} &\underset{\theta}{\text{maximize}}\ \mathbb{E}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_old}(a_t|s_t)}A(s_t,a_t)\right] \\ &\text{subject to}\ \mathbb{E}_t[KL[\pi_{\theta_old}(\cdot|s_t),\pi_\theta(\cdot|s_t)]] \leq \delta \end{aligned} \tag{2.13}$$

With this constraint, the policy is kept in a trust region, so that it avoids taking larger steps than recommended.

The **Proximal Policy Optimization (PPO)** [SWD+17] improves on TRPO by embedding the constraint on the objective function. This facilitates and speeds up calculations while maintaining the policy in a trust region. Let $r_t(\theta)$ denote the probability ratio $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_old}(a_t|s_t)}$, so that $r_t(\theta_{old}) = 1$ and ε is a hyper-parameter. Then the objective function of PPO is given by Equation 2.14.

$$\underset{\theta}{\text{maximize}}\ \mathbb{E}_t[\min(r_t(\theta)A(s_t,a_t), \text{clip}(r_t(\theta), 1-\varepsilon, 1+\varepsilon)A(s_t,a_t)] \tag{2.14}$$

The first term of the objective function is the objective function of TRPO. The second term clips the ratio between the old and new probability distributions of the policy, removing the incentive for changing the policy beyond $[1-\varepsilon, 1+\varepsilon]$. The minimum term is used so the final objective is a lower (pessimistic) bound of the unclipped objective. The performance improvement of PPO over A2C and TRPO for the same environments can be visualized in Figure 2.4.

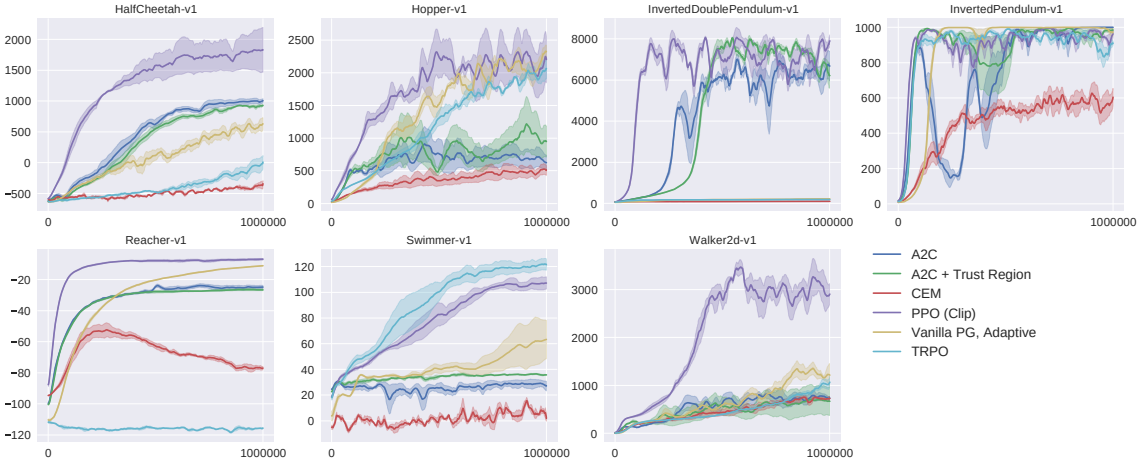

Figure 2.4: Comparison of several algorithms on different OpenAI Gym (Section 3.2.3) MuJoCo environments, training for one million time steps [SWD+17] (CEM stands for "cross-entropy method" [RK13]).

2.4 Multi-Agent Systems and Negotiation Games

In a multi-agent system, two or more agents exist and interact in the environment (Figure 2.5). Each agent is autonomous in their decision making and independent from the others. The system is decentralized because no agent determines the decisions of other agents. If the agents are not in the same physical location, but still share information by communicating with each other, then the system is also distributed. Some scenarios are inherently decentralized and may be naturally modeled as a multi-agent system. For example, if different people or organizations have different goals and conflicting information, a MAS is useful to manage their interactions [SV00]. Some other problems have characteristics that are solved more easily with a multi-agent system model. A task that may be divided into independent sub-tasks that can be handled by different agents could benefit from the parallelism that a MAS provides as a distributed system.

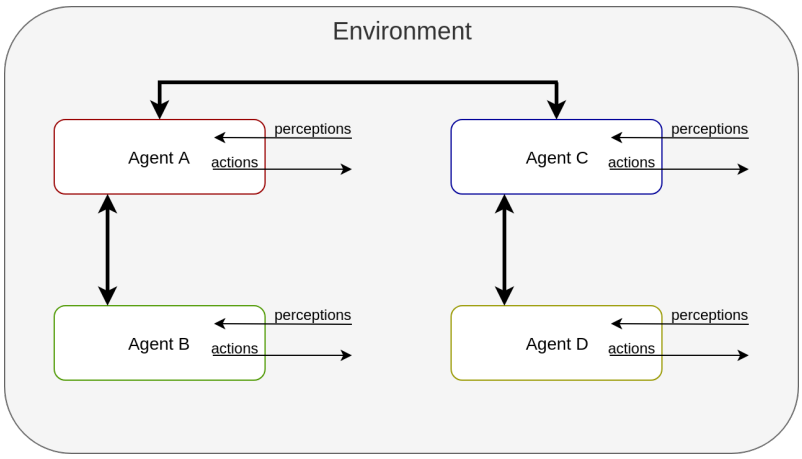

Figure 2.5: Interactions between agents in a multi-agent system.

Games have always been a target of research to experiment problem-solving methods that could be applied in real-life scenarios. However, most research has been aimed towards adversarial games. According to Nash [Nas51], *non-cooperative* (or competitive) games involve competition with individual opponents, where alliances may be created but are only *self-enforced*. This means that there is no external authority obliging players to maintain their agreements. The alliance is kept while the players deem it convenient. By contrast, *cooperative* games have *coalitions* (groups of individual players) competing against each other. These coalitions are enforced by a third party, effectively prohibiting players from breaking an arrangement.

If there is no communication between players, then the coalitions are formed due to *Nash equilibrium* [N$^+$50] or to *explicit action constraints* [GGR88], in competitive and cooperative scenarios, respectively. In a Nash equilibrium, if players have nothing to gain by competing against each other, they will form a self-enforced coalition. With constraints, a player is bound not take any action that may harm other players. However, if there is communication between players, then they may coordinate with each other to reach agreements, whether they are binding (cooperative games) or not (non-cooperative games). Games with communication may be modeled under game theory as *bargaining* or *negotiation games* [OR94].

Negotiation serves as a conflict solver between entities. For instance, people negotiate between themselves to achieve an agreement. Whether it is setting the time for a meeting or purchasing a house, negotiation is much present in the real-word. In a multi-agent system, each agent has its own set of goals and utility functions. Sometimes, it is beneficial for agents to cooperate with each other, but issues or conflicts may arise. Therefore agents also need to negotiate with each other. We will use negotiation games to test the efficacy of deep reinforcement learning in negotiation, more specifically a game called Diplomacy.

2.4.1 Diplomacy: the Game

Diplomacy [Cal00] is a board game with 20th century Europe as a setting. The game is best played by seven players. Each player represents one of the seven *great powers*: England, France, Austria, Germany, Italy, Turkey, and Russia. The objective of a great power (or country) is to conquer Europe.

A Diplomacy board (Figure 2.6) has 34 *provinces*. There are three types of provinces: *inland*, *water* and *coastal*. Some provinces have a *supply center*. Each great power possesses two types of *units*: *armies* and *fleets*. Only *armies* move on *inland* provinces and only *fleets* move on *water* provinces. Both types of units may occupy a *coastal* province.

At the start of a game, each *power* controls three supply centers, with the exception of Russia, which controls four. The remaining 12 supply centers are not occupied at the start of the game. The number of units a power controls at the beginning of the game is the same as the number of supply centers it owns.

In Diplomacy, a turn represents six months of time. The first turn is called the *Spring* turn and the next the *Fall* turn. In each turn, players may give units orders to *hold* (stay in the same province), *move* (attack adjacent provinces) or *support* (support the orders of units from the same

Figure 2.6: Figure depicting a standard board for a Diplomacy game. The circles represent *supply centers*. [Add01]

or other players). Because all units have the same strength, the only way to win conflicts (*standoffs*) is to support other units' orders. Each supporting unit adds one unit of strength to the order it is supporting.

The characteristic that makes Diplomacy appropriate for the study of cooperation and negotiation is the *Diplomatic Phase* that occurs at the beginning of every turn. In this phase, players meet together one-to-one or in small groups. They may create alliances publicly or secretly and coordinate unit orders with each other. However, the agreements that the players make are not binding. A player may disrespect an understanding or an alliance. To survive, a player needs to cooperate with others. To win, a player must eventually stand alone.

After the Diplomatic Phase, the players write down the orders they will give to each of their units. The plans are all revealed at once and are not reversible. The conflicts that emerge from the orders given are solved and the players settle the ownership of the provinces according to the conflicts' outcomes. A power gains or loses units in accordance with the number of supply centers it controls. If one power controls 18 or more supply centers at the end of a Fall turn, that player is declared the winner and the game ends.

Chapter 3

Related Work

In this chapter, we provide an overview of relevant breakthroughs in reinforcement learning, the current state of the art of deep reinforcement learning applied to games, and related work regarding negotiation in Diplomacy. We also evaluate the gap between the scope of the current project and other endeavors.

3.1 Reinforcement Learning Applied to Games

Donald Michie created one of the first systems to use reinforcement learning in 1961 [Mic61]. It consisted of a set of simple matchboxes that allowed an operator to play Tic-Tac-Toe (Noughts and Crosses) against a human. Michie eventually developed his idea by modeling problems into sub-games and solving them with a program called BOXES [MC68] which was capable of balancing a simulated pole in a cart. BOXES obtained knowledge through trial and error. Each "box" would contain information about what action to take and how many times it worked well towards the final objective, considering the actions taken before-hand.

In 1992, Christopher Watkins formalized the Q-Learning technique (presented in Section 2.2.1). The introduction of the *off-policy* Q-value function vastly improved the practicability and feasibility of reinforcement learning and boosted its interest in the scientific community.

A few years later, Gerald Tesauro created a neural network capable of learning how to play backgammon by playing against itself [Tes94]. The network, called TD-Gammon, achieved a staggering level of performance by closely matching world-class human players. It also surpassed the performance of a previous supervised learning approach of Tesauro [Tes89], trained with the data of actual human games, revealing the great potential behind reinforcement learning.

In 2013, DeepMind researchers combined deep neural networks with reinforcement learning to play more than one game of Atari, using only the frames of the game as an input (see Section 3.2.1). It was one of the first works using neural networks to approximate value functions for reinforcement learning and of generalizing the representation of the environment enough for the

agent to learn multiple games. DeepMind's research boosted the interest in deep reinforcement learning, showing that agents can learn how to execute complex tasks with performance equal or higher than humans.

3.2 Deep Reinforcement Learning Environments

Much of the recent research accomplished in deep reinforcement learning involves creating or mimicking an environment and then applying novel methods to solve the problem imposed by that environment. In this section, we explore some modern DRL research environments applied to games.

3.2.1 Atari Games

One meaningful challenge that deep reinforcement learning agents face is coming up with a meaningful representation of the input information they receive. The difficulty of this challenge increases with the dimension of the state representation. A video game, for instance, is constantly transmitting complex information to a human player. Mnih et al. [MKS+13, MKS+15] approached this challenge by creating an agent capable of playing Atari games, having the frames of the game as the only state representation. The frame's pixels are pre-processed and then fed to a Deep Q-Network with experience replay (see Section 2.3.1.1). The first layers of the DQN network are convolutional while the remaining are fully connected (Figure 3.1). For a frame of input, the neural network outputs the Q-values for the different actions that can be taken on that state. These actions map to the same controls a human has when playing the game. This approach achieved super-human performance in several Atari games, revealing the power of convolutional neural networks to generate representations of a complex environment.

3.2.2 Go

In 2016, DeepMind researchers created AlphaGo [SHM+16], a program capable of playing the Go board game. Go is a game of perfect information. However, the number of different sequences of moves is approximately 250^{150}, which makes exhaustive tree search infeasible. AlphaGo became a landmark when it defeated the number one ranked player in the world in a series of Go games.

AlphaGo uses convolution neural networks (CNN) to construct a representation of the game environment from the image of the board, similarly to the approach to Atari games (described in Section 3.2.1). This representation is passed to neural networks which reduce the breadth and depth of the action search tree. The depth of the search tree is reduced by replacing the sub-tree of each state s by its predicted value (which the probability of winning the game from that state), using an approximate value function $v(s)$. The breadth of the search is diminished by sampling the actions from a policy $\pi(s,a)$, which results in only considering a small set of actions rather than all possible actions. The neural network that approximates the value function is the *value network* and the neural network that determines the policy is the *policy network*. The policy

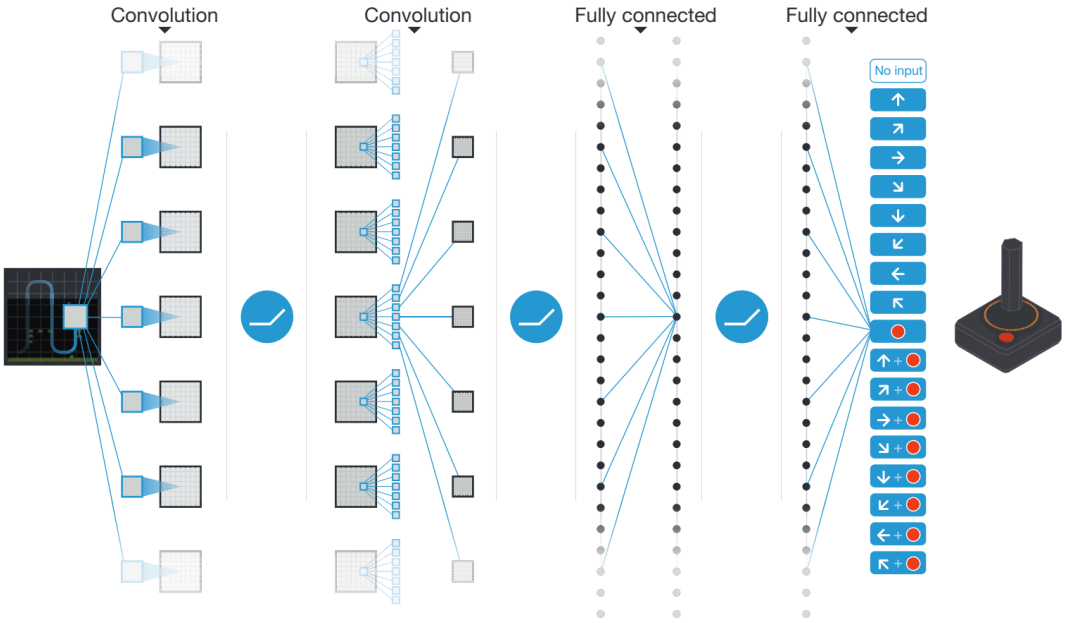

Figure 3.1: Schematic illustration of the convolutional neural network used to learn to play Atari games.

network is initially trained using supervised learning, with data from samples of games between Go professionals. After the initial training, the policy network is trained through *policy gradient reinforcement learning*.

The policy is improved through self-play. A modified Monte Carlo method called Monte Carlo Tree Search (MCTS) algorithm is implemented to take into account the policy values and explores moves with a lookahead search, expanding the most promising states and eventually selecting what considers to be the best move.

An improved version of AlphaGo called AlphaGo Zero [SSS+17] learned how to play Go without any human input or feedback. While the original AlphaGo was trained using data from actual human games, AlphaGo Zero learned how to play from scratch and only from playing against itself. AlphaGo Zero's performance surpassed the original program's, becoming the world's best Go player in just 40 days [SH17].

3.2.3 OpenAI Gym

OpenAI Gym [BCP+16] 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 just needs little (if any) adaptation to act on different environment, 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 the

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. Default is $[-\infty, \infty]$.

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 reuse a solution in different scenarios is very beneficial, as it adds to its usefulness. While Gym is directed to reinforcement learning, since it is built on Python, it is easier to connect existing Python machine learning libraries with Gym agents and make use of the deep reinforcement learning techniques that those frameworks provide.

3.2.4 Multiple Tank Defence

Multiple Tank Defence is a custom multi-agent environment where either one or two tanks must defend their base together from enemy tanks. The defending tanks must avoid getting hit by the enemies while eliminating them. The terrain of the game is customizable, which ensures that the agents will be challenged with new environment conditions.

Each agent receives a reward of 10 for killing an enemy tank. The necessity for such a simple reward function comes from the fact that agents become easily suck in local minimum solutions due to insufficient exploration in environment with sparse rewards. However, when training without human guidance, the tanks acquired an attacking stance, where they would try to kill as many enemy tanks as possible. To solve this problem, the agents were each given a target map. This map details which sections of the environment a tank can "see" (Figure 3.2). Therefore, it defines the area of action of each agent. The target maps are defined by a human, but have led the agents to take a more defensive and cooperative stance to defend the base. This indicates that providing human strategies leads agents to easily attain an expected solution in a short training time, without needing human feedback during that phase.

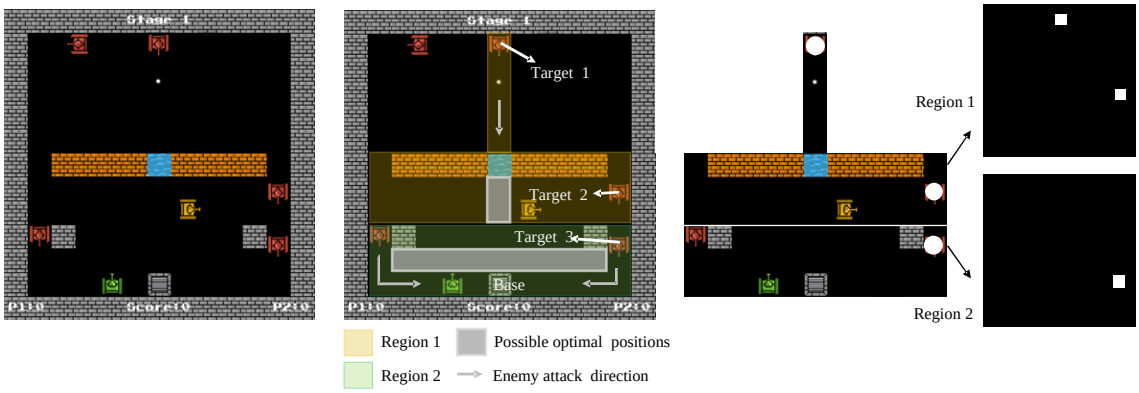

Figure 3.2: An example of the target map in Multiple Tank Defence.

3.2.5 OpenAI Five

Dota 2 is a real time multiplayer strategy game, consisting of two teams of five players. To win, a team must destroy the enemy's base. OpenAI have developed a team of neural networks called OpenAI Five [Ope18] to tackle the problem that Dota 2 presents. The tremendous complexity of Dota 2 poses great challenges to an AI. These challenges include: its long time horizon, where most actions have minor impact individually but others affect the game strategically, with consequences that can play out over an entire game; its partially-observed state as each team only knows what is around each player and base; and its high-dimensional observation and action space (up to 170,000 possible actions per player in a given time step).

OpenAI Five uses a dedicated system to run PPO (an algorithm described in Section 2.3.2.2) with an immense computational power. Several copies of the game are created with independent agents. The agents then sync their experience to a common optimizer. This results in a system capable of learning 900 years-worth of experience in a single day. OpenAI Five improved its performance over the course of two years, starting by beating individual players and eventually dethroning the world-champions of Dota 2. It demonstrated the strength of PPO and parallel learning.

3.3 Multi-Agent Negotiation Environments

Many real-life scenarios can be modeled as games where players benefit from communicating and compromising. One of the challenges of negotiation is determining the utility value of an agreement when it is not explicitly given or when the environment has complex rules. De Jonge & Zhang [DJZ17] developed an adapted Monte Carlo Tree Search agent to calculate the utility value of a negotiation in a perfect information game. Having full information about the opponents negotiations, the agent was able to accurately determine the opponent's utility function. However, realistically not all information is known when negotiating.

3.3.1 Diplomacy Environments

Diplomacy encompasses the challenge of negotiating without knowing the true strength of either player's position, as agreements are made in private. The challenge that Diplomacy poses is an intriguing test for software agents. Some environments have been developed to facilitate the implementation of computer Diplomacy players. In this section, we expose some of them.

DAIDE [RNW07] (*Diplomacy Artificial Intelligence Development Environment*) is a framework that defines a communication model, protocol, and syntax for an environment where multiple Diplomacy agents can compete with each other. The suggested model is a client-server design, where each Diplomacy player and observer are clients that connect to a single server over an IP network. This server handles all game logic. Each player is an independent process, that may communicate with other players through the game server. *Parlance* [RNW09] is a platform-independent Python application that follows the DAIDE specification. It materializes the game server, allowing clients to connect through sockets. Once a player connects to the server, it remains connected during the whole game, listening to the messages the server sends about the state of the game. Upon handling such communications, it decides whether or not it should send information back to the server, depending on the type of message.

DipGame [FSN] is a Java framework for developing bots (computer players) for DAIDE-based servers, such as Parlance. It includes a game manager, which allows to easily select which players should participate in a game and observe the output of a game in real time. It also includes a negotiation server, separate from the game server, which mediates all negotiation between players, obeying to the DAIDE syntax.

BANDANA [JS17] is an extension of DipGame, with a simplified negotiation server and language. It facilitates the creation of bots, control of game settings, and analysis of logs and results and it is implemented in Java. The Diplomacy league of the Automated Negotiating Agents Competition (ANAC) [dJBA+19] asks for participants to conceive their submissions using the BANDANA framework.

Two types of Diplomacy players can be created using BANDANA – 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 previously 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. Tactical orders and negotiations are handled by two different modules in a BANDANA player. 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.

3.3.2 Negotiating Diplomacy Bots

The current work is inspired by previous endeavors to create an AI that can master the Diplomacy game. While none used approximate reinforcement learning methods, each approach showed a distinct negotiation procedure.

Webb et al. [WCW+08] used the DAIDE environment to develop the *Diplominator*. The Diplominator has a custom tactical and negotiation module. At the beginning of the game, the bot tries to propose peace to all other players. Being in peace means that the Diplominator will be less probable to attack units of the power that makes part of the agreement. However, if the allied power is hindering the conquest of a supply center, it is treated as an enemy. If an allied power starts getting too strong, the Diplominator will "back-stab" it. The moment for the back-stab is decided according to how many supply centers it stands to gain by attacking the ally at a given time. If the Diplominator is put in a less favorable position after the back-stab, it will give up on it and re-request friendship.

Ferreira et al. [FLCR15] created a Diplomacy bot called DipBlue using the DipGame framework. DipBlue utilizes a trust ratio to adjust the likelihood of opponents fulfilling agreements. It changes the trust ratio according to the actions taken by opponents, such as attacks or betrayals. It then combines this trust ratio with a value for opponent strength. DipBlue tries to create alliances with the stronger players and attack the weaker ones, with the objective of surviving as long as possible.

Marinheiro and Lopes Cardoso [MLC17] suggested a generic architecture called *Alpha* for agents that act on games with a mix of cooperation and competition, including Diplomacy. Alpha contains different modules with different functions. They communicate between themselves, analyzing the various aspects of the game. For instance, the *Foreign Office* is the module responsible for negotiating and the *Intelligence Office* is in charge of predicting opponents moves. While each agent has its own individual goal, cooperating with other players is encouraged as it results in a better outcome for each actor. Some of the challenges noted are related to accurately evaluating moves and player positions, given the social context of the game. Also, the possibility of betrayals due to non-binding agreement adds a new problem related to trust.

In the 2018 edition of the Automated Negotiating Agents Competition (ANAC), four bots applied for the Diplomacy competition [dJBA+19]. However, there was no winner because no bot was capable of passing the minimum requisites to triumph. The competition was divided into two rounds with the objective of testing both strategies for deal proposal and deal acceptance. The two most notorious submissions to the competition were the bots named *CoalitionBot* and *Gunma*. *CoalitionBot*'s play-style is described as passive. It proposes deals to achieve peace and it always accepts incoming deals. Because of this, it is exposed to the opponents' strategies and it does not exploit the weaknesses of adversaries. *Gunma*, on the other hand, is a greedy player. It only proposes deals that would benefit itself and does not take into consideration its allies. However, it is less selfish when accepting deals, given that it will accept deals that do not cause it to lose anything, even if it gains nothing from it.

3.4 Gap Analysis

The environments presented in this chapter focus on different challenges. Therefore, the features that each one of them offers are distinct, as shown in Table 3.1.

Table 3.1: Feature comparison between the environments introduced in Sections 3.2 and 3.3.

Environment	DRL	Multi-Agent	Negotiation
Atari	X		
Go	X		
OpenAI Gym	X		
Multiple Tank Defence	X	X	
OpenAI Five	X	X	
BANDANA		X	X

On the one hand, negotiation in a multi-agent system is relatively unexplored in current RL works. On the other hand, recent Diplomacy negotiation bots do not employ any RL mechanism, and the available environments do not enable easy integration with deep reinforcement learning frameworks in particular. With the current work, we are attempting to model an approach that utilizes deep reinforcement learning to play cooperative negotiation games, using Diplomacy as our testbed. More specifically, we intend to contribute to the scientific community by creating an environment that allows the study of negotiation strategies in a multi-agent system using reinforcement learning techniques.

Chapter 4

Negotiation in Diplomacy

This chapter contains the details and formalization of the problem we are trying to solve according to the notions given in Chapter 2. More specifically, we frame *Diplomacy* (Section 2.4.1) in game theory and explain how it serves as a test-bed for our approach. Furthermore, we formalize it as a Markov Decision Process (MDP) and expose our assumptions and constraints.

4.1 Diplomacy in Game Theory

We can classify Diplomacy under game theory [OR94]. It belongs to the following types:

- **deterministic**. No game event is random. Every outcome depends only on the actions taken by the players, therefore for a given state, the same set of actions will always lead to the same result.
- **non-cooperative**. There is no external authority to enforce cooperation. Players will only cooperate while it benefits them.
- **symmetric**. The payoffs for playing a particular strategy depend only on the other strategies employed, not on who is executing them. For example, no player loses by gaining control of a supply center. The objective is the same for every player.
- **zero-sum**. For a player to win control over a *supply center* another one has to lose it. However, if we only consider a subset of players, such as two allies, there are some situations that can be considered *non-zero-sum*. For example, when an alliance conquers supply centers, the aggregate gain for the players in that alliance is superior to zero. Still, for a subset of players to win, another subset of players must lose. Therefore the scenario which includes all players is *zero-sum*.
- **simultaneous**. The orders of the players are all revealed at the same time, so no player can take advantage from knowing what the other players will do.

- **perfect information**. A player knows all of its opponents' previous moves because the orders are public and made on the board.
- **incomplete information**. The players do not possess full information regarding the true "strength" of the other players, because the deals made between players may not be public.
- **discrete**. The number of moves and outcomes is finite. However, the tree containing all the possible combinations of moves is immense.

Our purpose is to build a program (a bot) capable of playing and winning Diplomacy. While the original Diplomacy game is non-cooperative, for this work we will abide by the rules of the Diplomacy League of ANAC, which forces players to be implemented using the BANDANA environment (Section 3.3.1). In BANDANA, the deals players make with each other are **binding**. In this case, the negotiation becomes **cooperative**, while the main objective of the game remains **non-cooperative**. Cooperative scenarios are a reduction of non-cooperative scenarios, so the problem should be simpler. The ANAC rules also state that the BANDANA player must use the tactical module of a bot called D-Brane [JS17]. Therefore, for the bot developed in this work, the focus will be solely on the negotiating module.

4.2 Problem Formalization

One useful way of formalizing the problem we are facing with Diplomacy is by describing it as a Markov decision process (MDP), introduced in Section 2.2.

First, we need to define the boundary between agent and environment. In Diplomacy, a player is capable of giving orders to its own units and making deals with other players. Everything else, such as the board disposition, orders of adversaries, and deals on which the player does not partake are part of the environment. However, because we will be focusing on the negotiating part of Diplomacy, our agent's actions are limited to proposing deals to other players. It will not be giving orders directly. That responsibility falls to the tactical module. Therefore, tactical orders given to units of our player will also be considered part of the environment to our negotiator.

Having defined the environment and the agent, we carry on to define *state* and *action*. A state should contain relevant information about the environment that the agent can analyze to decide the action to take properly. One important component of the state is the board information in a given turn. We can break down the board by some of its components and relations, such as: players and respective powers; provinces occupied by powers; supply centers and the province they belong to; units, their type and who they belong to; the orders given by each power; and others.

By formalizing a problem as a MDP, we are declaring that a state follows the Markov property. Therefore, the action is only influenced by the information of the current state.

The actions our agent can make are related to the negotiating phase of Diplomacy. Because we are structuring the agent according to the ANAC competition rules and the BANDANA framework, we will follow the negotiation protocol it requires. According to it, a deal may consist of any number of the following:

- *Order Commitment*, which represents a promise that a power will submit a certain order o during a certain turn σ and a year y, represented by the tuple:

$$oc = (y, \sigma, o)$$

- *Demilitarized Zone*, which 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 turn σ and year y, represented by the tuple:

$$dmz = (y, \sigma, A, B)$$

Our agents actions will solely consist of accepting, declining or offering deals, because the creation and execution of orders is already made by the strategic module of the bot.

The next step is defining the reward function. One important consideration when defining the reward function is that the reward signal communicates to the agent what we want it to achieve, not how we want it to be achieved. What we want our agent to achieve is victory. Victory implies having in possession at least 18 supply centers on any turn. An agent tries to maximize its reward signal, therefore if the reward signal increases as the number of controlled supply centers increments, the agent will look for actions that lead to the conquest and defense of supply centers. Given n_s, the number of supply centers controlled in a state s, a simple reward function would be the following:

$$R_a(s, s') = n_{s'} - n_s \tag{4.1}$$

The agent is punished if it loses any supply centers and rewarded if it wins any. It gives freedom of decision to the agent, that can, for example, sacrifice a supply center and later on conquer it back. Given this reward function, the problem then becomes maximizing the total reward in a game, which in turn encourages the agent to gain control of as many supply centers as it can. Once the agent gets 18 supply centers, the game ends and it achieves victory.

4.2.1 Diplomacy Negotiation 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 a 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 4.2 gives an upper bound on the possible number of orders n_{orders} for each unit, where P is the number of *Provinces* in the board. The number of valid orders may be less than n_{orders}, because an order will only be valid if the unit can move to the *Province* referred in that order. However, we do not save information about the adjacency of provinces in our state representation, as it increases even more its complexity. The BANDANA framework will examine invalid orders, such as moving a unit to a non-adjacent province, and will replace them with `hold` orders.

$$n_{orders} = 1 + 2P \tag{4.2}$$

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. Because a deal may contain any number of order commitments (*oc*) and demilitarized zones (*dmz*), and the year parameter (y) can go up to infinity, the action space of negotiation is infinite. 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 turns in a year 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 4.5, where n_{oc} is the number of possible order commitments (*oc*) and n_{dmz} is the number of possible demilitarized zones (*dmz*).

$$n_{oc} = y_{max} * H * (u_{own} + u_{op}) * n_{orders} \tag{4.3}$$

$$n_{dmz} = y_{max} * H * L * 2^P \tag{4.4}$$

$$n_{deals} = 2^{(n_{oc}+n_{dmz})} \tag{4.5}$$

For instance, if we consider the upper bound of the number of possible deals with one year of planning $2^{n_{deals}}$, raising the number of years to three increases the number of possible negotiations to $2^{3n_{deals}}$. This demonstrates the incredibly large branching factor of the negotiation space search tree.

Besides submitting deals, a player needs to decide whether it will accept or reject incoming proposals. Given that a player is subject to binding agreements, it is not possible to renounce an agreement in a turn after accepting it. Therefore, for each incoming deal, the action space is binary.

4.3 Assumptions and Premises

The following is the current list of assumptions for our work:

(a) All agreements are binding.

(b) Every players' goal is to win the game.

(c) Each agent may only control one player.

We are also basing our work on the premise that cooperation **is beneficial** in Diplomacy as proven by DeJonge et al. [dJBA+19]. According to this article, a coalition between two players is generally advantageous, but the amount of advantage of the alliance depends on which two Powers they represent. This premise is relevant because we will rely on it to evaluate our results in Chapter 6.

4.4 Constraints and Limits

One of the main problems of reinforcement learning is what Bellman called the *curse of dimensionality*. The bigger the search space, the more data is needed to make an informed decision. To compute samples for a game with the search tree of Diplomacy is challenging. It is even more difficult when hardware resources are not that powerful. For instance, one iteration of DeepMind's AlphaGo had 176 graphics processing units [SH17] behind its calculations. For this work, we only have access to a system that is as powerful as an high-end personal computer. Therefore, the time necessary for the agent training may be quite long.

Another factor that can affect our processing speed is the agent's software. There is a risk of the language or of the code itself slowing down execution. An algorithm that is fast may become slow with the wrong coding. Hence, the lack of program optimization will affect the final results.

Chapter 5

OpenAI Gym Diplomacy Environment

Having the features of OpenAI Gym (Section 3.2.3) and BANDANA (Section 3.3.1) in consideration, we propose creating an OpenAI Gym environment that enables Diplomacy agents to learn how to play the game using BANDANA as the game logic handler. Creating a Gym environment makes it easier to adapt already existing reinforcement learning agents that could learn to play this game. For instance, OpenAI maintains a repository containing the implementation of several DRL methods [DHK+17] which are compatible with Gym environments. Utilizing these agents 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, the resulting model can be recycled to create environments for similar problems.

In this chapter we describe the proposed OpenAI Gym Diplomacy environment, which we call `gym-diplomacy` [CCLC19] (article available in Appendix A).

5.1 `gym-diplomacy` architecture

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. The custom environment encapsulates an adapted implementation of a BANDANA player and the communication between all the necessary processes. The main purpose is separating the agent's implementation form the environment's implementation. The agent interacts with the environment by calling the functions defined in the OpenAI Gym's interface. The dashed black arrows and the solid blue arrows in the figure show the interactions between the components when the agent calls the `reset` and `step` functions, respectively. As the agent is independent of the environment, so is the reinforcement learning module that the agent uses, which enables reusing approaches applied in other Gym environments.

In this section, we break down each of the components of the environment, their functionality, and the technologies used for each one.

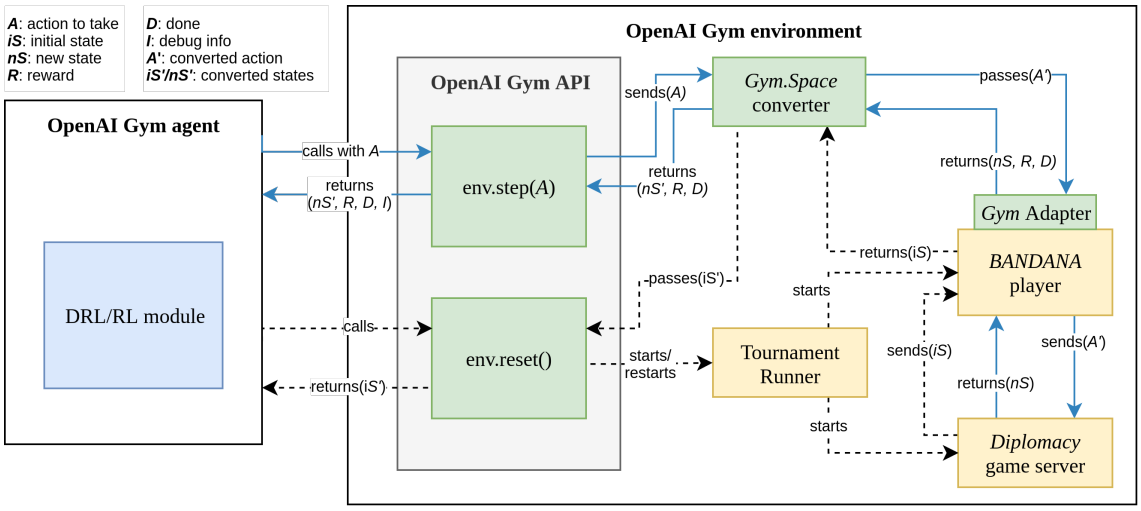

Figure 5.1: Conceptual model of the Open AI Gym Diplomacy environment and agent.

5.1.1 The BANDANA player: `OpenAINegotiator`

The first step to connecting OpenAI Gym and BANDANA is to create a bot to play a Diplomacy game in BANDANA. There are two types of bots in BANDANA: negotiation bots and tactical bots. For every phase of the game, a tactical bot must assign an order to every unit it controls. However, it is not allowed to settle agreements with other players. A negotiation bot also has the tactical burden, but it is allowed to negotiate with other players in the appropriate phases of the game. BANDANA provides a particular negotiation bot class which has the tactical module already implemented, using D-Brane [JS17] as a decision-maker. Therefore, we only need to worry about implementing negotiation methods.

Our negotiation bot, which corresponds to our BANDANA player, is called `OpenAINegotiator`. It extends the BANDANA class `ANACNegotiator`, which requires implementing the `negotiate` method. When the game server sends a message saying the negotiation phase is beginning, the bot handles that message by calling the `negotiate` method. Inside this method, the bot analyzes incoming proposals and sends its own. Because we want the decisions regarding proposals to be made by a deep reinforcement learning agent, we need to connect this method to that agent. To achieve this, we created the `OpenAIAdapter`.

5.1.2 The Gym adapter: `OpenAIAdapter`

We will need to connect the Gym's Python process to our `OpenAINegotiator`. To do so, we created the `OpenAIAdapter`, responsible for making the bridge between the two programs. Interprocess communication is, in itself, a complex problem.

Our first approach consisted of the traditional socket communication, where we would open sockets in both processes to allow two-way communication. However, it required a lot of attention

to exception handling when the programs crashed unexpectedly. To make implementation simpler, we chose to use gRPC[1] instead. One of the advantages of this framework is that it already implements the client-server interface. We only need to define the methods we wish to call from either side. Another advantage is the native compatibility with Protobuf[2]. Protobuf is extremely useful, as it allows us to define the structure of messages both for Python and Java using a single specification (a `.proto` file). Also, we can define gRPC methods in the same file. Given that the only thing we want from the DRL agent to generate an action from given a state/observation, we only need to define one method and the structure of the action and observation, which will depend on the environment model.

5.1.3 OpenAI Gym Spaces

In OpenAI Gym, an environment must define the `action space` and `observation space` fields, which correspond to the action and observation spaces, respectively, and 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), where o_i represents the player that owns province (0 if none) and sc_i is 0 if the province does not have a supply center or 1 otherwise. We use a `MultiDiscrete` space with $2n_p$ `Discrete` spaces, where n_p is the number of provinces. An observation for 75 provinces then becomes:

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

$$o_i \in [0..74]$$
$$sc_i \in [0..1]$$

Given the immense complexity of the negotiation action space, we reduced the scope of action of `gym-diplomacy` from the one introduced in Section 4.2.1. Instead of deciding over the whole action space, we limit the possible actions. We abstract the action space in different manners for each training scenario shown in Section 6.1.

One thing that is not considered part of the scope of the reinforcement learning agent's actions in any experiment is the act of accepting or rejecting incoming offers. Because Gym environments only have one observation space and one action space, analyzing proposals would require another environment. While connecting two environments is possible, there was not enough time to implement this interaction.

The alternative mechanism for acceptance and rejection utilizes the tactical D-Brane module to determine how accepting a deal will influence the game and is detailed in Algorithm 1. The D-Brane module provides a function `determineBestPlan` that allows us to determine how many

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

supply centers could be won or lost when a given set of agreements are imposed (lines 1-2). If accepting the agreement enables us to gain a supply center, we will accept the deal (lines 3-4). If it doesn't, we will only accept it if the proposing player is weaker than us (lines 5-10). We consider an opponent "weaker" if it controls at most two supply centers less than our player (line 7). We accept deals that do not present a direct benefit to us, because we may be able to increase the trust of other players without risking losing supply centers.

Algorithm 1: `accept_deal` implementation

globals: *tactic*: the D-Brane tactical module,
currentSC: the number of supply centers our player currently owns
input : *deal*: the incoming deal
output: *accept*: a Boolean value that is true if we wish to accept the deal, or false if we wish to reject it

```
1  newSC ← tactic.determineBestPlan(deal).getSC();
2  balanceSC ← currentSC - newSC;
3  if balanceSC > 0 then
4  |  return True
5  else
6  |  opponentSC ← deal.getPlayer().getSC();
7  |  if opponentSC ≤ currentSC - 2 then
8  |  |  return True
9  |  else
10 |  |  return False
11 |  end
12 end
```

5.1.4 Gym Converter

To exchange information between `gym-diplomacy`'s Python process and `OpenAINegotiator` Java process, we implemented functions that take a Protobuf message and convert it to a Python or Java variable and vice-versa.

We map the observation and the action specifications to Protobuf messages. Then, we generate Java and Python classes from the Protobuf file, which we integrate into the `gym-diplomacy` environment and the `OpenAINegotiator` BANDANA player. We configure the Gym process as the Remote Procedure Call (RPC) server and the BANDANA player as the client. The client is able to call the function `getAction()` that takes a single parameter of class `Observation` and returns an `Action`. However, we still need an extra step to get the action and observation Protobuf variables into variables that are useful to our Gym environment and player. In BANDANA, we convert the game state into an `Observation` variable. On the side of `gym-diplomacy`, we convert the received Protobuf `Observation` into a number array that belongs to the observation space. Then, we convert the action given by the agent back to a Protobuf `Action`. This action is returned to our `OpenAINegotiator`, which converts it to the respective game moves.

With the conversion of the necessary information and field definition concluded, we need only to implement the necessary functions of a Gym environment.

5.1.5 `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 any time it wants. However, in a board game such as Diplomacy, the players must wait for their turn to play.

To circumvent this issue, we have used two flags – `waitingForAction` and `waitingForObsToBeProcessed` – to indicate when the `step` function should proceed and when it should be blocked. These flags mimic the behaviour of a semaphore.

When the agent calls the `reset` function, depicted in Algorithm 2, it expects the initial state of the Diplomacy board in return. To obtain it, we start a BANDANA process and a gRPC server if they are not already running (lines 1-4). Initially, we set the observation to a null value (line 5), then we set the flags for the `step` function and wait for the initial observation (lines 6-10). After the game and the player (`OpenAINegotiator`) processes start, the first round of the game begins. When the first negotiation phase starts, the player will send a request for action, with the current game observation attached. The handler function, illustrated in Algorithm 4, takes the request, extracting information from it, and sets the relevant global variables (line 1). It then sets the semaphore flags and hangs (lines 2-6), waiting for the agent's action. The `reset` function is now allowed to continue and returns the initial observation to the agent (line 11, Alg. 2).

At this point, the Diplomacy player is still waiting to act. After getting the initial observation, the Gym agent is expected to call the `step` function, described in Algorithm 3, providing an action as the argument. When it does, the function sets the `action` global variable (line 4) and the `waitingForAction` flag (line 5), and the handler sends it as a response to the Diplomacy player request.

Now the `step` function is hanging, waiting for the observation that results from the action the agent took (lines 6-8). When the next negotiation phase begins, the observation is sent attached to an action request. Then everything is repeated, until the game ends.

The sequence diagram in Figure 5.2 presents this behaviour in the lifeline of the `gym-diplomacy environment` object, which executes the calls made by the system actor (in this case, the agent).

5.2 Development Tools and Coding

The entirety of this project is available in a public Git repository[3]. One of the purposes of having the repository public and with documentation is to make it possible for other people to contribute to it and to encourage the exploration of environments as complex as Diplomacy. As such, the repository contains the necessary tools for the community to contribute to this project.

[3]Repository URL: https://github.com/jazzchipc/gym-diplomacy

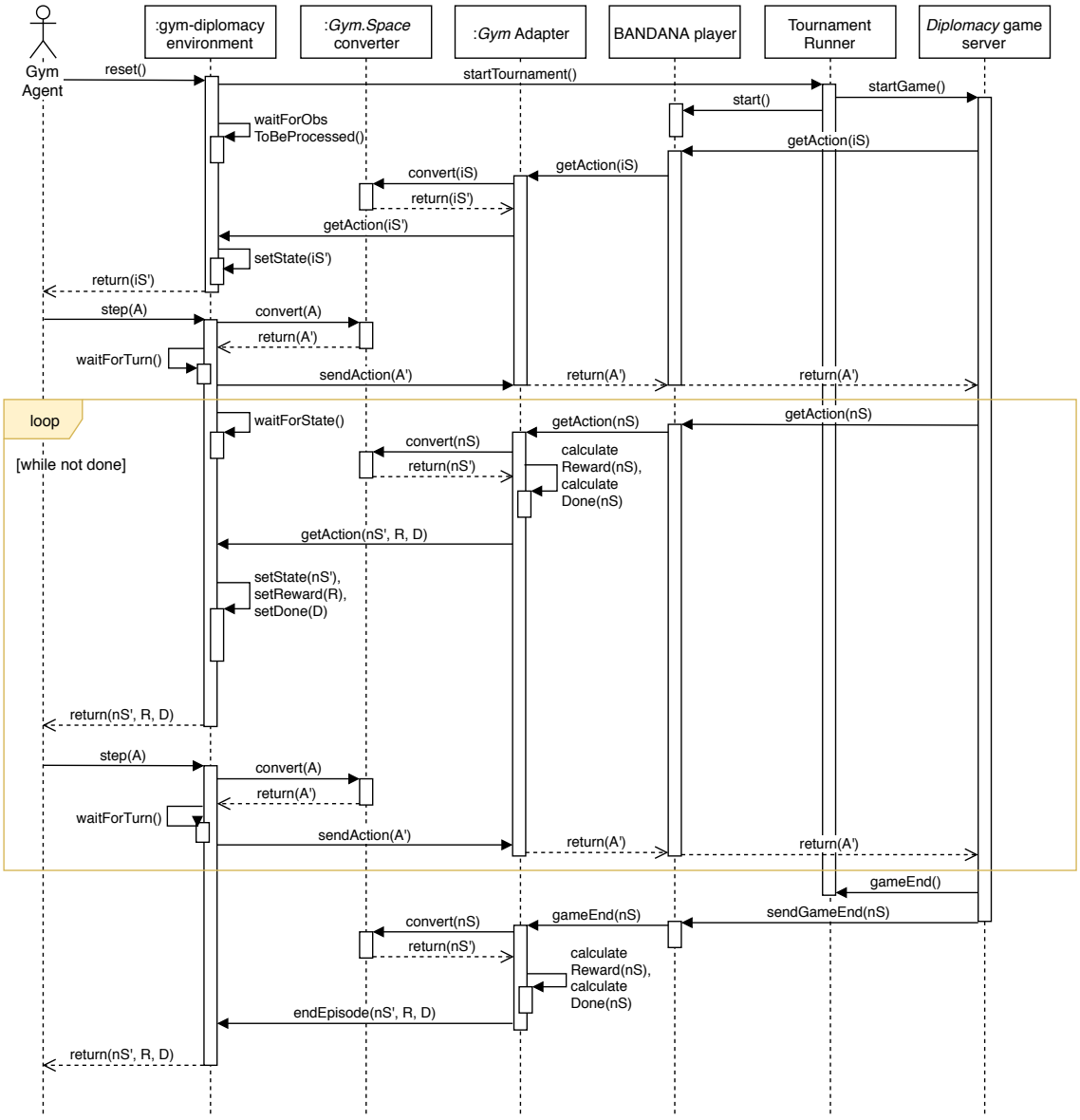

Figure 5.2: Sequential diagram of the Open AI Gym Diplomacy components, with the agent as the actor.

Algorithm 2: `reset` implementation

Result: If no BANDANA process exists, it'll start one. It'll also start the communication server and wait for the player to send the first observation.

globals: *bandanaProcess*: the process corresponding to the BANDANA game manager,
server: the gRPC server,
waitingForAction: a Boolean that determines whether the BANDANA player is waiting to be given an action or not,
waitingForObsToBeProcessed: a Boolean that determines whether we are waiting for the new game state observation has been processed or not,
observation: the current observation of the game state

output: *observation*: the observation corresponding to the initial game state

if *bandanaProcess* == *None* **then**
  bandanaProcess = initBandanaProcess();
if *server* == *None* **then**
  server = initiateGrpcServer();
observation ← None
waitingForAction ← False;
waitingForObsToBeProcessed ← True;
`// The observation is set by the action handler function.`
while *observation* == *None* **do**
  sleep(1);
end
return *observation*

Algorithm 3: `step` implementation

Result: If the BANDANA player is asked to negotiate, we determine which action the player should take, based on the current observation. After the negotiation phase, we get the new game observation, the reward of the action and whether the game has ended or not.

globals: *action*: the global variable holding the action to take in the environment,
waitingForAction, *waitingForObsToBeProcessed*

input : *new_action*: an action space element corresponding to the new action to take

output: *observation*: an observation space element corresponding to the new game state,
reward: the float value of the reward of the action,
done: a Boolean, determining if the game as ended or not

while *not waitingForAction* **do**
  sleep(1);
end
action ← new_action
waitingForAction ← False;
while *waitingForObsToBeProcessed* **do**
  sleep(1);
end
return *observation, reward, done*

Algorithm 4: `action_request_handler` implementation

Result: When the Diplomacy player sends a request to get an action, the handler sets the flags for the reset and step functions appropriately, returning the action defined by the agent as a response.
globals: *action*, *observation*, *reward*, *done*, *waitingForAction*, *waitingForObsToBeProcessed*
input : *request*: the request of the BANDANA player
output: *response*: the response to the request

```
observation, reward, done ← getInfoFromRequest(request);
// The observation has already been processed
waitingForObsToBeProcessed ← False;
// We are now waiting for the action
waitingForAction ← True;
while waitingForAction do
    sleep(1);
end
// The global variable action has been set by the step function.
response ← generateResponseFromAction(action);
return response
```

To manage the Java dependencies of BANDANA and the `OpenAINegotiator`, we used Maven[4]. Maven is also useful to bundle the necessary Java classes into a JAR file that is used to start a Diplomacy tournament. The agents employed were slightly modified, mostly because to customize logging and model saving/loading. All the agents are implemented using Python 3, and we have used Pipenv[5] to manage their dependencies.

The environment's repository has continuous integration configured. When a push is made to the main branch, unit tests run automatically and a new Docker image containing the agents and the environment is created and pushed to a registry, facilitating the deployment to any machine.

[4] https://maven.apache.org
[5] https://pipenv.readthedocs.io

Chapter 6

Experimental Evaluation

This chapter details the experiment scenarios and respective evaluation. First we specify the scenarios where we trained our agents and how they improved over time. Then, we detail the testing conditions, and discuss the results them.

6.1 Training Scenarios

The Gym framework is oriented to prepare a single agent. Therefore we could only control one Diplomacy player during training. The game and board configuration used was the standard one, with 75 provinces, 34 supply centers, and six opponents. During the learning phase, it was essential for the agent to negotiate with other players. Therefore, we chose to put our agent against six negotiating instances of the D-Brane bot with negotiation module [JS17]. The tactical modules are precisely the same for every player (the D-Brane tactical module), so what will be evaluated is the performance of the negotiation component. We limited the maximum number of years in the game to 20, which results in a limit of 40 turns. After 40 turns, if there is no winner, the game ends in a draw.

In BANDANA, the processes corresponding to the different players must communicate with each other using the game server as a messaging intermediary. When a power wishes to propose a deal to another player, the processes create at least four different messages. After an agreement is reached, another process must validate the deal. All this exchange of information needs time to be executed. Therefore, BANDANA has set a maximum time of three seconds for the negotiation phase (the Diplomatic Phase). For our training scenarios, we reduced the maximum time to just one second, to speed up training. Reducing the time below one second resulted in messages not being sent and received appropriately, so we could not further accelerate training.

The observation space of the agent is the one introduced in Section 5.1.3 for all experiments. By keeping the observation space for agents with different action spaces, we can compare the impact of the action scope in performance more accurately.

Each training scenario has either a different environment or agent model. All training was run on a Google Cloud Engine instance, with 4 Intel Haswell virtual CPUs, no GPU, and 6 GB of RAM, inside a Docker container.

6.1.1 Scenario 1 - Learning Valid Deals

The first experimental scenario was an effort to understand if the environment was well set-up and if an agent could effectively learn within the framework. To keep the learning process simple, instead of trying to propose deals with the objective of winning the game, we set out to solve a smaller predicament: whether the agent could learn to propose valid deals or not.

The agent we employed in this scenario uses ACKTR (introduced in Section 2.3.2.1) as a learning method. We made use of the implementation provided by OpenAI Baselines[1] with all parameter values set to the default values.

6.1.1.1 Action Space

The action space was significantly reduced from the one introduced in Section 4.2.1. 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 turn of the game and contains just two `move` order: one for a player's own unit and the other for an opponent's unit. Using the `Space` sub-classes (as introduced in Section 5.1.3), we 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:

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

$$sp_{own}, dp_{own}, sp_{op}, dp_{op} \in [0..74]$$
$$op \in [0..6]$$

6.1.1.2 Reward Function

Since negotiation does not directly affect the number of conquered supply centers, using only supply centers information 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. This reward function is explicit in Equation 6.1, where r represents a constant. Specifically, 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. For example, if a unit has been moved from the position it was in when the deal was created, then the order regarding that unit will be invalid. While this reward function does not directly lead to victory,

[1] Available at: https://github.com/openai/baselines/tree/master/baselines/acktr

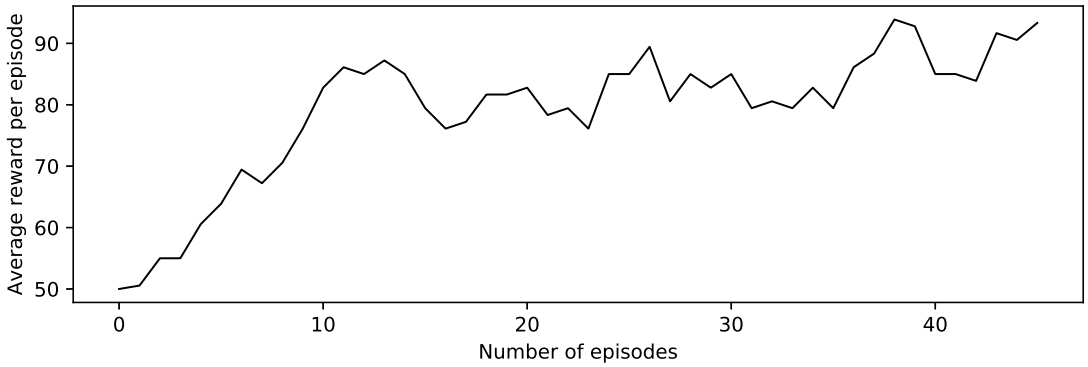

Figure 6.1: Average rewards per episode (game) of an agent learning with ACKTR in the negotiation environment from scratch (average of 3 executions over 46 episodes). The values have been smoothed using a window size of 3 episodes. Each game has a variable number of steps. A valid deal gives a positive reward $+5$, each invalid deal gives -5.

it helps the agent become better at playing the game. Because there is a time limit during the negotiation phase, it is important not to waste time by proposing invalid deals.

$$R_a(s,s') = \begin{cases} r, & \text{if deal is valid} \\ -r, & \text{otherwise} \end{cases} \tag{6.1}$$

6.1.1.3 Results

Figure 6.1 contains the average results of three different executions. In each execution, the agent is learning from scratch, so that we can show that the learning rate Because each game may have a different number of turns, instead of demonstrating the episode reward over the number of steps, we show the average reward over the number of episodes, with $r = 5$.

Because the Diplomatic Phase takes a whole second and is not the only phase in a turn, 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.

6.1.2 Scenario 2 - Abstracting Action Space with Forethought

The next objective we set out for the agent was to win the game. However, the vast dimension of the observation and action space does not allow us to test and evaluate models, as training requires a tremendous amount of time. Because we believe reducing the observation space would obscure relevant information from the agent, we have chosen to reduce the action space.

We reviewed agents that were submitted to the ANAC Diplomacy competition to understand what kind of agreements they proposed (Section 3.3.2). We noticed bots relied on either attacking with the support of another power or on trying to make peace so that nobody would attack them. With this in mind, we defined four different types of agreements:

- "mutual unit defense": choose a unit we control and decide to *hold* it, i.e., not move the units anywhere. Then choose a random adjacent province and propose a mutual *support hold* order to the owner of that province, where the owner also *holds* their units and we support each other's orders. A "mutual unit defense" agreement is represented by a tuple:

$$mud : (provinceIndex, year, turn)$$

  where *provinceIndex* is the index of a province our player controls and *year*,*turn* is the year and turn of the game we want the deal to take effect.

- "supply centers mutual non-aggression": choose another power besides us and propose a *demilitarized zone* agreement for all the provinces with supply centers that belong to us and to that power. This agreement is represented by:

$$scmn : (powerIndex, year, turn)$$

  where *powerIndex* is the index of the opponent we wish to broker peace with.

- "supported attack": choose a province we control and randomly select a province to attack while asking an adjacent province that is controlled by a different power to support our attack. This agreement is represented by:

$$sa : (provinceIndex, year, turn)$$

  The deal is proposed to the power that controls the province specified by *provinceIndex*.

- "support attack": choose a province we control and randomly propose to a neighbour power supporting an attack on an adjacent province that is controlled by a different power. This agreement is represented by

$$sat : (provinceIndex, year, turn)$$

On a given turn, we allow the negotiating player to propose up to one agreement of each type. Therefore, associated with each agreement, we have a binary value determining if that deal should be proposed or not.

The agent we employed in this scenario uses Proximal Policy Optimization (PPO) (introduced in Section 2.3.2.2) as a learning method. We made use of the PPO implementation[2] provided by Stable Baselines [HRE+18] with all parameter values set to the default values.

[2] Available at: https://github.com/hill-a/stable-baselines/tree/master/stable_baselines/ppo2

6.1.2.1 Action Space

Based on the abstract actions that were specified, we represent the negotiation action space with a `MultiDiscrete` space consisting of nine `Discrete` spaces. For each type of action, we have a binary value e associated, which determines if the respective type of action should be executed. For actions of type mud, sa, sat, we have a *provinceIndex* (pr) associated and for a *scmn* we need a *powerIndex* (po). We also need to define which year and turn we want our deal to take effect. Given that a year has two turns, we can define year and turn with a single integer. For example, if we wish to make a deal two years from the current one we define the number of *turnsAhead* (t) as $t = 2 \times 2 = 4$. In this experiment, the maximum number of *turnsAhead* is 19. Therefore, an action is characterized by:

$$\text{action: } (e_{mud}, pr_{mud}, e_{scmn}, po_{scmn}, e_{sa}, pr_{sa}, e_{sat}, pr_{sat}, t)$$
$$e \in [0..1]$$
$$pr \in [0..17]$$
$$po \in [0..6]$$
$$t \in [0..19]$$

The values of the *provinceIndex* variables are **clipped** to match the current state of the game. For example, if a "mutual unit defense" action has $provinceIndex = 5$ but the player only controls three provinces, then the value will be clipped at 2 (the first index has value 0).

6.1.2.2 Reward Function

The objective of the agent is to win the game and, to achieve this, it is required to conquer 18 supply centers. 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. However, this means that the agent's network is only updated with inputs different than zero after the end of an episode. To foster the learning process, we define the reward function to consider the supply centers that the agent conquers with its actions. We also consider whether the deals the player proposes in a given turn are accepted or not. Therefore 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 6.2, where r_{sc} is a constant defining the reward for conquering one supply center, $SC(a)$ is the number of supply centers controlled in state a, r_d is a constant defining the reward for a proposed deal being accepted by another power and d is the number of deals accepted by other powers, which we proposed in the last turn.

$$R_a(s,s') = r_{sc} * (SC(s') - SC(s)) + r_d * d \tag{6.2}$$

We include the number of accepted deals in the reward function because the number of conquered supply centers is directly related to the tactical decisions and, therefore, would not correctly encourage the agent to negotiate. With the proposed reward function we want the agent to be good at negotiating (making other players accept its deals), while conquering supply centers.

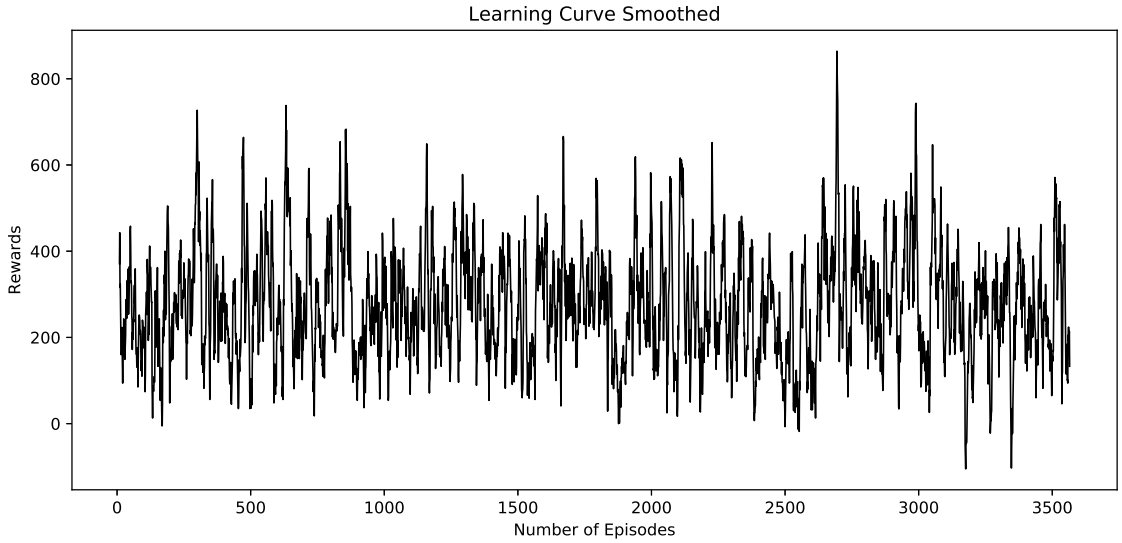

Figure 6.2: Rewards per episode (game) of an agent learning with PPO in scenario 2. The values have been smoothed using a window size of 10 episodes.

6.1.2.3 Results

In this experiment, we set the value of r_{sc} to 100 and the value of r_d to 10. The results of the learning process are displayed in Figure 6.2. We can verify that the learning process was not effective in increasing the reward function. We attribute the lack of learning to the insufficient number of time steps used to train the agent. Even with an algorithm as efficient as PPO, the policy network was not able to converge. The main reason for this is the time that it takes to run a single time step, which averages between one and two seconds. The training of the agent for this scenario took a bit more than 110 hours.

6.1.3 Scenario 3 - Abstracting Action Space without Forethought

Planning ahead to make a deal adds a high level of complexity to the action space. For this scenario, we decided to simplify Scenario 2. Instead of planning ahead, the deals take effect in the current turn. Therefore, the only alteration made to the action space is removing the *turnsAhead* variable. The reward function remains the same.

6.1.3.1 Results

Similarly to the results obtained in scenario 2, the training curve (Figure 6.3) does not indicate any degree of learning. The agent had almost 70 hours of training, which is less than the one in scenario 2, however there is no sign of convergence.

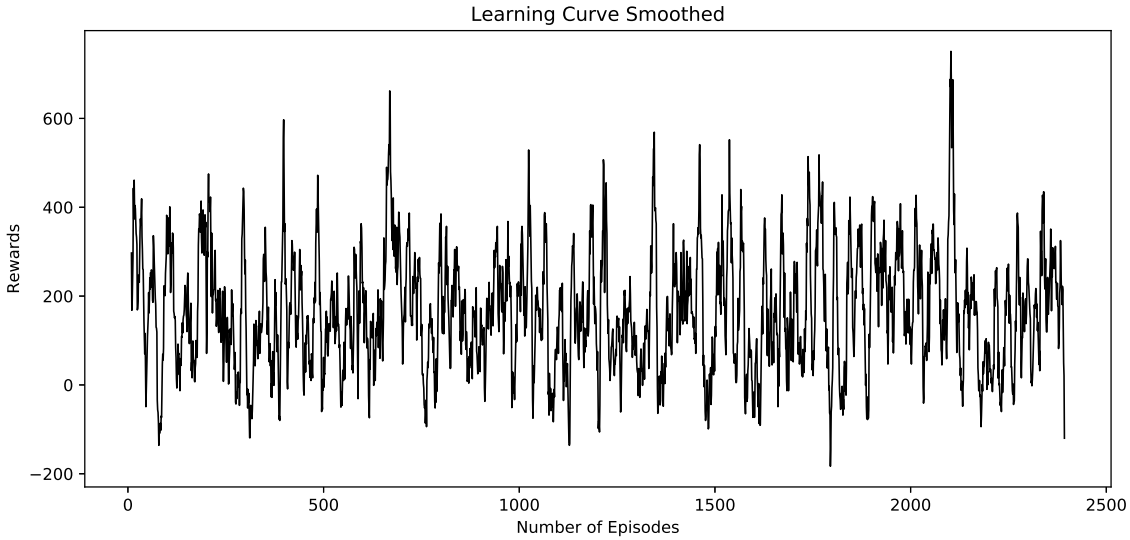

Figure 6.3: Rewards per episode (game) of an agent learning with PPO in scenario 3. The values have been smoothed using a window size of 10 episodes.

6.2 Testing Scenarios

We will not be testing the agent trained in Scenario 1. As the purpose of this agent was to verify that the environment allowed a reinforcement learning agent to train on a small action scope, it does not contribute to the final objective of having a winning player in Diplomacy.

We took both models that resulted from the training in Scenario 2 and 3 and tested them in the same conditions they trained in. Namely, we put them up against six instances of the D-Brane bot with a negotiating module, in a standard map with a limit of 40 turns. We ran 80 games for each agent.

6.2.1 Assessment System

To assess the results of each agent model, we will evaluate the results using statistical hypothesis testing. More specifically, we will make a Student's t-test to decide whether our results are statistically significant or not. For a large enough number of games, we expect the average of supply centers conquered by a player ($\bar{x}$) to be $\frac{34}{7} \approx 4.86$, as there are 34 provinces and 7 players in a game. Therefore, our null hypothesis (H_0) will state that a player's average is equal to or less than 4.86, and our alternative hypothesis (H_1) will state that a player's average is greater than 4.86. The significance value (α) we propose is equal to 0.05. The hypothesis test (one-tailed test) is formalized as:

$$H_0\text{: } \bar{x} \leq 4.86$$
$$H_1\text{: } \bar{x} > 4.86$$
$$\text{with } \alpha = 0.05$$

If we can reject the null hypothesis, then we can conclude whether or not our agent performs better than the other agents in the game. Given our premise that cooperation is beneficial to the performance of an agent (established in Section 4.3), we can also conclude whether our model is well suited for negotiation scenarios or not. This hypothesis test is not focused only on victories, as winning the game generally takes a great number of turns and in our testing scenarios, we are limiting the maximum number of turns.

6.2.2 Results

The results of both models (with and without forethought) are displayed in Table 6.1. The average number of conquered supply centers is lower than the average of 4.86 that we were aiming at. We can confirm that the performance is inferior to what we expected because the p-value for both approaches is superior to the significance level we set for our hypothesis test ($\alpha = 0.05$). Because of this, we cannot reject the null hypothesis. Therefore, we can assert that none of the approaches resulted in a player capable of surpassing its competitors. However, the agent trained to negotiate planning with forethought was able to do much better than the one with no forethought. While it seems that this is a positive result, we will show that this derives from the power with which the agent played.

Table 6.1: Victories and supply centers conquered by the the models trained in scenario 2 (forethought) and scenario 3 (no forethought) in 80 games played against six D-Brane bots with negotiation.

	Forethought	**No Forethought**
Games played	80	80
Victories	2	2
Supply centers	379	288
Average of supply centers	4.74	3.6
Standard deviation	4.13	4.18
t-statistic (hypothesis test)	-0.27	-2.70
p-value (one tail)	0.605	0.996

6.2.3 Errors and Deviations

The Power that a player controls is decided randomly at the beginning of the game. Different Powers have different strengths, related mostly to the position of their initial supply centers on the board. Therefore, there may exist some deviation in the final results due to the different Powers our agent plays with. For instance, Table 6.2 reveals that when the agent plays with Russia, the average of supply centers it conquers in each game is quite superior in comparison to other powers. However, playing as Italy often results in poor performance. This difference in outcome is probably due to the fact that Russia starts with one more supply center than other powers, as mentioned in Section 2.4.1. Therefore, the difference in the average of conquered

supply centers between the agents trained with and without forethought can be attributed to the agent with forethought playing more times with Russia.

Also, the fact that the set of actions we defined for the agent involve some degree of randomness supplies the environment with a stochastic nature. That is, for a given state, the same action may not lead to the same outcome, which hinders the learning process and may lead to deviations when testing the agents. However, the alternative to this policy would be expanding the action space significantly, so that the agent may be fully in charge of the impact of its actions. We considered the former approach more beneficial than the latter, although we were not able to verify if it really is more beneficial.

Table 6.2: Number of games played as each power and average number of supply centers (SCs) conquered per game for each power (80 games for each agent).

	Forethought		**No Forethought**	
	Games played	Average SCs	Games played	Average SCs
Austria (AUS)	10	2.75	15	1.07
England (ENG)	11	4.64	12	4.25
France (FRA)	10	4.30	12	4.92
Germany (GER)	12	2.42	10	3.2
Italy (ITA)	11	1.78	15	0.73
Russia (RUS)	14	8.64	8	8.63
Turkey (TUR)	12	6.50	8	6.25

6.2.4 Discussion

The system performed poorly against the D-Brane negotiation bot both in scenario 2 and 3. From the training data, we can visualize the learning process was extremely slow. We attribute this to the vast action and observation space and lack of computational power. A possible solution to this problem would be distributing the learning process by more than one machine, effectively multiplying the learning capacity and reducing the time needed for training. Adding a single machine would cut in half the time required for the same amount of training.

Still, we proved that the `gym-diplomacy` environment is an effective way of using existing reinforcement learning implementations to learn how to play Diplomacy, as the results of the first scenario indicate. While the developed model was not very strong compared to existing alternatives, we have successfully created the test-bed for other models to be tested and improved. We have simplified the process of building a BANDANA reinforcement learning player, while making the bot compatible with most current Python reinforcement learning frameworks.

Chapter 7

Conclusions and Future Work

In this chapter we summarize the work and its results, as well as the answers to the research questions proposed in Chapter 1. We also contemplate improvements that can be applied in the future.

7.1 Conclusions

In Chapter 1, we exposed the motivations and objectives of this work. These goals included understanding whether a reinforcement learning approach was suitable for negotiation scenarios with an immense search space and, if so, how should it be structured. In Chapter 2, we showed works where applying reinforcement learning to adversarial games generated positive results. Deep reinforcement learning methods, in particular, have exceptional performance in situations where an agent has an extensive scope of action. Recent deep reinforcement learning methods, such as Deep Q-Network and policy gradient methods, have led programs to learn how to play a very complex game with performance similar to a human, or even higher. With this in mind, we set out to develop a reinforcement learning agent capable of acting in multi-agent games that require negotiation. One suitable example of such a game is Diplomacy, where occasionally cooperating with other opponents is necessary to achieve victory.

Diplomacy revealed itself as a fit candidate for studying negotiation between agents. However, the existing Diplomacy environments were not created with reinforcement learning agents in mind. To solve this, we adopted the OpenAI Gym interface and used it as a wrapper to the BANDANA environment, which contains all the logic necessary for an agent to play Diplomacy. The resulting environment, which we called `gym-diplomacy`, enables reinforcement learning agents to negotiate in a Diplomacy game. Agents designed for other Gym environments or created from scratch can connect to the environment straightforwardly.

The results of our experiments lead us to believe that reinforcement learning is, indeed, an appropriate method to allow a software program to learn how to negotiate in large search spaces.

The performance that DRL methods, in particular, have achieved in games where the search space is immense is and indicative of their potential. As the interface between Gym environments is standard, we were able to utilize already implemented agents with state-of-the-art deep reinforcement learning methods, such as Proximal Policy Optimization (PPO). We tested agents in slightly different scenarios and examined their performance. The outcomes show that the extensive observation and action space of negotiation in Diplomacy makes it very difficult for an agent to learn how to negotiate. We compared our agent against other Diplomacy bots capable of negotiating and concluded that it did not perform better than them. However, we believe the limitation that caused this underwhelming result is related to the little amount of useful computational power that was used. While the environment we developed allows reusing agents that have proven their performance in other Gym environments, it lacks the feature of distributed learning, that is, the trait of being able to use different machines to accelerate the training process. Because Diplomacy is such a complex game, training agents on a single computer prevented the learning process from stabilizing. Therefore, we reckon that a reinforcement learning model for negotiation in Diplomacy is only appropriate if the environment where it acts enables its learning potential with suitable processing capability.

As we were not able to achieve trustworthy performance in our environment, we are not able to understand how to model a reinforcement learning agent to succeed in a game with a mix of cooperation and competition. Each scenario we used to train our agents had the purpose of understanding what specific part of negotiation could lead to a great outcome in situations where competition and cooperation are necessary. We used different action spaces and reward functions, but were not able to pinpoint what influenced the results the most, as the agent never stabilized during the training process.

However, with our work, we hope to contribute to the spread of enthusiasm regarding employing DRL techniques in negotiation games. By making the `gym-diplomacy` environment available to the scientific community, we wish to facilitate the creation of works related to negotiation in Diplomacy. Hopefully, new approaches can be more successful in this task.

7.2 Future Work

One of the main flaws of our environment is the time it takes for the agent to take a single step. Because different processes must communicate between each other to negotiate, it is not feasible to reduce drastically the time permitted for the negotiation phases. However, there is room for optimizing the communication between the players and the game server, which would allow a faster negotiation phase. Another step towards making the learning process quicker would be allowing one agent to learn from different parallel environments. With the current configuration, the learning process is restricted to a single machine and its resources. By enabling distributed learning, we could use more resources, speeding up the training.

The OpenAI Gym framework is oriented to environments where it is only possible to control one agent. Learning through self-play has been utilized by some researchers in order to allow and

agent to learn quicker. As Diplomacy is a seven-player game, self-play would add a great value to the learning process. Therefore, allowing a single agent to control more than one player would also be a great improvement.

Connecting BANDANA with the Gym process adds an intricate layer to the final environment. It is quite hard to maintain two different code bases, as well as the mechanism that connects them. From a development point-of-view, a nice upgrade would be having the whole environment written in Python. The game server, provided by Parlance, is already a Python application, therefore the two components that would need to be translated are the player itself and the negotiation server, which BANDANA provides.

These and any other improvements that contribute to the acceleration of the training speed are extremely important to enable fast prototyping of different approaches to the negotiation in Diplomacy. Consequently, a reliable Diplomacy environment can lead to the development of agents capable of acting in complex situations that demand negotiation and compromise. If a reinforcement learning agent succeeds in negotiation in Diplomacy, it demonstrates that this paradigm can be used to model scenarios analogous to real-life and create systems capable of extremely complex decision-making.

References

[Add01] Rob Addison. My original standard map with a little color added. http://www.xcelco.on.ca/~ravgames/dipmaps/standard2_ra.gif, Nov 2001.

[BCP+16] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[Cal00] Allan B. Calhamer. *The Rules of Diplomacy*. Avalon Hill, 4 edition, 2000.

[CCLC19] Diogo Cruz, José Aleixo Cruz, and Henrique Lopes Cardoso. Reinforcement learning in multi-agent games: Openai gym diplomacy environment. In Paulo Moura Oliveira, Paulo Novais, and Luís Paulo Reis, editors, *Progress in Artificial Intelligence: 19th EPIA Conference on Artificial Intelligence, EPIA 2019*. Springer, 2019.

[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. https://github.com/openai/baselines, 2017.

[dJBA+19] Dave de Jonge, Tim Baarslag, Reyhan Aydoğan, Catholijn Jonker, Katsuhide Fujita, and Takayuki Ito. The challenge of negotiation in the game of diplomacy. In Marin Lujak, editor, *Agreement Technologies 2018, Revised Selected Papers*, pages 100–114, Cham, 2019. Springer International Publishing.

[DJZ17] Dave De Jonge and Dongmo Zhang. Automated negotiations for general game playing. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, pages 371–379. International Foundation for Autonomous Agents and Multiagent Systems, 2017.

[Dro93] Alexis Drogoul. When ants play chess (or can strategies emerge from tactical behaviours?). In *European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, pages 11–27. Springer, 1993.

[FLCR15] André Ferreira, Henrique Lopes Cardoso, and Luís Paulo Reis. Strategic negotiation and trust in diplomacy–the dipblue approach. In *Transactions on Computational Collective Intelligence XX*, pages 179–200. Springer, 2015.

[FSN] Angela Fabregue, Alejandro Serrano, and David Navarro. Dipgame.

[GGR88] Michael R Genesereth, Matthew L Ginsberg, and Jeffrey S Rosenschein. Cooperation without communication. In *Readings in distributed artificial Intelligence*, pages 220–226. Elsevier, 1988.

[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.

[JS17] Dave de Jonge and Carles Sierra. D-Brane: a diplomacy playing agent for automated negotiations research. *Applied Intelligence*, 47(1):158–177, 2017.

[KL51] Solomon Kullback and Richard A Leibler. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86, 1951.

[Li17] Yuxi Li. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*, 2017.

[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*, pages 50–56. ACM, 2016.

[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. *CoRR*, abs/1602.01783, 2016.

[MC68] Donald Michie and Roger A Chambers. Boxes: An experiment in adaptive control. *Machine intelligence*, 2(2):137–152, 1968.

[Mel01] Francisco S Melo. Convergence of q-learning: A simple proof. *Institute Of Systems and Robotics, Tech. Rep*, pages 1–4, 2001.

[MG15] James Martens and Roger B. Grosse. Optimizing neural networks with kronecker-factored approximate curvature. *CoRR*, abs/1503.05671, 2015.

[Mic61] Donald Michie. Trial and error. *Science Survey, Part*, 2:129–145, 1961.

[MKS+13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[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, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[MLC17] João Marinheiro and Henrique Lopes Cardoso. A generic agent architecture for cooperative multi-agent games. In *ICAART (1)*, pages 107–118, 2017.

[N+50] John F Nash et al. Equilibrium points in n-person games. *Proceedings of the national academy of sciences*, 36(1):48–49, 1950.

[Nas51] John Nash. Non-cooperative games. *Annals of mathematics*, pages 286–295, 1951.

[NNN18] Thanh Nguyen, Ngoc Duy Nguyen, and Saeid Nahavandi. Multi-agent deep reinforcement learning with human strategies. *arXiv preprint arXiv:1806.04562*, 2018.

[Ope18] OpenAI. Openai five. https://blog.openai.com/openai-five/, 2018.

[OR94] Martin J Osborne and Ariel Rubinstein. *A Course in Game Theory*. MIT press, 1994.

[RK13] Reuven Y Rubinstein and Dirk P Kroese. *The cross-entropy method: a unified approach to combinatorial optimization, Monte-Carlo simulation and machine learning*. Springer Science & Business Media, 2013.

[RNW07] Andrew Rose, David Norman, and Hamish Williams. Daide: Diplomacy artificial intelligence development environment, 2007.

[RNW09] Daniel Loeb Andrew Rose, David Norman, and Eric Wald. Parlance, 2009.

[SB18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2 edition, 2018.

[SH17] David Silver and Demis Hassabis. Alphago zero: Learning from scratch. https://deepmind.com/blog/alphago-zero-learning-scratch/, Oct 2017.

[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, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

[SLM+15] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015.

[SSS+17] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[SV00] Peter Stone and Manuela Veloso. Multiagent systems: A survey from a machine learning perspective. *Autonomous Robots*, 8(3):345–383, 2000.

[SWD+17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[Tes89] Gerald Tesauro. Neurogammon wins computer olympiad. *Neural Computation*, 1(3):321–323, 1989.

[Tes94] Gerald Tesauro. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural computation*, 6(2):215–219, 1994.

[WCW+08] Adam Webb, Jason Chin, Thomas Wilkins, John Payce, and Vincent Dedoyard. Automated negotiation in the game of diplomacy. *Master's thesis, January*, 2008.

[WD92] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[WML+17] Yuhuai Wu, Elman Mansimov, Shun Liao, Roger B. Grosse, and Jimmy Ba. Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation. *CoRR*, abs/1708.05144, 2017.

[Yoo19] Chris Yoon. Understanding actor critic methods and a2c. https://towardsdatascience.com/understanding-actor-critic-methods-931b97b6df3f, 2019.

REFERENCES

Appendix A

Reinforcement Learning in Multi-Agent Games: OpenAI Gym Diplomacy Environment - EPIA 2019

Reinforcement Learning in Multi-Agent Games: Open AI 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.

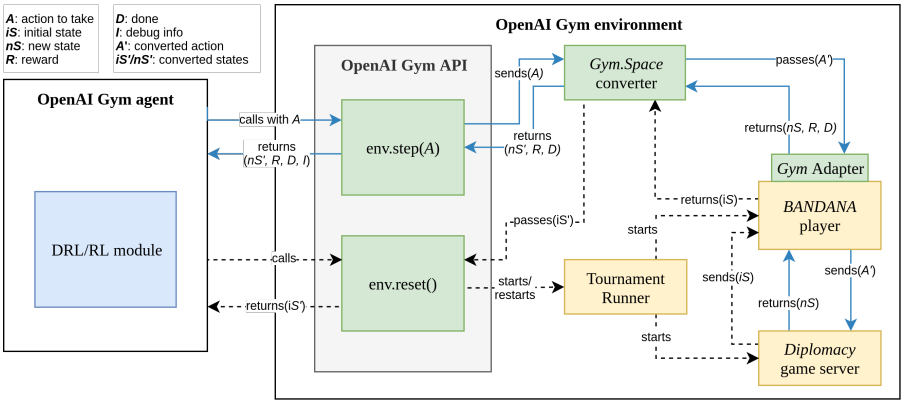

Fig. 1. Conceptual model of the Open AI Gym Diplomacy environment and agent. The solid and dashed arrows represent the interactions between the components when the agent calls the `step` and `reset` functions, respectively.

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.

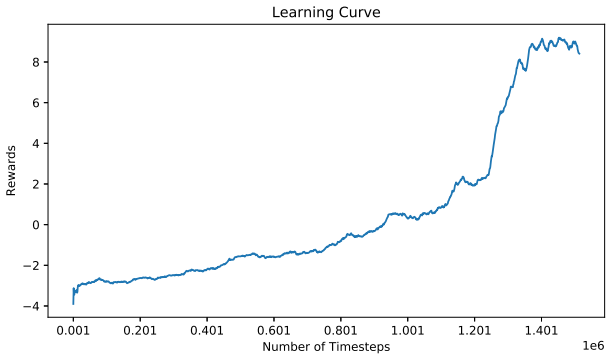

Fig. 2. Rewards per episode of a PPO agent in the 'small' board. A positive reward indicates that the agent was not eliminated from the game. A reward is higher than 10 when the agent has won the game.

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.

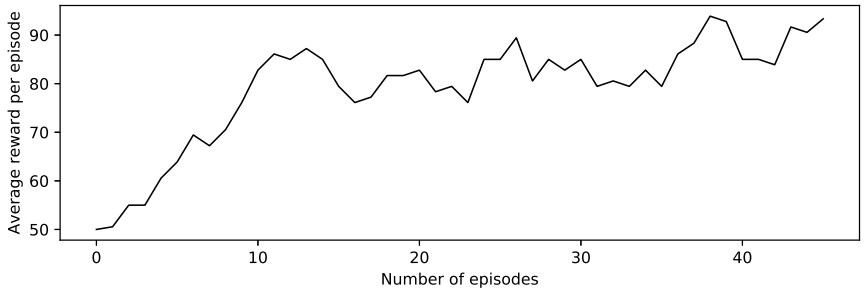

Fig. 3. Average rewards per episode (game) of an agent learning from scratch with ACKTR in the negotiation environment (3 executions over 46 episodes). The values have been smoothed using a window of size 3. Each game has a variable number of steps. A valid deal gets a positive reward +5, each invalid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