UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR
DE INGENIEROS DE TELECOMUNICACIÓN

GRADO EN INGENIERÍA DE TECNOLOGÍAS Y SERVICIOS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

DESIGN AND DEVELOPMENT OF A REINFORCEMENT LEARNING-BASED INTELLIGENT AGENT FOR SOLVING TEXT GAMES USING TEXTWORLD

BRUNO GONZÁLEZ LÓPEZ

JUNIO 2020

TRABAJO DE FIN DE GRADO

Título:	Design and Development of a Reinforcement Learning-based Intelligent Agent for Solving Text Games using TextWorld
Título (inglés):	Design and Development of a Reinforcement Learning-based Intelligent Agent for Solving Text Games using TextWorld
Autor:	Bruno González López
Tutor:	Carlos Ángel Iglesias Fernández
Departamento:	Departamento de Ingeniería de Sistemas Telemáticos

MIEMBROS DEL TRIBUNAL CALIFICADOR

Presidente:	—
Vocal:	—
Secretario:	—
Suplente:	—

FECHA DE LECTURA:

CALIFICACIÓN:

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE
INGENIEROS DE TELECOMUNICACIÓN

Departamento de Ingeniería de Sistemas Telemáticos
Grupo de Sistemas Inteligentes

TRABAJO FIN DE GRADO

DESIGN AND DEVELOPMENT OF A REINFORCEMENT LEARNING-BASED INTELLIGENT AGENT FOR SOLVING TEXT GAMES USING TEXTWORLD

Bruno González López

Junio 2020

Resumen

El objetivo de este proyecto es entrenar dos agentes inteligentes para que sean capaces de resolver juegos de texto, en concreto, un juego simple inspirado en la ETSIT. Además se compararán métricas para medir su rendimiento en relación a un agente que toma decisiones de forma aleatoria.

Los juegos basados en texto (muy populares en la década de los 80s) son simulaciones complejas e interactivas en las que el texto describe el estado del juego y los jugadores progresan introduciendo comandos textuales. Para resolver este tipo de juego, un agente inteligente debe ser capaz de explorar el entorno, aprender mecánicas, identificar su propósito, entender texto y adquirir una cierta percepción temporal.

Para conseguir esto, el proyecto se apoyará en la rama del machine learning, concretamente en el deep reinforcement learning que ha demostrado ser la mejor solución para entornos virtuales en los que agentes tienen que aprender a escoger las mejores acciones para alcanzar determinados objetivos. Esta rama se ha convertido en una de las ramas más prometedoras dentro del área de la inteligencia artificial. Los algoritmos de deep reinforcement learning DQN y A2C han demostrado un gran rendimiento en este tipo de entornos por lo que nuestros agentes se basan en estas arquitecturas.

Al tratarse de un entorno puramente textual, el proyecto utilizará herramientas propias del campo del procesamiento del lenguaje natural. La comprensión del lenguaje requiere habilidades como la memoria a largo plazo, la planificación y el sentido común, cualidades que nuestro agente necesitará.

Para la gestion y creacion de juegos textuales se utilizará Textworld. TextWorld es una biblioteca de Python desarrollada por microsoft que gestiona estos juegos además de proporcionar funciones de seguimiento y control de recompensas. También permite crear nuevos juegos o generar automáticamente otros. Sus mecanismos generativos dan un control preciso sobre la dificultad, ámbito y el lenguaje de los juegos construidos.

Palabras clave: Machine learning, Deep Reinforcemen Learning, Textworld, Natural Lenguage Processing.

Abstract

The objective of this project is to train two intelligent agents to be able to solve text games, in particular, a simple game inspired by ETSIT. In addition, metrics will be compared to measure their performance in relation to a random decision making agent.

Text-based games (very popular in the 80s) are complex and interactive simulations in which the text describes the state of the game and the players progress by introducing textual commands. To solve this type of game, an intelligent agent must be able to explore the environment, learn mechanics, identify its purpose, understand text and acquire some temporal perception.

To achieve this, the project will be based on the machine learning field, specifically on deep reinforcement learning that has proved to be the best solution for virtual environments where agents have to learn to choose the best actions to achieve certain objectives. This branch has become one of the most promising branches in the area of artificial intelligence. The deep reinforcement learning algorithms DQN and A2C have demonstrated a great performance in this kind of environments, so our agents are based on these architectures.

As games are a purely textual environment, the project will use tools from the field of natural language processing. Understanding language requires skills such as long-term memory, planning and common sense, qualities that our agent will need.

Textworld will be used for the management and creation of textual games. TextWorld is a Python library developed by Microsoft that handles these games in addition to providing tracking and reward control functions. It also allows us to create new games or automatically generate others. The generative mechanisms it offers give precise control over the difficulty, scope and language of the games built.

Keywords: Machine learning, Deep Reinforcemen Learning, Textworld, NaturalLenguage Processing.

Agradecimientos

Gracias en primer lugar a mis padres por apoyarme y motivarme durante toda la carrera además de financiarla.

Gracias a Carlos Ángel Iglesias por orientarme y guiarme durante este proyecto además de potenciar mi interés en el campo de la Inteligencia Artificial.

Gracias a mis amigos por estar siempre dispuestos a compartir cualquier momento y ayudar en cualquier situación.

Gracias a todos los compañeros y profesores que han conseguido hacer de esta una gran etapa de aprendizaje.

Gracias a todas las personas anónimas que suben conocimiento a internet de forma desinteresada.

Contents

List of Figures

CHAPTER 1

Introduction

This chapter includes a summarized explanation of the context concerning this project and a list of the objectives to be achieved.

1.1 Context

Text-based games [4] are complex, interactive simulations in which text describes the game state and players make progress by entering text commands. They are fertile ground for language-focused machine learning research.

Language understanding [5] requires skills like long-term memory and planning, exploration, and common sense. Intelligent agents are able to implement these properties with the combination of natural language processing and deep reinforcement learning.

In this work, Textworld [4] has been selected as an enviroment for testing and training intelligent agents. TextWorld is a Python library that handles interactive playthrough of text games and provides functions to tracking and control rewards. It also enables users to handcraft their own games or automatically generates new ones. Its generative mechanisms give precise control over the dificulty, scope, and language of constructed games.

1.2 Project goals

The objectives of this work are the following:

- Understand the uses and limitations of the Textwold learning environment.
- Study and analyze the state of the art of reinforcement learning and in particular of natural lenguage processing and the resolution of text-based games.
- Train and test different agents using the state of the art algorithms.
- Create a custom game using TextWorld and use it as a benchmark for trained agents.
- Compare and measure the performance of algorithms.

1.3 Structure of this document

In this section we provide a brief overview of the chapters included in this document. The structure is as follows:

1. **Introduction:** this chapter provides a global vision of the project.
2. **State of the art**: the most recent stage in the development of all main technologies involved in the project.
3. **Enabling Technologies**: this chapter describes all the tools and technologies that have been used.
4. **Development of the Games**: description of the games creation process.
5. **Deep Reinforcement Learning agents**: description of the design details involving the intelligent agents.
6. **Experiments**: the results of the intelligent agents in the training process and the etsit game test.
7. **Conclusions and future work**: this chapter provides a description of the conclusions that have been obtained as a result of this thesis as well as a proposed approach to apply in future works.

CHAPTER 2

State of the art

This chapter is a summary of the latest developments in the fields of Natural Language processing, Machine Learning and specifically Reinforcement Learning.

The constant stream of innovation and success on this areas has made it possible to solve challenges previously deemed impossible for computer systems.

2.1 Machine Learning

Machine Learning (ML) [6] is a branch of artificial intelligence, whose objective is to develop techniques that enable to emulate human intelligence by learning from the surrounding environment. It is said that an agent learns when its performance improves with experience, that is to say when the skill was not present in initial conditions. That yields not to program explicitly the rules that a model follows. These models must be able to generalize behaviors and inferences to a broader set of data.

Techniques based on machine learning have been applied successfully in diverse fields ranging from pattern recognition, natural language processing, art, computer vision, finance, spacecraft engineering, entertainment, and computational biology to biomedical and medical

applications.

The different machine learning algorithms are grouped according to their output. The main types are [6]:

1. **Supervised learning**: Algorithms that establish a correspondence between the inputs and the desired outputs of the system. It is often used to solve classification problems, where the learning system tries to label (classify) a series of vectors using one among several categories (classes).

2. **Unsupervised learning**: The entire modeling process is carried out on a set of examples consisting of system inputs only. There is no information about the categories of these examples. Therefore, in this case, the system has to be able to recognize patterns in order to label the new entries.

3. **Semi-supervised learning**: This type of algorithm combines the two previous algorithms in order to classify properly. Marked and unmarked data are taken into account.

4. **Reinforcement Learning**: The algorithm learns by observing and interacting the world around it. The model input is the feedback it gets from the outside world in response to its actions. Therefore, the system learns on an assay-error basis in order to increase a reward system.

This thesis focuses in the Reinforcement learning area.

2.2 Reinforcement Learning (RL)

Reinforcement Learning (RL) [7] is an area of machine learning (ML) whose occupation is to determine what actions a software agent should choose in a given environment in order to maximize some reward. To achieve this goal the agent must learn behavior through trial and error interactions with a dynamic environment. In the standard reinforcement learning model, an agent is connected to its environment via perception and action .The environment changes when the agent acts on it, but may also change on its own.

The agent also perceives a reward signal from the environment, it is a scalar that measures the quality of the decision. The goal of the agent is to maximize its cumulative reward. Reinforcement learning methods are ways that the agent can learn behaviors to achieve its goal.

The great promise and goal of reinforcement learning are agents that can learn to solve a extensive range of important problems. According to some definitions, an agent that can learn to perform at or above human level across a vast variety of tasks is an artificial general intelligence [8].

The agent has two main objectives exploration of the unknown environment and exploitation of current knowledge. To achieve this the agent must address three challenges simultaneously [9]:

1. **Generalization**: be able to learn efficiently from data it collects.

2. **Exploration**: interact and learn the rules of the environment prioritizing the right experience to learn from.

3. **Long-term consequences**: consider effects on future states, beyond a single time step.

2.2.1 Environment and actions

The environment is a fundamental part of understanding RL, depending on the type of environment, some algorithms can be used or others. The different environments are normally modeled in consonance with to the following characteristics [1]:

- According to the world's access to information it can be fully observable (it is always known the state of the environment at any time, chess is an example of this) and partially observable, not all information in the environment is accessible with poker as an example.

- In relation to the number of agents interacting in the environment it can be single agent or multi agent.

- According to the influence of the azaar on the development of the future states of the system, the enviroments are deterministic (there is no azaar influence) or stochastic (the influence of the azaar appears).

- Depending on the range of values or states in which the environment can be found discrete enviroments, or continuous enviroments.

2.2.2 Definitions

A **state** s [6] is a complete description of the state of the world. An **observation** o [6] is a partial description of a state, which may omit information.

The action space [6] is the set of all valid actions in a given environment. Some environments, have discrete action spaces, where the agent has a limited number of actions available. Other environments have continuous action spaces, where the agent has a infinite number of actions available.

A **policy** [6] is a rule used by an agent to decide what actions to take. It can be deterministic, in which case it is usually denoted by μ:

$$a_t = \mu(s_t) \tag{2.1}$$

or it may be stochastic, in which case it is usually denoted by π:

$$a_t \sim \pi(\cdot|s_t) \tag{2.2}$$

A trajectory [6] or **episodes** is a sequence of states and actions in the world, it is denoted by τ :

$$\tau = (s_0, a_0, s_1, a_1, ...) \tag{2.3}$$

State transitions depend on only the most recent action, a_t. They can be either deterministic,

$$s_{t+1} = f(s_t, a_t) \tag{2.4}$$

or stochastic,

$$s_{t+1} \sim P(\cdot|s_t, a_t) \tag{2.5}$$

Since the transitions only depend on the most recent state-action and no prior history, they are modeled as Markov Decision Processes (MDP) [7].

An MDP is a 5-tuple, $\langle$ S, A, R, P, ρ_0 $\rangle$, where S is the set of all valid states, A is the set of all valid actions, R is the reward function, with R : S $\times A \times S \rightarrow Rr_t = R(s_t, a_t, s_{t+1})$,

P is the transition probability function P : S $\times A \rightarrow \mathcal{P}(S)$.

The general case of a model with an intelligent agent in an environment is as follows:

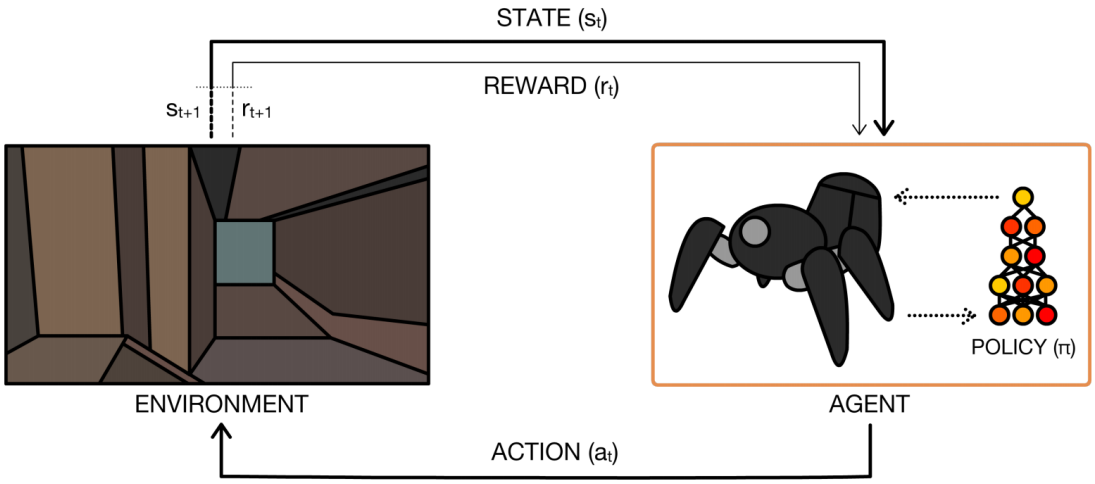

Figure 2.1: The perception-action-learning loop [1].

.

On each step of interaction, the agent receives as **input** i an **observation** of the current state of the environment (could be partial or total) s. Based on this information the agent chooses an **action** a to generate as an output to perform, this action changes the estate of the environment. To measure the quality of the agent's decision, a **reward function** gives the agent a reinforcement signal, $r_t = r(s_t, a_t)$ and producing the succeeding state s_{t+1}.

The **agent's behavior** B should choose actions that tend to increase the long-run values of the reinforcement signal. The task of the agent is to learn a policy, for selecting the next action at based on the current state Π : S $\rightarrow$ A, such that the reward V is maximised [9] :

$$\sum V^{\pi}(s_t) = r_t + \gamma r_{t+1} + \gamma r^2 r_{t+2} + ... = \sum_{i=0}^{\infty} \gamma r^i r_{t+i} \tag{2.6}$$

where γ is a constant between 0 and 1 that determines the relative value of delay reward versus immediate rewards. The **optimal policy** is often denoted Π* and the corresponding cumulative reward V* .

The agent's job is to find a policy, mapping states to actions, that maximizes some long-run measure of reinforcement [7].

Video games provide plenty of problems for agents to solve, making their perfect environments for AI research. There are virtual environments safe and controllable. In addition, these game environments provide a large supply of useful data for reinforcement learning algorithms.

These characteristics make games the unique and favorite domain for AI research. Games are an excellent testbed for reinforcement learning research. Studying games, we can learn about human intelligence, and the challenges that human intelligence needs to solve [10].

2.2.3 Reinforcement Learning algorithms

One of key points in an RL algorithm is the question if the agent has access or learns a model of the environment. A model of the environment is a function which predicts state transitions and rewards.

An agent with access to the model of the environment can plan by thinking ahead, predicting what would happen for a range of possible choices, and explicitly deciding between its options. Agents can then select the best results from planning into a learned policy. A famous example of this approach is AlphaZero [11]. Model-based algorithms have sow improvement in sample efficiency over methods that do not have a model.

The problem is very difficult or impossible to know the ideal model in advance, therefore the agent must learn the model through pure experience. This brings a series of challenges that must be analyzed as the possibility that the agent learns a biased model that still allows him to reach his goal.

Algorithms which use a model are called model-based methods, and those that do not are called model-free. For each of these types the agents have different challenges in terms of optimization and learning.

2.2.3.1 Model-free algorithms

On model-Free Algorithms, there are many different ways of using models. The following is a list of the most popular ways. In each case, the model may either be given or learned [2].

- **Policy Optimization Methods:** The policy is represented explicitly as $\pi_\theta(a|s)$.

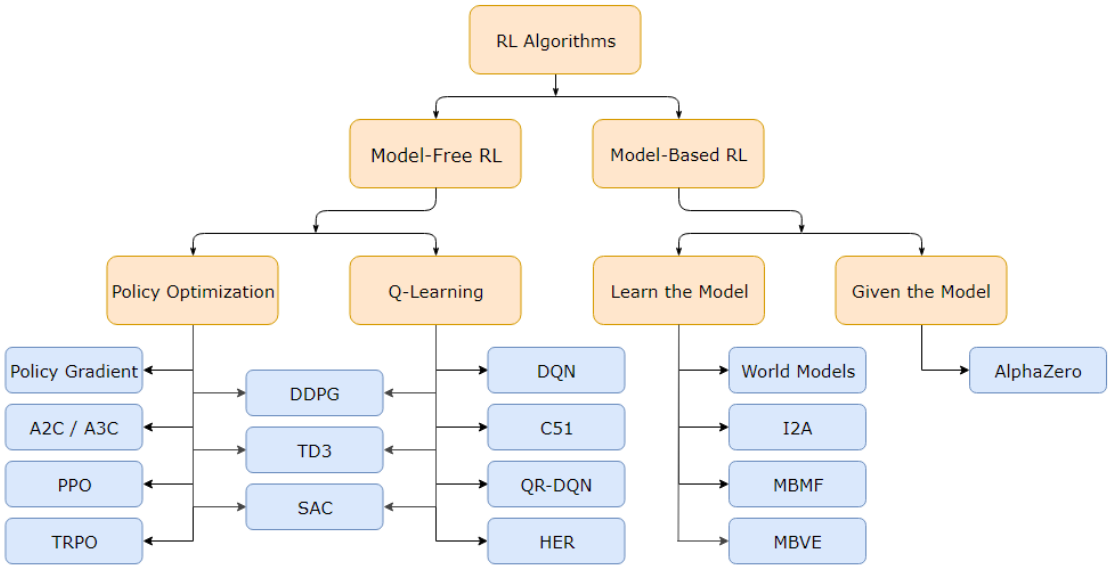

Figure 2.2: Open AI - Taxonomy of algorithms in modern RL [2].

The objective is optimize the parameters θ by gradient ascent on the performance objective $J(\pi_\theta)$, or by maximizing local approximations of $J(\pi_\theta)$. Each update only uses data collected while acting according to the most recent version of the policy. Policy optimization also usually involves learning an approximator $V_\phi(s)$ for the on-policy value function $V^\pi(s)$, which is used the update of the policy.

Examples of these algorithms are A2C, A3C or PPO that have proven to be the most efficient for certain problems.

- **Q-Learning Methods:** In this family the agents learn, for the optimal action-value function, $Q^*(s,a)$, an approximator $Q_\theta(s,a)$. Typically they use an objective function based on the Bellman equation. This optimization is performed off-policy, the data collected at any time by the agent during the training is accessible for each update, regardless of how the agent has explored the environment. The corresponding policy is obtained via the connection between Q^* and π^*: the actions taken by the Q-learning agent are given by:

$$a(s) = \arg\max_a Q_\theta(s,a). \tag{2.7}$$

A couple of examples of policy Q-learning methods are DQN, used for solving Atari games for DeepMind or C51 a variant of DQN.

2.2.3.2 Model based algorithms

In the model-based reinforcement learning algorithms, there are many orthogonal ways of using models. This is a list of the most used and highest performed methods. In each case, the model may either be given or learned [2].

- **Background Pure Planning**. It do not represents the policy instead, uses pure planning techniques like model-predictive control (MPC) to select actions. In MPC, In fixed time windows, this family of algorithms calculates the optimal series of actions based on the model and discards the rest. MBMF is an example of this type of algorithms.
- **Expert Iteration.** The agent uses a planning algorithm (like Monte Carlo Tree Search) in the model. Pure planning generates possible actions by sampling an explicit representation of the policy, $\pi_\theta(a|s)$. AlphaZero is an example of this approach.
- **Data Augmentation for Model-Free Methods**

  This methods Uses a model-free RL algorithm to train a policy and fictitous experience for updating the agent. or augment real experiences. MBVE or World Models are an example of this approach.
- **Embedding Planning Loops into Policies.**This approach makes model bias less of a problem. The action plan is produced into a policy as a subroutine this allows the policy learn to choose how and when to use the plans. I2A is an example of this type of agents.

2.3 Deep Reinforcement Learning

Deep reinforcement learning combines artificial neural networks with a reinforcement learning techniques that gives reinforcement learning agents the ability to solve complex problems of the environment [12].

While deep neural networks are behind recent AI breakthroughs in a plenty kind of problems like machine translation and time series prediction they can also combine with reinforcement learning, as demonstrated by Deepmind and OpenIA, allowing them to achieve much higher performance than those obtained without using deep neural networks. They have designed algorithms than can beat human experts playing numerous Atari video games [12], Starcraft II and Dota-2, as well as the world champions of Go. This is a large im-

provement over reinforcement learning's previous accomplishments. With the help of deep neural networks, the state of the art is progressing rapidly.

2.3.1 Deep neural networks

A standard Neural Network (NN) [13] consists of many simple, connected processors called neurons, Each one producing a sequence of real value activations. Input neurons are activated through sensors that measure the environment, other neurons are activated through weighted connections of active neurons. Some neurons are able to effect the environment by causing actions and generating emerging behaviors. Depending on the problem and how the neurons are connected, such behaviour may require long causal chains of computation stages. The Deep Neural Network (DNN) refers to many of these stages [13].

A deep neural network [6] is an neural network with multiple layers between the input and output layers. DNN moves through the layers by calculating the probability of each output finding the correct mathematical manipulation to convert input to output, whether it is a linear or non-linear relationship [6].

Deep neural networks can model complex non-linear relationships through the following process: At first, the DNN assigns random numerical values, or weights, to connections between neurons. The weights and inputs are multiplied and return an output. Then, an machine learning algorithm would adjust the weights, until it determines the correct mathematical manipulation to fully process the data.

Since the emergence of Deep Neural Networks as a prominent technique in many different fields, many different architectures have emerged, each with its own peculiarities such as recurrent neural networks (RNNs), in which data can flow in any direction, are used in language modeling, long short-term memory (LSTM) is particularly effective for this use. Or convolutional deep neural networks (CNNs) which are used in computer vision and speech recognition [3].

2.3.2 Deep Q-learning

This approach has several advantages over standard Q-learning. Each step of experience is potentially used in many weight updates, which enables greater data efficiency [12].

The weight of a step from a **state** Δt is calculated as $\gamma^{\Delta t}$, the **discount factor** γ is a number between 0 and 1 ($0 \leq \gamma \leq 1$) and evaluates rewards received previously with a higher value than those received later. It makes rewards from the uncertain far future less

important for our agent than the ones in the near future. γ can also be interpreted as the probability of succeeding (or surviving) at each step of Δt.

The algorithm, therefore, has a function that calculates the quality of a state-action combination [12]:

$$Q : S \times A \rightarrow R. \tag{2.8}$$

Before learning begins, Q is initialized to an arbitrary random value. Then, at each time t the agent selects an **action** a_t, observes a **reward** r_t, introduces a new state $s_t + 1$(which depends on the previous state s_t, and Q is updated. The core of the algorithm is an update of the simple iteration value, making the weighted average of the old value and the new information. However, it does not have access to Q* because the world is complex and huge. But, neural networks are used as universal function approximators to create an approximation to Q*. To update rule, it is used fact that every Q function for some policy obeys the Bellman equation, for a deterministic environment [12]:

$$Q(s,a)^{new} = Q(s,a) + \alpha[r_t + \gamma \max Q(s_t{+}_1, a) - Q(s_t, a_t)] \tag{2.9}$$

Q function predicts what the return would be, then it is easy to construct a policy that maximizes our rewards [12]:

$$\pi(s) = \arg\max_a Q(s,a). \tag{2.10}$$

2.3.3 Advantage Actor Critic (A2C)

Actor-Critic algorithms [14] is a Policy-Based algorithms that try to find the optimal policy directly. Actor-Critics try to find or approximate the optimal value function, which is a mapping between an action and a value. The higher the value, the better the action.

The principal idea is to split the model in two: one for computing an action based on a state and another one to produce the Q values of the action.

The actor [15] takes as input the state and outputs the best action. It essentially controls how the agent behaves by learning the optimal policy (policy-based). The critic [15], on the other hand, evaluates the action by computing the value function (value based). Those two models participate in a game where they both get better in their own role as the time

passes. The result is that the overall architecture will learn to play the game more efficiently than the two methods separately.

The actor used to be a function approximator like a neural network and its task is to produce the best action for a given state.

The critic is another function approximator, which receives as input the environment and the action by the actor, concatenates them and output the action value (Q-value) for the given pair.

Policy gradient [14]:

$$\Delta J(Q) = E_\tau[\sum_{t=0}^{T-1} \nabla_Q log\pi_Q(a_t, s_t)G_t] \tag{2.11}$$

Rewriting [14]:

$$\Delta J(Q) = E_\tau[\sum_{t=0}^{T-1} \nabla_Q log\pi_Q(a_t, s_t)Q(s_t, a_t)] \tag{2.12}$$

The Q value can be learned by parameterizing the Q function with a neural network as DQN methods do. Then the Critic estimates the value function, this could be the action-value (the Q value) or state-value (the V value). On the other hand Q-learning algorithm critiques the action that the actor selected, providing feedback on how to adjust.The Critic network and the Value network are updated at each update step.

The advantage value is called to the subtract the Q value term with the Value, this represents how much better it is to take a specific action compared to the average, general action at the given state [14]:

$$A(s_t, a_t) = Q(s_t, a_t) - V(a_t) \tag{2.13}$$

The objetive of the agent is minimizing policy loss and value loss [14].

$$Policyloss = \sum_{t=0}^{T-1} \nabla_Q log\pi_Q(a_t, s_t)A(s_t, a_t) \tag{2.14}$$

$$Valueloss = \frac{1}{2}(V(a_t) - E_\tau)^2 \tag{2.15}$$

2.4 Natural Language Processing (NLP)

Natural Language Processing (NLP) [16], is a branch of computer science, artificial intelligence and linguistics that studies the interactions between computers and human language. Its objective is to read, decipher, understand and make sense of the human languages [16].

Natural Language Processing is generally used on **text summary**, the algorithm should find the central idea of an article and ignore what is not relevant. **ChatBots**, they should be able to have a fluid chat with the user and answer his questions automatically. **Automatic keyword generation** and **text generation** following a particular style. Entity Recognition, find People, Commercial, Government Entities or Countries often used this approach. **Sentimental Analysis**, identify subjective moods or opinions in large amounts of text, including sentiment mining and average opinions, it is in charge of analyzing and detecting the feelings associated with a communication and classifying it, widely used in social networks, politics, product reviews and recommendation engines. **Automatic Language Translation**, automatic classification of texts into pre-existing categories or from full texts, detecting recurrent topics and creating the categories.

An NLP model should typically solve the following high-level problems [6]:

1. **Content Categorization**: A linguistics-based summary of the document, including search and indexing, content alerts, and duplication detection.
2. **Topic discovery and modeling**: Accurately capture meaning and topics in text collections, and apply advanced analytics to text, such as optimization and forecasting.
3. **Contextual extraction**: Automatically extract structured information from text-based sources.
4. **Speech-to-text and text-to-speech conversion**: Transforming voice commands into written text and vice versa.
5. **Summarizing documents**: Automatic generation of synopses of large bodies of text.
6. **Machine based translation**: Automatic translation of text or speech from one language to another.

NLP [16] includes different techniques for interpreting human language, ranging from statistical and machine learning methods to rule-based and algorithmic approaches. Text-based and speech-based data vary widely, as do practical applications.

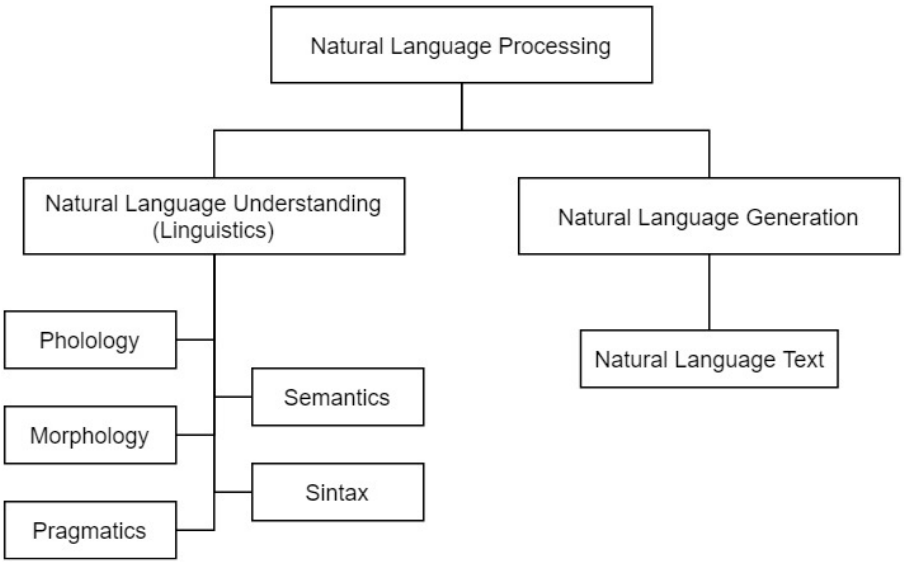

Figure 2.3: Phases of NLP architecture [3].

Basic NLP tasks include symbolization and syntactic analysis, derivation, speech labeling, language detection, and identification of semantic relationships. NLP tasks break language into shorter elementary pieces, attempt to understand relationships between the pieces, and explore how the pieces work together to create meaning.

A text is a set of words formed by letters with a specific disposition, to process text you must first treat it, the techniques generally used to treat the text and make it accessible to the algorithms are the following [6]:

- **Tokenization** is the process of demarcating and possibly classifying sections of a string of input characters, separating words from text into entities called tokens.

- **Tagging Part of Speech (PoS)**: is the process of marking up a word corresponding to a particular part of speech based on its definition and context. Classifying sentences in verb, noun, preposition adjective, etc.

- **Shallow parsing** : used to understand the grammar in sentences. The tokens are parsed and a structure tree is created from their PoS.

- **Meaning of the words**: lexical semantics and word sense disambiguation.

- **Pragmatic Analysis**: it focuses on taking a structure set of text and figuring out what the actual meaning was. Is used to detecting how things are said such as irony or sarcasm.

- **Bag of words**: this is a way of representing the vocabulary that will be used in a model.
- **Word2vec**: this is a technique that learns from reading huge amounts of text and memorizing which words appear to be similar in different contexts. Similar words are placed close together. By using pre-trained vectors, we are able to have a wealth of information to understand the semantic meaning of the texts.

Advances in techniques and hardware for deep neural network training have recently provided significant improvements in the accuracy of many critical NLP tasks. Their needs fewer engineering features due to achieving high performance. The main examples of these new techniques are convolutional neural networks (CNN) and recurrent neural networks (RNN). Gateway mechanisms have been developed to alleviate this. Some limitations of basic RNN, resulting in a long term memory (LSTM) and a closed recurrent unit (GRU) [17].

2.5 Conclusion

As shown in the previous sections, the machine learning field, in particular the deep learning field has had an explosive development in the last years. These advances have been reflected in the reinforcement learning models due to the use of these techniques have achieved superhuman performances in different environments.

On the other hand advances in NLP are abundant allowing researchers to create intelligent models capable of working with text at much deeper levels.

At the intersection of these two worlds appear frameworks like Textworld that mixes the best of NLP and RL. This opens the way to new lines of research that, if the trend continues, will grow over the next few years.

This will allow the training of intelligent models in purely textual environments with the aim of achieving some kind of goal, which will have a great impact on emerging technologies such as chatbots.

CHAPTER 3

Enabling Technologies

This chapter provides an overview about the technologies in which this project is based.

3.1 GYM

OpenAI Gym [18] is a toolkit for developing and comparing reinforcement learning algorithms. Gym is an open source interface to reinforcement learning tasks. It is a flexible interface that does not have a predefined structure of the agents and is compatible with any numerical computation library, such as Pytorch.

Gym is based on the construction of environments in a structured way that allows intelligent agents to interact with the environment in an orderly and quantifiable way. The core gym interface is Env, which is the unified environment interface. There is no interface for agents. Env's main methods that will be used in this project are:

- Reset(self), reset the environment's state and returns observation (object), an environment-specific object representing your observation of the environment.
- Step(self, action), step the environment by one timestep and returns the observation (object), reward (float), the amount of reward achieved by the previous action, if the

episode is done (boolean) and inf (dict), it is diagnostic information for debugging. It can be useful for learning.

- Render(self), render one frame of the environment.

Gym provides a list of pre-built enviroments where to train the smart models although in this project we will not use any of this list. Instead we will build our custom enviroment.

3.2 PYTORCH

PyTorch [19] is a library for Python programs that facilitates building deep learning projects, it emphasizes flexibility and allows deep learning models to be expressed in a easy compressible way. It is a scientific computing package that uses the power of graphics processing units.

It provides tensor computations with strong GPU acceleration support and building deep neural networks on a tape-based autograd systems.

In addition to this, PyTorch offers dynamic computational graphs that can be changed during runtime. Provides classes and functions implementing automatic differentiation.

3.3 TextWorld

TextWorld [4] is an extensible Python framework for generating text-based games and train and test AI agents in abilities such as language understanding, memory, planning, generalization and transfer learning. This framework has two main components: a game generator and a game engine. The game generator returns a executable game based on some game specifications, such as number of rooms, number of objects, game length, and winning conditions. The game engine allows us to play and test different models.

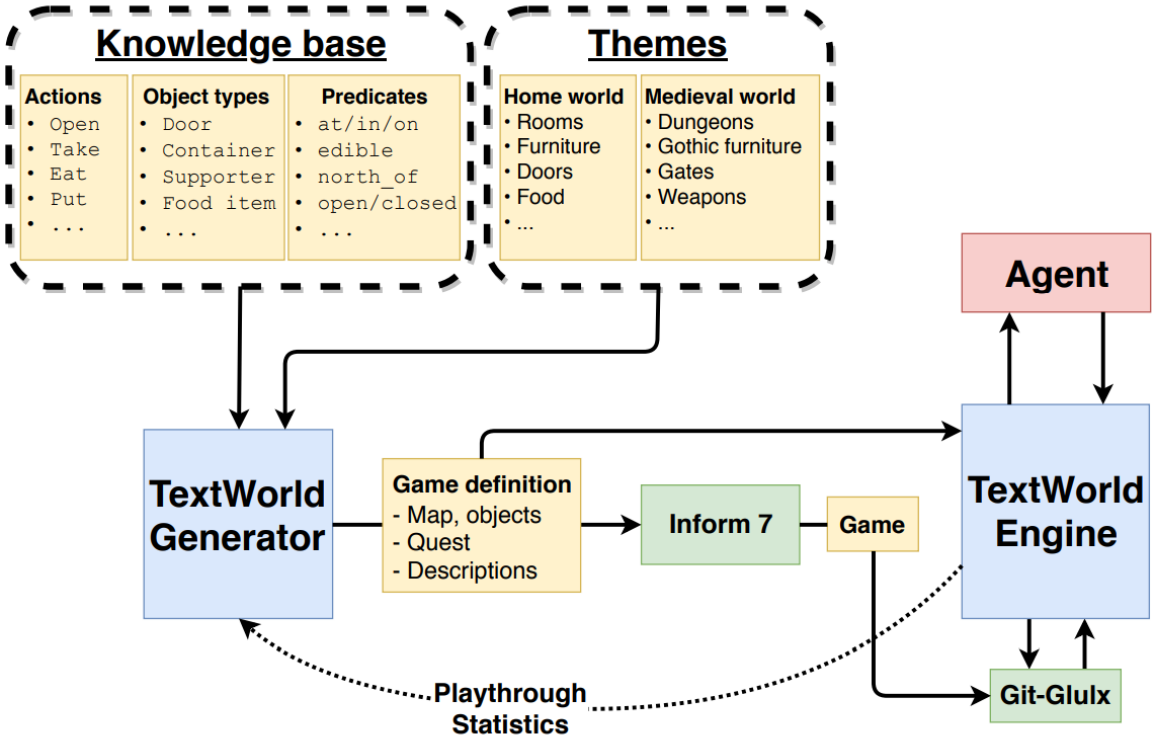

Figure 3.1: An overview of Textworld's framework architecture [4].

Inform 7 and Git-Glulx are third-party libraries, they works as follow: some game definitions are first converted to Inform 7 code and compiled into a Glulx executable file. The way agents interact with the game is by communicating with the Git-Glulx interpreter via TextWorld.

3.3.1 Game Generation

TextWorld's game generator takes the game specification values: the number of rooms and objects, the length of the quest, the winning conditions and options for the text generation. Texworld uses all this information to outputs the corresponding executable game. It made possible to generate a combinatorial set of games [4].

The way to use it is as follows:

```
textworld.make(world_size=1, nb_objects=5, quest_length=2, grammar_flags
    ={}, seed=None, games_dir='')
```

Where it is posible to control the number of rooms in the world `worl_size`, the number of objects `nb_objects`, the number of actions the quest requires to be completed `quest_length` and modify the grammar options. Also, it is possible to track the random games generation with a seed.

In order to build the executable game by high-level specifications, TextWorld must deal with the world generation, the quest generation and the text generation.

3.3.1.1 World Generation

To generate a world, Textworld first create a map of the world. The map generation process is parameterized by the grid size of the world, the number of rooms, and whether room connections should have doors or not. With this information the framework generates maps using the random walk algorithm [20], this made possible to create a huge variety of configurations.

After the map is generated, objects are added to the world randomly across the rooms. There are two main types of objects, portable ones which means they can be grab in the inventary or throw and the fixed in place objects which can not be grabed, this object can be supporter or container of portable ones.

3.3.1.2 Quest Generation

A quest [4] is a sequence of actions the player must perform to win the game. This sequence does not have to be optimal or unique. Many trajectories could lead to a winning state but no all paths are interesting, for example, the ones which contain cycles. The Textworld process of determining interesting end goal is defining the nature of a quest and choosing significant rewards for the agent.

Therefore, the framework's forward quest generation algorithm produces all possible quests from an initial condition then searches for those that complete the ending constraint. However, this approach becomes intractable if the length of the desired quest increases. To remedy this, TextWorld also supports backward chaining. Backward chaining starts from a specified end state rather than an initial state. This approach solves the complexity problem

of forward quest generation.

The generative process can add missing objects and facts to the world as needed, which yields more diverse quests and games.

3.3.1.3 Text Generation

The Text Generation module [4] renders coherent text using logical elements of the game state. The engine uses a context-free grammar of Chomsky, to generate object names, room descriptions, and quest instructions in constrained natural language. Using a context-free grammar gives a degree of textual variation, while also ensuring strict control over the results.

The module is essentially a set of grammars, each generating distinct aspects. These grammars can be extended and modified easily with additional production rules, enabling the generation of simpler or more complex sentence structures that may act as a level of game difficulty. On a high level, it is possible to differentiate the following text generation groups [4]:

Object Names are randomly selected from the context-free grammar and uniquely assigned to objects. An object name is decomposed in adjective and noun. The adjective is optional and is used to create more complex object names and descriptions.

The **Room Descriptions** are the concatenation of every object they contain. Textworld detects an on-game change to the states of objects in the room and updated the description.

Quest Instructions are used to explain to the player what should do in a game. An instruction describes a particular action. It is possible to manage the difficulty by adjusting how often the descriptions are shown, descriptions can be displayed for every action of a quest for an easier game, only the final action, or never, this force the player to figure out what to do from scratch.

Text options offer some control over different aspects of the text generation. Two themed grammars are available to choose: the house grammar describes the world as if the game takes place in a modern house and the basic grammar uses a simple grammar with almost no linguistic variation. The basic grammar reduces the vocabulary and the language complexity to ease the training of agents.

3.3.2 Learning Enviroment

TextWorld can be used to play any text-based game that are interpretable either by Z-machine or by Glulx. The framework handles launching and interacting with the necessary game processes, it provides a simple API for game interaction. Textworld also contains additional information about the game course in the game state object that can be used to train intelligent agents [4].

The game state object contains useful information such as: The interpreter's response to the previous command, the description of the current room, the player's inventory, the current location, the name of the current room and the current score.

It can also include Additional information: A text describing what the player must do to win the game, a list of commands that are Admissible, an intermediate reward and a list of commands that guarantees to win the game starting from the current game state the winning policy.

Determine a winning policy for any game state is possible by tracking the state of the player. Textworld guaranteed that a winning policy exists from the player's initial position, then it is possible to update the winning policy by monitoring state changes. The policy attaches to the last action the agents perform depending on if this action made the agent closer, farther o equal to the goal.

TextWorld can provide a final reward [4] and an intermediate reward which is tied to the winning policy. After every command, if the winning policy increases, meaning that as a result of the last action, additional commands are required to solve the game, then a negative reward is assigned. If the winning policy shortens, meaning the last action brought the agent closer to the goal, a positive reward is assigned.

CHAPTER 4

Development of the Games

This chapter provides an explanation of how Textworld works. This framework made possible an easy way to build random or custom games that will be used for training and testing intelligent agents. Furthermore, it will be designed a custom game for a more natural and real world environment inspired in a ETSIT quest.

4.1 ETSIT game

This section provides an explanation of the process of building a game using Textworld than will be used as a benchmark to test our intelligent agents and be able to measure their performance in a simulated real environment.

Textworld provides a simple way to build our custom handcrafting text-based games using the GameMaker API. Also It is possible to visualize the world, rooms, doors and objects.

This game will be inspired by the "Escuela Técnica Superior de Ingenieros de Telecomunicación" (ETSIT). The player will start in the hall and his goal will be to eat an apple hidden in the microwave of the coffee shop, the door to the cafeteria will be locked by a

key, so the player must explore the school to find the key that will open the door and then explore the cafeteria to find the apple.

Figure 4.1: Final structure of the world.

An intelligent system must be able to explore the world, learn mechanics, identify its purpose automatically, understand text and acquire a certain sense of temporal and semantic meaning of words in order to complete their mission. All this makes it an ideal environment to test our agents.

The steps to build this game will be explained in the following points.

4.1.1 Creating the world

First of all, it is needed a wold than contains all the elements we want, to do that it is used the `GameMaker()` constructor of the textworld library.

```
World = GameMaker()
```

The object `World()` has the functions to test and play the game. It will contain all the objects but for now it is empty.

4.1.2 Structure of the world

First of all, it is needed to design the structure that the world will have. It is necessary to have a clear design taking into account the different rooms, the corridors that connect them and where the player will start using the `new_room` `connect` and `set_player` methods respectively. It could also be possible to add items to the character's inventory, but it will not be done in this case.

```
hall = World.new_room("Hall")
coffee_shop = World.new_room("Coffee shop")
library = World.new_room("Library")

corridor_hall_coffee_shop = World.connect(hall.east, coffee_shop.west)
corridor_hall_library = World.connect(hall.west, library.east)

World.set_player(hall)
```

Our world will be composed of 3 rooms: the hall, the coffee shop and the library. The player will start in the hall without any object in his inventory.

It is possible to visualize the current world structure using `World.render()` method:

Figure 4.2: A world with a player in a hall, and a library and a coffee shop adjacent.

4.1.3 Add objects

The engine allows us create different objects with the `new()` method of several types : “r” room can contain objects and can be connected with other rooms, “d” door can be lockable, openable or closable, “c” container can hold objects and can be lockable, openable or closable, “s” supporter can hold objects, “o” portable object can be carried by the player. Portable objects have the subtypes “k” key, match a door or container’s lock, “f” food, can be eaten, “oven” oven, provide a heat source to cook food item, “stove” stove, provide a heat source to cook food item.

Our world will have 2 notes that will serve the player as clues, a table, a closed microwave and an apple that is the ultimate goal.

```
note_welcome = World.new(type='o', name="Welcome", desc = "You are in an
    ETSIT game!, You can go west (library) or east (Coffee shop). Eat an
    apple to win!")
note_library = World.new(type='o', name="Hint", desc = "This is the library,
     find the coffee shop key.")

table = World.new(type='s', name= "Table")

apple = World.new(type='f', name="Apple")

microwave = World.new(type='c', name="microwave")
World.add_fact("closed", microwave)
```

In order to insert objects to the rooms the `add()` method is used.

```
coffee_shop.add(microwave)
microwave.add(apple)
hall.add(note_welcome)
library.add(table)
library.add(note_library)
```

The world visualization after that is the following:

Figure 4.3: The world after adding notes, recipients (table and microwave) and an apple.

4.1.4 Create the key and door

It is possible to add the following properties to the interactive objects using the `add_fact()` method: "match" creates a connection between a key and the container or door's lock, "open", "closed" and "locked" are the states of containers and doors, "edible" made the food is consumable, otherwise needs to be cooked first.

The world will have a looked door in the corridor between the hall and the coffee shop

and the key to open it will be on a table in the library.

```
door_coffee_shop = World.new_door(corridor_hall_coffee_shop, name="Coffee
    shop door")
World.add_fact("locked", door_coffee_shop)

key = World.new(type="k", name="Coffee Shop Key")
World.add_fact("match", key, door_coffee_shop)

table.add(key)
```

Our world has already finished, his final structure is the following:

Figure 4.4: Final structure of the world.

4.1.5 Record the quest

We have the full structure of the world but there is not a defined goal or objective to complete the game. Textorld allows us create different quests ussing the `record_quest()` method.

```
quest = World.record_quest()
```

This throws an instance of the game that allows us to type all commands (or steps) to include in the quest. It will be recorded the optimal path to achieve the goal: go west, take coffee shop key from table, go east, unlock coffee shop door with coffee shop key, open coffee shop door, go east, open microwave, take apple from microwave, eat apple.

```
-= Hall =-
You arrive in a Hall. A typical kind of place. You begin to take stock of what's
here.


You need an unblocked exit? You should try going east. You need an unblocked
exit? You should try going west.

There is a Welcome on the floor.

Available actions: ['examine Coffee shop door', 'examine Welcome',

'go west', 'inventory', 'look', 'take Welcome']


> go west

-= Library =-
This is the ETSIT library!. There is a table. There is a key on the table.

There is a Hint on the floor.

Available actions: ['examine Coffee Shop Key', 'examine Hint', 'examine Table',


> take key
You take the Coffee Shop Key from the Table.

Available actions: ['drop Coffee Shop Key', 'examine Coffee Shop Key',

'go east', 'inventory', 'look', 'take Coffee Shop Key from Table', 'take Hint']

'examine Hint', 'examine Table', 'go east', 'inventory', 'look',

'put Coffee Shop Key on Table', 'take Hint']


> go east

-= Hall =-
You arrive in a Hall. A typical kind of place. You begin to take stock of what's
here.
```

Figure 4.5: The quest record process using `record_quest()` method (part1).

```
You need an unblocked exit? You should try going east. You need an unblocked
exit? You should try going west.

There is a Welcome on the floor.

Available actions: ['drop Coffee Shop Key', 'examine Coffee Shop Key', 'examine Coffee shop door',

> unlock door
(with the Coffee Shop Key)
You unlock Coffee shop door.

Available actions: ['drop Coffee Shop Key', 'examine Coffee Shop Key', 'examine Coffee shop door',
'examine Welcome', 'go west', 'inventory', 'lock Coffee shop door with Coffee Shop Key', 'look',
'open Coffee shop door', 'take Welcome']


> open door
You open Coffee shop door.

Available actions: ['close Coffee shop door', 'drop Coffee Shop Key', 'examine Coffee Shop Key',
'examine Coffee shop door', 'examine Welcome', 'go east', 'go west', 'inventory', 'look', 'take Welcome']


> go east

-= Coffee Shop =-
This is the ETSIT coffee shop, ther is a microwave, it looks closed.

Available actions: ['close Coffee shop door', 'drop Coffee Shop Key', 'examine Coffee Shop Key',
'examine Coffee shop door', 'examine microwave', 'go west', 'inventory', 'look', 'open microwave']


> open microwave
You open the microwave, revealing an Apple.

Available actions: ['close Coffee shop door', 'close microwave', 'drop Coffee Shop Key', 'examine Apple',
'examine Coffee Shop Key', 'examine Coffee shop door', 'examine microwave', 'go west',
'insert Coffee Shop Key into microwave', 'inventory', 'look', 'take Apple from microwave']

> take apple
You take the Apple from the microwave.

Available actions: ['close Coffee shop door', 'close microwave', 'drop Apple', 'drop Coffee Shop Key',
'eat Apple', 'examine Apple', 'examine Coffee Shop Key', 'examine Coffee shop door', 'examine microwave',
'go west', 'insert Apple into microwave', 'insert Coffee Shop Key into microwave', 'inventory', 'look']

> eat apple
You eat the Apple. Not bad.

Available actions: ['close Coffee shop door', 'close microwave', 'drop Coffee Shop Key', 'examine Coffee Shop Key',
'examine Coffee shop door', 'examine microwave', 'go west', 'insert Coffee Shop Key into microwave', 'inventory', 'look']

Done after 9 steps. Score 0/0.
```

Figure 4.6: The quest record process using `record_quest()` method (part2).

4.2 Random games

Textworld has the possibility to create multiple random games using a predefined `tw-make` [21] command. It provides a systematic way to create plenty of different games with chosen characteristics.

The most interesting features to measure the difficulty of the games are the following: the reward frequency, it can be dense, balanced, or spars, and the description of the game's objective shown at the beginning of the game, it can be detailed, bried, or none.

```
tw-make tw-simple [-h] [--output PATH] [--seed SEED] [--format {ulx,z8}]
                  [--overview] [--save-overview] [--third-party PATH] [-f]
                  [--list] [--silent | -v] --rewards {dense,balanced,sparse}
                  --goal {detailed,brief,none} [--test]
```

We will use these games to train our model in different types of environments with different difficulties in order to guarantee generalization before our agent deal with a custom real game.

CHAPTER 5

Deep Reinforcement Learning Agents

In this chapter, it will be cover the agents' design, the implementation details involving its architecture and an explanation of the environment where these agents will interact.

5.1 Overview

The purpose of this project is to build two agents (A2C and DQN) that are able to solving text games in addition to comparing their performance in relation to a random agent. To achieve this objective we will use Deep Reinforcement Learning. The intelligent agents will learn based on the interaction with the environment, in this case games generated by Textworld.

The agents will interact in the environment (the games) in an action-perception loop that will be as follows:

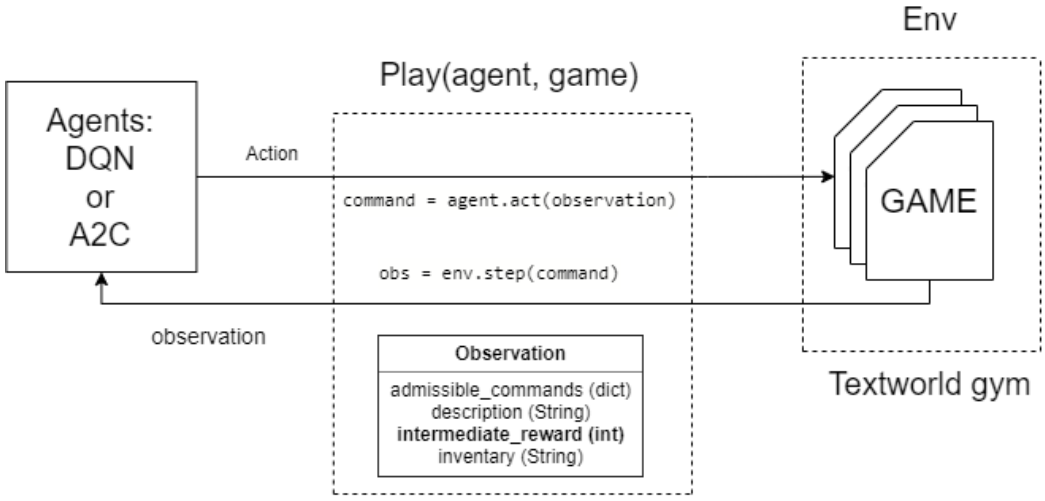

Figure 5.1: Models' action-perception loop.

The main elements are: the games, differentiated by their use (training, validation and test), the observations made by the agent in the environment and the play() function that loads the games and acts as a pipeline between the decisions of the agent and the environment. These elements are fixed and are used for each experiment.

The problem of solving a text game is extensive, an intelligent system must be able to explore their environment, learn mechanics, identify its purpose automatically, understand text, have a vocabulary of the words it know, acquire a certain sense of temporal and semantic meaning of words in order to complete their mission.

To complete all these tasks it will be used two agents composed of different neural networks that will decompose these problems to solve them separately.

5.1.1 Agents

The objective of the agents is to choose a command among those available, they receive the game information in text form (string), process it and decide a command, interact with the world and receive new game information, this cycle is repeated.

Through this cycle we can reward the behaviors that bring the agents closer to the goal and punish the undesired ones causing to converge over time into desired behaviors.

The agents have two states: "train" where it deals with learning, minimizing the error calculated in the compute loss function and updating the weights of their neural networks.

And "test" in which the agents act without modifying their networks, this will be explained in depth in the 5.3 point.

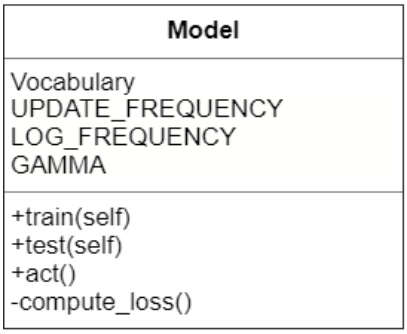

Figure 5.2: Model' class.

The agents have a vocabulary object where it stores the words it knows. It also has parameters to control training: learning rate and update frequency, this allows us to adjust the training of the model.

5.1.2 Games

To train the model it is necessary that it faces different games with different situations and environments in order to the agents will be able to generalize the knowledge learned.

As explained in section 4.2, Textworld allows us to generate random games with variable characteristics. Two sets of games will be created, one for training and the other for evaluation with the following characteristics: rewards will be plentiful and game status descriptions detailed.

The training games set will contain 200 games:

```
seq 1 200 | xargs -n1 -P4 tw-make tw-simple --rewards dense --goal detailed
    --output games/training_games/ --seed
```

The validation games set will contain 20 games:

```
seq 1 20 | xargs -n1 -P4 tw-make tw-simple --rewards dense --goal detailed
    --test --output games/testing_games/ --seed
```

The validation set will be used to guarantee generalization and prevent the model from overfitting to training games set. These games will be stored and not modified, this allows to compare the performances of the agents in the same environments.

After successfully completing these two phases, the agents will progress to the texting phase where they will face the game developed in point 4.3, this game is also the same for both agents.

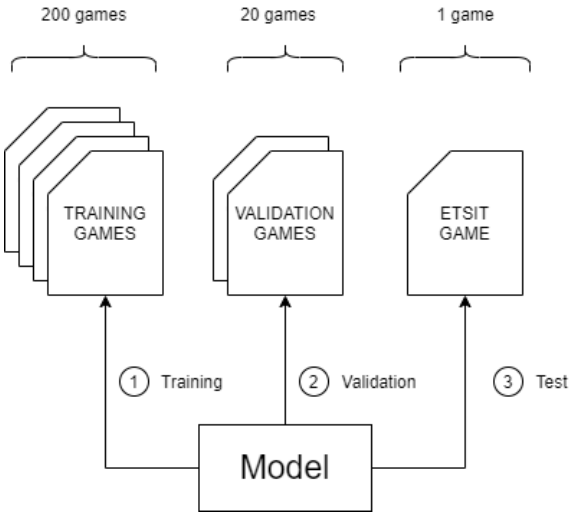

Figure 5.3: Models' learning phases.

To overcome these 3 steps guarantees that the agents have reached a certain cognitive level and are able to face real environments.

5.1.3 Observations

As shown in chapter 4, TextWorld can provide us information in a more structured way, giving information about admissible commands, intermediate reward, inventory and even path to the solution.

This information is stored in an environment object (env) wich Textworld provides in every game created. It is possible requesting for this information to passing specific flags on environment creation, the dict returned contains the following fields:

- `admissible_commands`: a list of commands that can be executed from the current state.
- `description`: a string of generic description of the scene (observation).

- `intermediate_reward`: it shows if the agent is on the right or wrong track upon completing the game. It equals +1 every time path to the solution becomes shorter, -1 if the agent goes away from the final game goal state and 0 if the if the distance to the solution remains the same. The objective of the model will be to maximize this variable.
- `inventary`: a string of description of things the agent is carrying.
- `policy_commands`: a list of commands we need to execute to get to the goal from the current state.

The agents will have access to all this information except `policy_commands` because to solve the problem with this information is trivial. Both models will be fed with the same environmental observations to ensure a comparison under the same conditions.

5.1.4 Play function

The play function is used to perform the experiments in a structured way.

This function has the following functions: manage the environment, extract the information provided by Textworld and make a pipeline between the agent and this information. It also execute the actions decided by the agent inside the environment.

In addition this function collects basic statistics such as number of movements and score.

```
def play(agent, path, max_step=100, nb_episodes=10, verbose=True):
                                ...
while not done:
            command = agent.act(obs, score, done, infos)
            obs, score, done, infos = env.step(command)
            nb_moves += 1
                                ...
```

5.2 Random agent

This model is the simplest one, it chooses a command from the admissible ones randomly. It will be used as a baseline, any other model should beat its performance.

```
class RandomAgent(textworld.gym.Agent):
                ...
        self.rng = np.random.RandomState(self.seed)
                ...
    def act():
        return self.rng.choice(infos["admissible_commands"])
```

5.3 Agents' architecture

In this section it will be shown the agents design. They receive the environment information in the form of strings and must make a decision within the available commands.

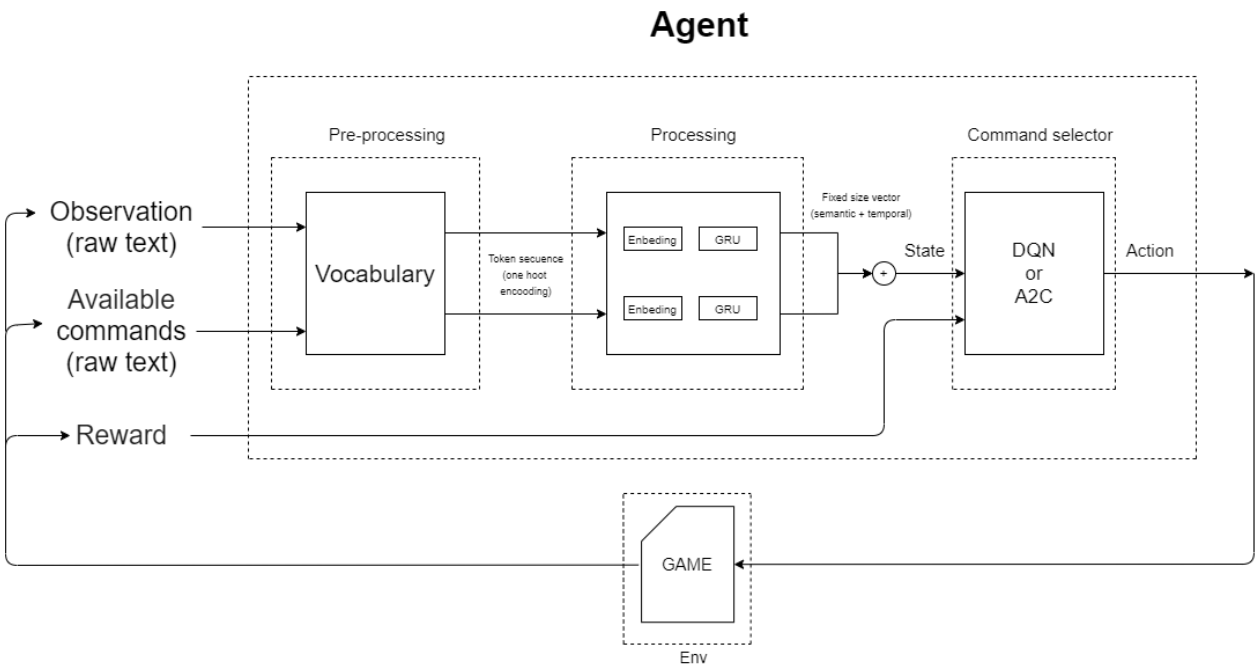

Figure 5.4: Models' Arquitecture.

To achieve that the agent design has these steps: text pre-processing where words are transformed into a sequence of perpendicular vectors. The processing phase where the perpendicular vectors are transformed into a fixed size vector sequence. This vector represents the current state of the game, it contains semantic and temporal information. And a final phase where the agent chooses an action based on the codified state and learn from their choices based on the reward. The different learning mechanisms will also be

explained.

Both models have the same two first stages, differing in the training method and the choice of commands.

5.3.1 Text pre-processing

The agents are not able to process strings so it is necessary to pre-process the information coming from the environment (observations) and transform them into vectors that model can understand. The Vocabulary class 5.5 will make this transformation.

The vocabulary contains a list of words and a single id. It will be filled when the agent finds an unknown (unlisted) word until it reaches a fixed size (`MAX_VOCAB_SIZE`).

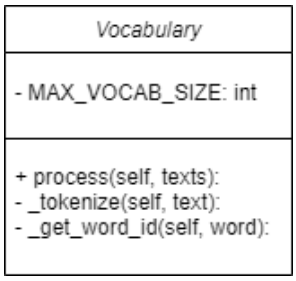

Figure 5.5: Vocabulary class.

The function process will collect information that comes as strings of characters, it will tokenize it filtering the punctuation marks and it will return a vector that associates the word and the position in vocabulary word's id (one-hot encoding).

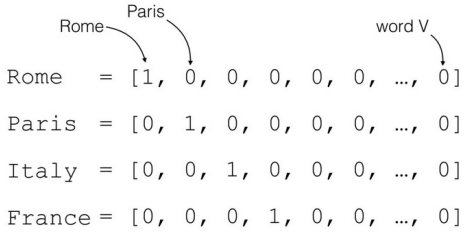

Figure 5.6: One hot encodding example.

5.3.2 Processing

The agent receives the observations and commands as strings, these pass through the pre-processing phase transforming it into token sequences, then through embeddings, this sequence is transformed into a dense vector sequence (in which the semantic content of the

words is coded). This is a variable size vector due to the fact that the size of the text is not fixed. This variable size vector is then inserted into the encoders to code the time information and to create a fixed vector size.

As a result of all these steps we are able to transform a string of characters taken as observation into a fixed size vector that has the semantic and temporal information of the text encoded, this defines the current state of the game. Finally, this information is received by the agent to perform an action.

Each of these phases will be explained in more detail in the following sections.

5.3.2.1 Embeddings

At the output of the pre-processing we have a vector of vocabulary size dimension for each world in the text, this is a sequence of vectors that encode each word. However there are words with similar semantics as "man" and "woman" that are very different from "table", this fact is what tries to represent this stage.

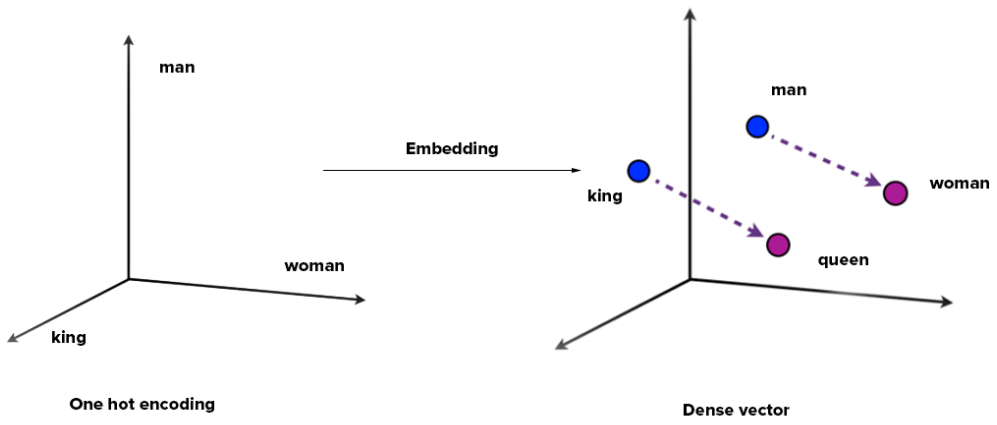

Figure 5.7: Word embeddings visualization.

After going through this phase we have a sequence of vectors smaller than the vocabulary size, now each word is not a perpendicular vector, the closer the semantics between words the closer these vectors are in a multidimensional space.

There are pre-trained models that perform this function as GloVe [22], however they are designed for larger vocabularies so we will choose to train our own word embedding.

It will be use pytorch's nn.Embedding module.

```
self.embedding    = nn.Embedding(input_size, hidden_size)
```

5.3.2.2 Temporal encoders

Communication through language is intrinsically temporary. It does depend on the total string of characters and not on the final word. Widely dispersed words in the timeline may have more weight in the information to be transmitted, in order to encode this information it will be used recurrent neural networks, specifically it will be used Gated Recurrent Unit (GRU).

The vector sequence coded by the embedding are received by the GRU encoder (`encoder_gru`), it is used to extract temporal features at the current game step. Those features are then provided as input to another GRU (`state_gru`) that serves as a state history and spans the whole episode.

The same idea applies to the temporary encoding of commands, another GRU encoder (`cmd_encoder_gru`) receives the available commands embedded and encode each command separately.

```
self.encoder_gru  = nn.GRU(hidden_size, hidden_size)
self.cmd_encoder_gru  = nn.GRU(hidden_size, hidden_size)
self.state_gru    = nn.GRU(hidden_size, hidden_size)
```

After going through this phase we have a fixed size vector, this phase normalize size for all situations. In this normalized vector the semantic and temporal content of the text is encoded and will be used as the state of the game.

5.4 DQN

We already have all the structure needed to train a DQN agent, they are pairs state (the temporary and semantic observations encoded) action (the output of the command selector) and a reward that textworld provides (1 if it approaches the solution and 0 if not).

$$Q^* : S \times A \to R \tag{5.1}$$

If the future reward is known for each action, choosing the best action is as simple as selecting the action witch returns the biggest reward. However, the reward is known after taking an action, so we do not have access to Q*. But, it is possible to resemble Q* using a neural network and training our agent according to the experiences it gets from playing and exploring the world.

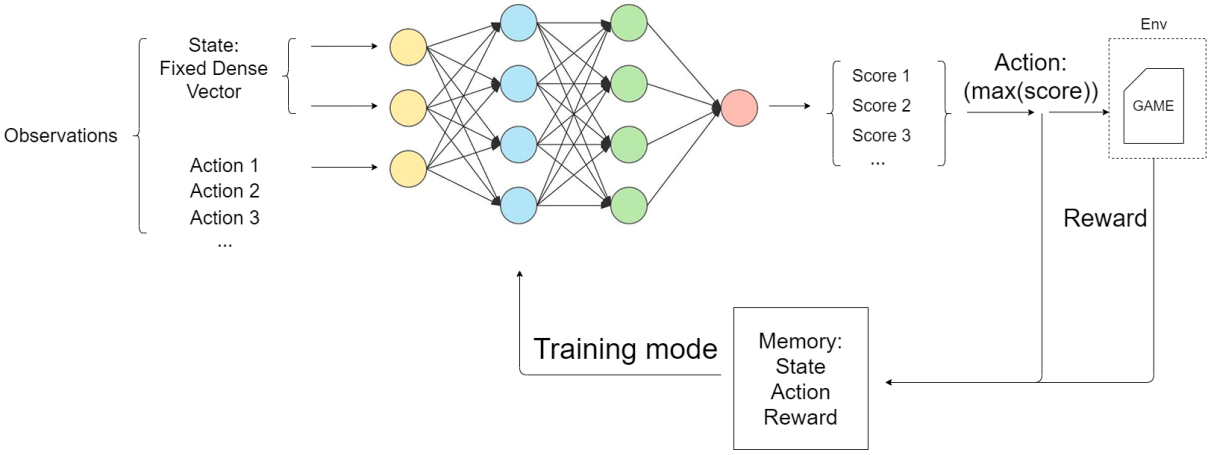

Figure 5.8: DQN Arquitecture.

The procedure to train the agent is as follows:

Initially it is necessary to ensure exploration of the environment and collect as much information as possible. To do this the first movements will be random and the probability of performing a random movement will decrease over time as convergence approaches. We will use `EPS_DECAY` to control the decline of random elections.

The agent will store the status, the action it took and the result (the reward) in memory, once he have enough samples, each `UPDATE_FREQUENCY` step, the model will be updated with the experience stored in the memory in such a way that minimize the mean square error between the Q(s,a) reward prediction and the real reward. The optimal policy defined by the Bellman equation is:

$$Q(s,a)^* = r_t + \gamma \max Q(s',a') \tag{5.2}$$

The difference between the two sides of the equality is known as the temporal difference error, δ :

$$\delta = Q(s,a)^{\pi *} - Q(s,a) \tag{5.3}$$

Learning consists of minimizing the mean square error of δ.

```
def compute_loss():
  ...
    state_action_values = policy_net(state_batch).gather(1, action_batch)
    expected_state_action_val = (next_state_val * GAMMA) + reward
    loss = (.5 * (state_action_val - expected_state_action_val) ** 2.).sum()
```

5.5 A2C

As shown in DQN agent, the Q value can be learned by parameterizing the Q function with a neural network. It is possible to use the Q fuction to build Actor Critic Methods. This method is composed of two parts, the actor who makes the decisions and the critic who measures how good these actions are.

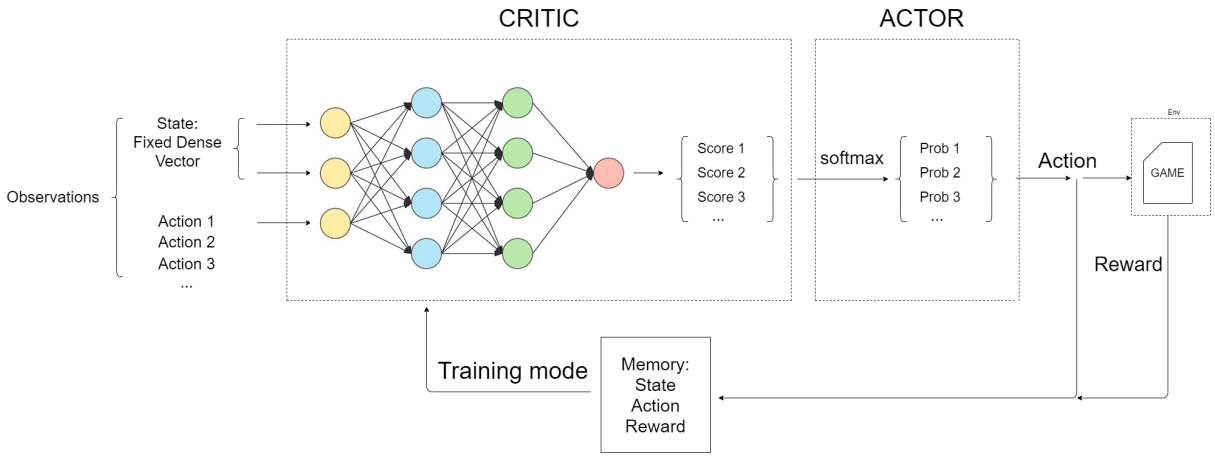

Figure 5.9: A2C Arquitecture.

A simple neural net is used to output a score for each pair state-action (observation-command). This score must be normalized and transformed into a probability using the softmax function, the selected command will be chosen based on this probability.

The actor must decide what action to take within all possible. The Critic estimates the value function and correct the actor's actions, to do this, A2C use the advantages.

Advantages are calculated subtracting the Q value term with the Value. This is the measure the critic will use to decide how much better it is to take a specific action compared to the average general action at the given state.

$$A(s_t, a_t) = Q(s_t, a_t) - V(a_t) \tag{5.4}$$

And the spected return that will be used to assemble Q correctly.

$$E_\tau = r_t + \gamma \max Q(s', a') \tag{5.5}$$

```
def _discount_rewards(self, last_values):
        R = last_values.data
        for t in reversed(range(len(self.transitions))):
            rewards, _, _, values = self.transitions[t]
            R = rewards + self.GAMMA * R
            adv = R - values
```

The critic scores the possible decisions and the actor decides what action to take, the most likely being the one that the critic gave the highest score. This allows the model to sporadically choose actions in which it predicts a bad result, promoting exploration.

Learning consists of minimizing policy gradient loss (what choice to make in each situation) and value loss (the predictions of how much reward the model will get).

$$Policyloss = \sum_{t=0}^{T-1} \nabla_Q log \pi_Q(a_t, s_t) A(s_t, a_t) \tag{5.6}$$

$$Valueloss = \frac{1}{2}(V(a_t) - E_\tau)^2 \tag{5.7}$$

The model also uses entropy regularization to promote the selection of more stochastic policies and to improve policy optimization.

```
def compute_loss():
  ...
        advantage        = advantage.detach()
                probs            = F.softmax(outputs_, dim=2)
                log_probs        = torch.log(probs)
                log_action_probs = log_probs.gather(2, indexes_)
                policy_loss      = (-log_action_probs * advantage).sum()
                value_loss       = (.5 * (values_ - ret) ** 2.).sum()
                entropy     = (-probs * log_probs).sum()
                loss += policy_loss + 0.5 * value_loss - 0.1 * entropy
```

CHAPTER 6

Experiments

In this chapter it will be explained the training and validation of our models. Also the models will be tested in the real world game.

6.1 Training

The training process will be similar for both models. The agents will be faced with 200 different games created by the "tw-make" command explained in point 4.2. The model will face each game 5 times with a batch size of 1000 steps. The following statistics will be collected to measure model training for each batch:

- Max score: it is a direct measure of the agent's performance in the game, it is the highest score achieved in the batch.
- Vocabulary: this is the number of words that the agent has collected, it does not imply that he has full semantic knowledge of each one however it is a measure of the agent's lexical range.
- Reward: this is a measure of the quality of the agent's learning. The agent must maximize this variable so it is a measure of the training process of the model.

The result of DQN and A2C agents after playing 200 and facing each other 5 times is as follows:

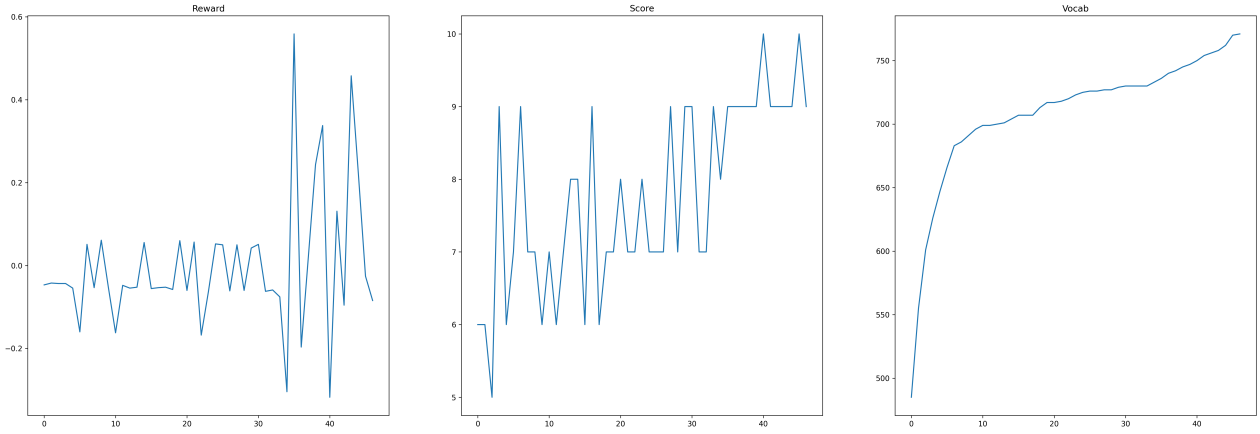

Figure 6.1: DQN model training results.

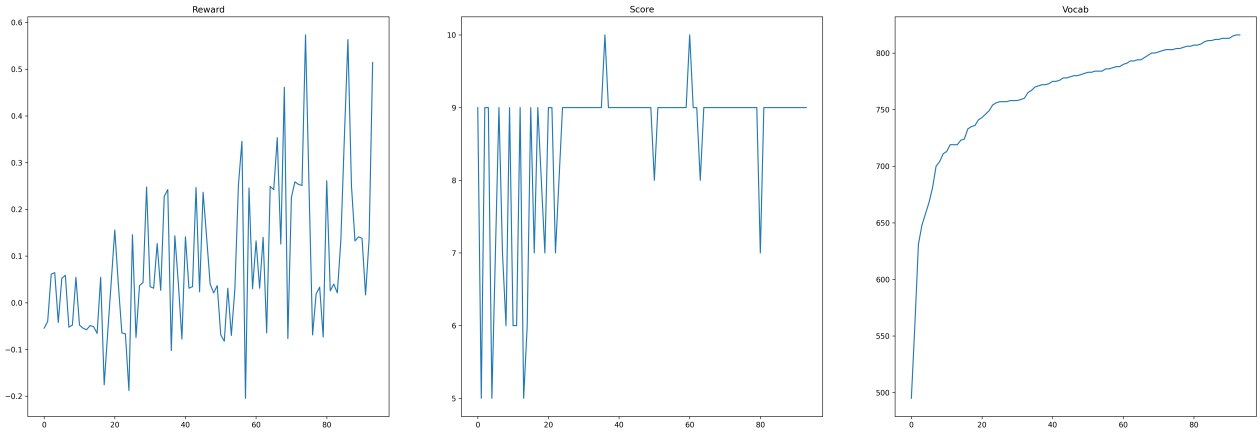

Figure 6.2: A2C model training results.

All indicators have a positive trend which may imply a correct training. However, a positive trend does not ensure correct learning, so it will be necessary to check the results of the following phases.

The DQN noise of the reward chart may reflect relative minimums when the agent reach a maximum in the punctuation for a concrete case, but when find a different situation the model leaves the relative minimum, entering a phase of low performance to later reach a more optimal minimum.

The A2C reward has a constant growth which indicates that the agent is being able to increase his reward over time and agent performance in the games improves. Although without the following phases study a possible overfeeding cannot be excluded.

The DQN score graph has a constant growth, which implies that agent performance in the games improves, although without the following phases study a possible overfeeding cannot be excluded.

The A2C score graph has a rapid growth at the beginning to stabilize. This may reflect a fast convergence of the model.

The vocabulary graphs grows very quickly at first, the agents do not know any word, and slows down as fewer new words appear. The DQN agent finish the training with a vocabulary of 771 words and the A2c with 816 words.

6.2 Validation

To ensure the generalization of the models we will measure their performance with another 20 games created in a similar way as explained in point 4.2. In this case the model has not faced any of these games in the training phase, they are completely new to the agents so we can measure the generalization of our models and detect possible overfitting.

Each model will face each game 5 times with a maximum limit of 200 steps.

Validation	A2C	DQN
Average steps	90.5	83.1
Average score	0.8	0.9

Since the games are completely unknown to the agent we can guarantee a certain level of generalization of the model and dismiss overfitting.

6.3 Test

Once the models have passed the training and validation phases, they can face the real problem, a game created to simulate a natural environment in the ETSIT school as explained in point 4.3.

This scenario is completely unknown for the models and the process of creating the game has no relation with the previous points process, this make this problem much more difficult for the agents. This also makes useless possible biases or strategies based on taking advantage of the random Textworld's game generation structure.

The agent will face the game 500 times to avoid statistical variations in performance and ensure convergence in results.

Test	A2C	DQN	Random
Average steps	135.8	158.2	185.9
Average score	0.7	0.5	0.2

It can be seen that the performance of DQN and A2C agents are far superior to a random choice of commands.

The A2C model is slightly higher than the DQN at this stage despite the fact that in the validation phase the result was higher

This could be due to the fact that the DQN agent tends to stand in relative minimum because it lacks ways to boost its exploration causing a worse generalization. On the other hand, the A2C model has mechanisms that promote the exploration of the environment, making it more generalizable.

This fact can be checked in the training phase by looking the number of words each vocabulary reaches. Each agent faces the same number of times the same games, so the higher the number of words, the higher the exploration of the environment.

CHAPTER 7

Conclusions and future work

In this chapter we will describe the conclusions extracted from this project and thoughts about future work.

7.1 Conclusions

The results obtained in this project have demonstrated the potential that reinforcement learning has within the world of natural language processing. In this project our models surpassed the performance of a random agent which means they were able to explore their environment, learn mechanics, understand text, identify its purpose automatically, increase their vocabulary and acquire a certain sense of temporal and semantic meaning of words in order to complete their mission.

We do not know what cognitive level they are compared to the average human, since the performance of different humans in the game has not been measured. However it is possible that they may not reach human performance but should not be too far.

In this project we have been able to train intelligent agents to reach a specific objective in a textual environment. These results open the door to possible implementations in chatbots

where it is desired to achieve a objective such as a loggin, where the objective is to reach the state in which all the necessary data has been provided by the user, or even the sale of a product where the final objective is to reach the state in which the client buys the product.

7.2 Achieved goals

As a result of this study, the following objectives have been achieved:

- Understand the uses and limitations of the Textwold learning environment.
- Study and analyze the state of the art of reinforcement learning and in particular of natural language processing and the resolution of text-based games.
- Train and test different agents using the state of the art algorithms.
- Create a custom game using TextWorld and use it as a benchmark for trained agents.
- Compare and measure the performance of algorithms.

7.3 Future work

There are many ways to improve this project but we can highlight 3 general lines:

- Analysis: the models are composed of many parts and it is not clear what impact each one has on the performance of the model. For this line we propose to improve the understanding of what mechanisms underlie the learning process. For this purpose, it is proposed to analyze in depth the data of each model and to extract new ones if necessary. By this way it could be found semantic relations of the vocabulary, possible situations that are difficult for the agents and to understand how the learning process works in depth.
- Model: in this line we propose to increase the size of the model and try different architectures. In general, increasing the size of the model generates better results by increasing the training time, one possibility would be to increase the size of each module and measure the performance. On the other hand, there are modules in the architecture that could be replaced by others of better performance. As a word embedding could be used pretrained models like word2vec or glove. For temporary

processing of sequences we could try to use LSTM. And for training, new agents like A3C could be tested.

- Applications: this section proposes possible applications of these results. The most obvious one is a chatbot where the user tries to interact in order to achieve an objective or reach a final state of the conversation.

APPENDIX A

Impact of this project

This appendix reflects, quantitatively or qualitatively, on the possible impact impact that the work carried out in this thesis may suppose.

A.1 Social impact

The results of this project could lead to more persuasive chatbots. Due to their conversational character, chatbots are potentially effective tools for engaging with customers, and are often developed with commercial interests at the core. However, chatbots also represent opportunities for positive social impact, they can make needed services more accessible, available, and affordable.

A.2 Environmental impact

The environmental impact is very reduced, the only impact that exists is that of the computers used, these are composed of polluting parts.

A.3 Ethical implications

Users need to know that the interactions they have with chatbots will remain private and secure. Chatbot responses, and all other communications, should also include some level of empathy and sensitivity when it comes to interacting with users.

In the worst case an advanced artificial intelligence might be able to persuade using bad techniques such as lie or manipulation.

APPENDIX B

Economic budget

This section exposes a detailed budget table of the whole project, covering aspects such as labor costs, material resources, and cloud computing. Most part of the final budget is derived from the developer salary.

B.1 Human resources

This section deals with the different economic aspects of human resources used in this thesis. The estimated salary for the developer is 15€ per hour.

The necessary hours to carry out the project have been around 320 hours, therefore the total earnings is 4,800 €.

B.2 Physical resources

The budget for the physical devices necessary for the development of this project is a computer whose minimum requirements allow the development the system. It consists basically of the personal computer.

Its features are: Ubuntu 18.04, 16GB RAM, Intel i7, 500GB HDD.

The price is around 750€.

B.3 Cloud computing

To train different models it may be necessary to use cloud computing, the price per hour for a c5.4xlarge AWS machine is: 0.65€/hour.

An estimate of 100 hours is 65€.

B.4 Licences Taxes

When it comes to licences, this project has been accomplished with Open Source Software (OSS), which means that they are free and it is no needed to pay for the use of any of the technologies.

Related to taxes we should take into account the Spanish legislation which establishes that a company must pay an extra 32.6 % of the employee's salary. This amount breaks down as follows:

- 23.6% for common contingencies.
- 5.5% for unemployment.
- 3.5% for possible work-related accident.

Bibliography

[1] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. A brief survey of deep reinforcement learning. *arXiv preprint arXiv:1708.05866*, 2017.

[2] OpenAI. A taxonomy of rl algorithms. url `https://spinningup.openai.com`. Accesed on 26-05-2020.

[3] Diksha Khurana, Aditya Koli, Kiran Khatter, and Sukhdev Singh. Natural language processing: State of the art, current trends and challenges. *CoRR*, abs/1708.05148, 2017.

[4] Marc-Alexandre Côté, Ákos Kádár, Xingdi Yuan, Ben Kybartas, Tavian Barnes, Emery Fine, James Moore, Matthew J. Hausknecht, Layla El Asri, Mahmoud Adada, Wendy Tay, and Adam Trischler. Textworld: A learning environment for text-based games. *CoRR*, abs/1806.11532, 2018.

[5] Brenden M. Lake, Tomer D. Ullman, Joshua B. Tenenbaum, and Samuel J. Gershman. Building machines that learn and think like people. *CoRR*, abs/1604.00289, 2016.

[6] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, USA, 3rd edition, 2009.

[7] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *CoRR*, cs.AI/9605103, 1996.

[8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[9] Ian Osband, Yotam Doron, Matteo Hessel, John Aslanides, Eren Sezener, Andre Saraiva, Katrina McKinney, Tor Lattimore, Csaba Szepezvari, Satinder Singh, et al. Behaviour suite for reinforcement learning. *arXiv preprint arXiv:1908.03568*, 2019.

[10] Kun Shao, Zhentao Tang, Yuanheng Zhu, Nannan Li, and Dongbin Zhao. A survey of deep reinforcement learning in video games. *arXiv preprint arXiv:1912.10944*, 2019.

[11] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017.

[12] 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.

[13] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *CoRR*, abs/1404.7828, 2014.

[14] Vijay Konda and John Tsitsiklis. Actor-critic algorithms. *Society for Industrial and Applied Mathematics*, 42, 04 2001.

[15] Vijay Konda and John Tsitsiklis. Onactor-critic algorithms. *SIAM Journal on Control and Optimization*, 42, 01 2000.

[16] Alla Kravets, Maxim Shcherbakov, Marina Kultsova, and Olga Shabalina. *Creativity in Intelligent Technologies and Data Science: First Conference, CIT&DS 2015, Volgograd, Russia, September 15-17, 2015. Proceedings*, volume 535. Springer, 2015.

[17] Wenpeng Yin, Katharina Kann, Mo Yu, and Hinrich Schütze. Comparative study of CNN and RNN for natural language processing. *CoRR*, abs/1702.01923, 2017.

[18] OpenAI. Gym. `https://gym.openai.com/docs/`. Accesed on 26-05-2020.

[19] Pytorch. Pytorch. `https://pytorch.org/docs/stable/index.html`. Accesed on 26-05-2020.

[20] Karl Pearson. The problem of the random walk. *Nature*, 72(1867):342–342, 1905.

[21] Microsoft. Textworld. `https://textworld.readthedocs.io/en/stable/`. Accesed on 26-05-2020.

[22] Stanford. Glove. `https://nlp.stanford.edu/projects/glove/`. Accesed on 26-05-2020.

[23] Oscar Araque. Design and Implementation of an Event Rules Web Editor. Trabajo fin de grado, Universidad Politécnica de Madrid, ETSI Telecomunicación, July 2014.

[24] J. Fernando Sánchez-Rada. Design and Implementation of an Agent Architecture Based on Web Hooks. Master's thesis, ETSIT-UPM, 2012.

[25] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. An analysis of deep neural network models for practical applications. *CoRR*, abs/1605.07678, 2016.

[26] Bruno González López. Codigo. `https://lab.gsi.upm.es/tfg/tfg-brunogonzalezlopez`, May 2020.