UNIVERSITÀ CATTOLICA DEL SACRO CUORE

Campus of Milano

Faculty of Economics, Mathematical, Physical and Natural Sciences

Master program in Data Analytics for Business

UNIVERSITÀ CATTOLICA del Sacro Cuore

Deep Reinforcement Learning and frame based explanations for videogames

Supervisor

Professor Marco Della Vedova

Dissertation by

Camilla Di Lorenzo

Id Number: 5005298

Academic Year 2021/2022

TABLE OF CONTENTS

TABLE OF FIGURES

INTRODUCTION

Reinforcement learning is a type of approach to machine learning (and artificial intelligence more generally) that has exploded in recent years and is enjoying considerable success in many application areas both in practice and in research and literature.

In simple terms, the technique consists of training an agent with artificial intelligence by making it perform repetitive actions and then rewarding it. The agent in the reinforcement learning experiment will perform various actions. If the actions are correct, the agent receives a reward. On the other hand, if it performs wrong actions, it receives a punishment. This will increase the learning capacity of the agent in performing the actions.

If one examines the detailed psychological definition of VR, one can understand it even more. The term reinforcement refers to something that increases the chances of progress in any task or action. According to this concept, learning with reinforcement means anything that helps improve behavior.

The purpose of this thesis is to study reinforcement learning algorithms in order to train an agent in the field of video games. In particular, a classic videogame was chosen, namely Super Mario Bros.

In Super Mario Bros, the agent must learn how to get through the level without getting stuck behind obstacles or crashing into any enemies. The agent will be trained in different ways to be able to achieve the best results.

Nevertheless, the aim of this thesis is not only to train the agent but also to try to provide explanations for the output of the model, i.e., to provide explanations based on the frames. In order to achieve such explanations, it is necessary to use explainable AI algorithms.

One might wonder why such algorithms are needed to understand a model. The answer lies in the fact that in the field of Artificial Intelligence, the interpretation of models is

often complicated. In fact, mathematical models in this field essentially provide numbers that are difficult to interpret. This is due to the fact that models are very complicated at their internal, to the point that it becomes impossible for a human being to understand the reasons behind their predictions. In technical jargon, this type of AI is called a “black box”, i.e., a black box that accepts incoming information, processes it in a completely obscure way for comprehension purposes, and provides an output that would be the prediction.

In cases where the consequences of an Artificial Intelligence's predictions could have an impact on a person's safety (e.g., medical diagnosis) or on his or her rights (e.g., access to mortgages, job opportunities), it is necessary to be able to answer why the machine made a certain decision. Indeed, it is natural to wonder what is inside this AI-generated magic number and whether it did not make a wrong prediction by focusing on inaccurate information.

In this thesis, therefore, after training the agent, an attempt will be made to implement an explicable AI algorithm to try to explain the choices of the neural network used by the reinforcement learning algorithm to train the agent.

In the implementation, problems will arise related to the fact that existing explainable AI algorithms are not adequately adapted to the field of reinforcement learning. Therefore, techniques and strategies will be tested in order to adapt Explainable Artificial Intelligence to the problem at hand.

In the end, the results of the training will be presented, giving information on which approach is best to train the agent, and explanations of the methodologies used to obtain the explanations will also be given, along with some useful learning comparisons.

1. REINFORCEMENT LEARNING

Reinforcement learning is one of the three branches of Machine Learning that allows the creation of autonomous agents capable of selecting the actions. Such actions are carried out to achieve objectives through interaction with the environment in which the agent operates.

Unlike the other two branches (supervised and unsupervised learning), reinforcement learning involves sequential decisions. In reinforcement learning, the action to perform depends on the system's current state and influences the future one. Furthermore, in reinforcement learning, programs are not fed with data.

The concept of reinforcement inspires the reward to which a positive numerical value is attributed to encourage correct behavior. In general, the technique is able to perceive and interpret a given environment in which the autonomous agent operates, choosing the actions and learning through trials. In fact, a trial-and-error strategy generates the data during training and, at the same time, marks it with a label (the action).

Reinforcement learning assigns positive values to actions to be encouraged and negative values to behaviors to be discouraged. This program allows the agent to seek a long-term reward and maximum gratification to provide the optimal solution. The program carries out several training cycles within a simulation environment until an exact result is provided.

This training aims to allow artificial intelligence to autonomously solve very complex control problems without any manual input from humans.

The policy, i.e., the behavior learned by the agent, can be represented in a Q table, where the Q stands for Q-learning algorithm, and it comes from the Q function, which computes the expected reward of action in the state in order to create the best policy. The rows of the Q-table contain all possible observations, while the columns include all possible actions. Therefore, the cells are filled during training with values representing the expected reward.

The Q table works only with small observation spaces. When the dimensions of the spaces increase, the agent must use a neural network. The use of deep neural networks as approximators for reward function or policy in reinforcement learning algorithms introduces the concept of deep reinforcement learning. Deep reinforcement learning combines deep learning and reinforcement learning.

Deep learning uses an enormous amount of data to learn complex models through computational advances and training techniques. While reinforcement learning perceives and interprets the environment in which the agent is immersed, the objective is to make the agent perform sequential actions learned through the reward mechanism (to encourage the performance of virtuous actions) and penalties (to discourage unwanted actions).

1.1 How Reinforcement Learning works

Reinforcement learning helps an agent to learn behavior through repeated interactions of the type "trial-and-error" with a dynamic environment.

The agent's learning comes from its continuous interaction with the environment. Specifically, the agent obtains a state from the environment and based on the actions the agent takes, the state itself changes. The environment is defined as everything that surrounds the agent and with which it can interact.

At a given time step t, both the agent and the environment are in a state s, which contains all the relevant information on the situation, such as the position of an object. Then, from the state s can be executed an action a, obtaining a couple of state-action which can be part of either a discrete set of state-action pairs or a continuous one. Before moving to the next time step $t + 1$, the agent receives a reward that is transferred to the next state. The policy allows the selection of the action to choose in a given state.

The policies can be deterministic when the same action is undertaken for a given state or stochastic when the action is chosen based on some distribution between actions and states. *Figure 1* recapitulates the functioning of the reinforcement learning algorithms.

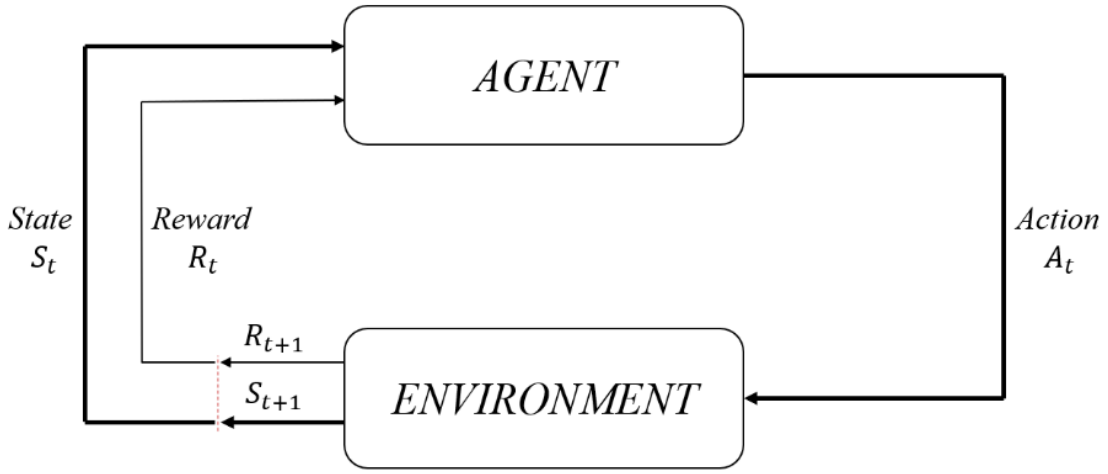

Figure 1 Reinforcement Learning scheme

The agent is not told what action to take; instead, it must find which action will lead to a greater reward simply by exploring and exploiting all the possible actions.

One of the reinforcement learning challenges is indeed the trade-off between exploration and exploitation. In order to get more or higher rewards, the agent must prefer actions already performed in the previous time steps and for which it has obtained good rewards. However, to find out these actions, it should try actions that it has never performed before. So, the agent must exploit what it learned to obtain the reward, but at the same time, it must explore to select better actions in the future.

Finally, it is possible to recap the fundamental steps to follow in order to solve a reinforcement learning problem.

In general, it is necessary to define the activity that the agent must learn, which also includes understanding how the agent interacts with the environment and the possible goals the agent must achieve.

Then, it is important to create the environment in which the agent operates. This step also includes the interface between agent and environment and the dynamic model of the environment. Therefore, it is necessary to specify the reward signal that the agent uses to measure its performance against goals. In addition, it is also required to specify how the environment calculates this signal.

After defining the environment and the reward signal, it is possible to proceed with the creation of the agent, its training, and its evaluation. The validation of the agent aims to evaluate the performances of the trained agent by simulating the agent and environment simultaneously.

1.2 Elements of Reinforcement Learning

Beyond agents and the environment, there are four other elements that play a key role in reinforcement learning programs. These elements are the policy, the reward signal, a value function, and a model for the environment.

The policy defines the way in which the agent learns and its behavior at a given moment. For ease of understanding, a policy can be seen as the perceivable states of the environment and the actions to be taken in such states. In some cases, the policy can be expressed as an easy function or lookup table; in others, it can involve extensive computation. The policy is the heart of a reinforcement learning agent in the sense that it alone determines its behavior. In general, policies can be deterministic, that is, only depending on the state, or stochastic, which is defined through the probability distribution of the actions given a state.

A reward signal defines the goal in a reinforcement learning problem. At any given time step, the environment sends a single value, also called a reward, to the agent. The only objective of the agent is to maximize the cumulative reward received during one episode of the program. So, the reward signal defines the good or bad behavior of the agent. This signal is the starting point to change the policy appropriately; if the selected action from the policy has a low reward, then the policy could change to select another action in a future situation. In general, the reward signal can be a stochastic function of the state of the environment and the action taken.

While the reward provides information on what is immediately good, the value function specifies what is good in the long run. The value of a state is the total amount of reward an agent is expected to store in the future, starting from that state. So, the value determines the long-term desirability of the states after considering the best states to follow and the possible reward in these states. For instance, a state could always bring a

low immediate reward, but it has a high value as it is regularly followed by other states that bring high rewards and vice versa.

The rewards can be defined as primary elements while the values as secondary elements. Without rewards, there can be no values, and the only aim of estimating values is to achieve higher rewards. However, values are the elements of comparison used to select and evaluate decisions. Actions are selected in such a way that they produce states of maximum value, not maximum reward, because these actions get a greater reward in the long run. Unfortunately, it is much more challenging to calculate values than to determine a reward. The rewards are basically given directly from the environment, while the values must be estimated and re-estimated by the sequences of observations that an agent performs during its entire existence. Indeed, the most important component is a method of estimating values efficiently, even if this reasoning is not said to be valid for any environment.

The fourth and last element is the model of the environment. The model means an entity capable of simulating the behavior of the environment. For instance, given a state-action pair, the model can predict the result of the next state-action pair. Models are used to plan, that is, to decide in advance which actions will be performed according to the reachable states.

To wrap up all elements explained until now and their roles, look at *Figure 2*.

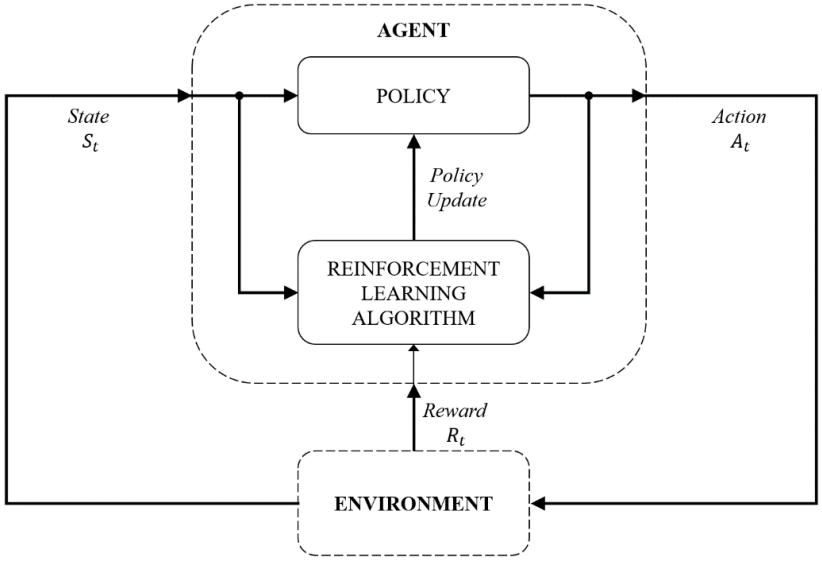

Figure 2 Detailed RL scheme

1.3 Markov decision process

In mathematics, a Markov decision process (MDP) is a discrete-time stochastic process; MDPs provide a mathematical framework for modeling the decision process in situations in which results are partly random and partly under decision control.

MDP is an extension of the Markov Chain. The difference between the two lies in the introduction of actions in Markov decision processes, which allow for choice, and rewards, which then allow giving motivations. If there is only one action for each state and the reward is fixed, i.e., always the same, the Markov decision process reduces to a Markov chain regardless of state and action. If the states and actions spaces are finite, the problem is called finite Markov decision process, and finite MDPs are particularly important for the reinforcement learning theory.

Let us now understand how a Markovian process works. At each time step the process is in some state s, and the decision-maker can choose any available action a. The process responds to the next time step by moving randomly in a new state s', and giving a corresponding reward, $R_a(s, s')$. The probability that the process evolves into the new state s' is affected by the selected action; specifically, it is given by the state transition function $P_a(s, s')$. For this reason, the next state s' depends on the current state s, and the decision a is taken by the decision-maker.

However, in light of the above, it is crucial that, given a and s, there is conditional independence from all previous actions and states, namely, the state transitions of a Markov decision process satisfy the Markovian property.

A Markov stochastic process is defined as a random process in which the probability of transition that leads to a system state depends only on the state of the system immediately preceding it and not on how it came to this state.

A Markov process can be described by means of the enunciation of Markov's property, or condition of "no memory", which can be written as:

$$P(X_{n+1} = j \,|X_n = i_n, \ldots, X_0 = i_0) \; = P(X_{n+1} = j \,|X_n = i)$$

A Markov chain is a Markovian process with a discrete state space, so it is a stochastic process that takes on values in discrete space and satisfies Markov's property. The set of states space S can be finite or infinite countable; in the first case, it is called a finite-state Markov chain. A Markov chain can then be time-continuous or time-discrete, depending on the T set of the time variable (continuous or discrete).

A Markov decision process is defined by:

- A state space S.
- An action space $A = U_s \in_s A(s)$ that can be undertaken according to the state.
- The transition probabilities $P_a(s, s'): S \times A \times S \to \mathbb{R}$ define the one-step dynamics of the environment, that is, the probability that, given a state s and an action a at time t, the next possible state reached is s':
$$P_a(s, s'): Pr\left(s_{t+1} = s'^{|s_t=s, a_t=a}\right).$$
- The expected value of the reward $R_a(s, s')$: given a state s and an action a, when switching to the state s', the resulting reward is equal to
$$R_a(s, s') = R\left(s'^{|s,a}\right) = \mathbb{E}(R_{t+1}|s_t = s, a_t = a, s_{t+1} = s').$$
- $\gamma \in [0, 1]$ is the discount factor representing the difference between future and present rewards.

Notice that the theory of the Markov decision process does not specify that S or A are finite, but the basic algorithm assumes that they are.

The purpose of Markov decision processes is to identify the best action to perform in a given state in order to get the maximum possible value of a cumulative reward function. The function that for each state $s \in S$ identifies the action $a \in A$ to perform is called stationary policy $\pi: S \mapsto A$. Typically, this function that evaluates the reward is the expected value of an infinite discounted sum, that is:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{a_t}(s_t, s_{t+k})$$

where $a_t = \pi(s_t)$ are the actions given by the policy π.

1.4 Value function

When deciding what action to take at a specific moment, the agent needs to know how "good" it is to be in a particular state. A way to measure this goodness of the state is the value function. It is defined as the expected sum of rewards ($\mathbb{E}_\pi$) that the agent will receive while following a particular policy π starting from a particular state s. The value function value, $V_\pi(s)$ for the policy π is given by:

$$V_\pi(s) = \mathbb{E}_\pi[\, G_t | s_t = s] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k \, R(s_t, s_{t+k}) | s_t = s \right]$$

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

$$Q_\pi(s, a) = \mathbb{E}_\pi[\, G_t | s_t = s, a_t = a] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k \, R(s_t, s_{t+k}) | s_t = s, a_t = a \right]$$

1.5 Bellman's equations

Bellman's equations formulate the maximization problem of the expected sum of the rewards in terms of the recursive relationship with the value function. A policy π is considered better than another policy π', if the expected return of that policy is greater than π' for all $s \in S$, which implies $V_\pi(s) \geq V_{\pi'}(s)$ for all $s \in S$. Therefore, the optimal value function $V^*(s)$ can be defined as

$$V^*(s) = \max_\pi V_\pi(s), \forall s \in S$$

Similarly, the action-value optimal function $Q^*(s, a)$ can be defined as

$$Q^*(s, a) = \max_\pi Q_\pi(s, a), \forall s \in S, \forall a \in A$$

Moreover, for an optimal policy, it is possible to define the following equation

$$V^*(s) = \max_{a \in A(s)} Q_{\pi*}(s, a)$$

If the transition probabilities and the reward function are known, the Bellman equations can be solved in an iterative way. This approach is known as dynamic programming.

The algorithms assuming these probabilities as known are said model-based. However, for many more algorithms, these probabilities are assumed unknown and are estimated by applying one or more simulation steps to test possible consequences. These algorithms, instead, are said model-free.

1.6 Algorithms

In the generic learning problem, even neglecting the exploration problem and even if states are observable, there remains the problem of figuring out how to use past experience to determine which actions lead to the greatest cumulative rewards.

In the following subsections, let us explain how to approach the learning problem.

1.6.1 Optimality Criterion

The agent's selection of the action is modeled as a map, called a policy:

$$\pi: A \times S \longrightarrow [0,1], \qquad \pi(a,s) = \mathbb{P}(a_t = a, s_t = s).$$

This equation gives the probability of undertaking the action a when in the state s. There also exists a non-probabilistic policy.

The value function $V_\pi(s)$ is defined as the expected reward starting from the state s, that is $s_0 = s$, and subsequently following the policy π. So, approximately, the value function estimates "how good" it is to be in a specific state.

Therefore, the algorithm must find a policy that gives the maximum expected reward. From Markov decision process theory, it is known that the search can be restricted to the set of so-called stationary policies without loss of generality.

A policy is stationary if the distribution of actions it returns depends only on the last state visited (relative to the agent's observation history).

The search can be further narrowed to deterministic stationary policies: such a policy selects actions based on the current state deterministically, i.e., it always repeats the same choices, given where it is; there is no stochasticity. Since any of these policies can

be identified through a mapping from the set of states to the set of actions, these policies can be identified through such mappings without loss of generality.

1.6.2 Brute Force

The brute force approach (or exhaustive search) involves two steps:

- For each possible policy, sample the rewards while running.
- Choose the policy with the largest expected cumulative reward.

The problems with this method are that the number of policies can be large or even infinite and that the variance of the rewards can be wide, requiring many samples to estimate the return of each policy accurately. These problems can be ameliorated by assuming a particular structure and by allowing the samples generated by one policy to influence the estimates made for the others. The two main approaches are the value function estimation and the direct policy search.

1.6.3 Model-free methods

In model-free models, policy and value function should be estimated, through the system's roll-out, or by applying one or more simulation steps to test the possible consequences. Many model-free approaches try to learn the value function and infer the optimal policy from it or by searching for the optimal policy directly in the policy parameters space. These approaches can be classified as on-policy or off-policy approaches.

On-policy methods use the current policy to generate actions and use it to update the policy itself. In contrast, off-policy methods use a different policy to generate actions than the updated policy.

1.6.4 Monte Carlo methods

Monte Carlo methods are used in algorithms that are based on GPI, i.e., the Generalized Policy Iteration. The generalized policy iteration is an iterative scheme consisting of two steps: the policy evaluation step and the policy improvement step.

The policy evaluation step uses Monte Carlo methods. Thus, given a stationary and deterministic policy π, the goal is to compute the value function $Q_\pi(s,a)$ (or an approximation) for all the state-action pairs (s,a).

Assuming (for simplicity) that:

- The MDP is finite
- There is enough memory available to hold the action values
- The problem is episodic
- After each episode, the next one starts from some random initial state

Then, the estimate of the value of a given state-action pair (s,a) can be computed through the average of the sampled states. So, given sufficient time, this procedure can give a precise estimate Q.

In the policy improvement step, the successive policy is obtained by developing a greedy policy with respect to Q: given a state s, this new policy returns an action that maximizes $Q(s,\cdot)$.

This procedure has some drawbacks. Firstly, it could be too time-consuming to evaluate a sub-optimal policy. It also uses samples inefficiently due to the fact that a long trajectory only improves the estimation of the single state-action pair from which the trajectory itself begins. Furthermore, when the rewards along the trajectory have high variance, the convergence is low. Finally, the method only works when problems are episodic and for small and finite Markov decision processes.

1.6.5 Temporal Difference methods

Temporal Difference (TD) methods are also built on the idea of GPI, but they differ from Monte Carlo methods in the policy evaluation step. Instead of using the total sum of rewards, these methods calculate the temporal error, that is, the difference between the new estimate and the old estimate of the value function, considering the reward received at the current time step and using it to update the value function. The update equation of the value function is given by:

$$V(s) \leftarrow V(s) + \alpha[r + \gamma\, V(s') + V(s)]$$

where α is the learning rate, r is the reward received at the current time step, s' is the new state, and s is the old state. Therefore, the TD methods update the value function at each time step, unlike the Monte Carlo Methods, which wait for the completion of the episode to update the value function.

1.6.6 Q-Learning algorithm

Q-Learning is an off-policy temporal difference algorithm, and it is in the class of value-based algorithms. This class of algorithms updates the value function based on an equation (particularly Bellman's equations). Moreover, it is off-policy since it approximates Q^* directly regardless of the policy it follows. Q is a table storing state-action pairs, and it is also called Q – *table interchangeably*.

An experience of the algorithm is defined as (s, a, r, s') where the agent starts from state s, carries out the action a, receives a reward r, and moves to a new state s'. The update on $Q(s, a)$ is given by receiving the maximum possible reward from action from s' and applying the update:

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} \underbrace{\alpha}_{\text{learning rate}} \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)}^{\text{temporal difference}}.$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{new value (TD target)}}$$

Where r_t is the reward when shifting from state s_t to state s_{t+1}, $\alpha \ni]0, 1]$ is the learning rate, and $\gamma \ni [0, 1[$ is the discount factor.

Finally, the algorithm can be written as follows (*Figure 3*):

Algorithm 1 Q-Learning

Initialize randomly $\boldsymbol{Q(s, a)}$

repeat

Observe the initial state $\boldsymbol{s_t}$

for $\boldsymbol{t = 1}$ **to** $\boldsymbol{T}$ **do**

Select an action $\boldsymbol{a_t}$ using the policy derived from $\boldsymbol{Q}$ (e. g. epsilon greedy)

Carry out action $\boldsymbol{a_t}$

Observe the reward r_t and the new state s_{t+1}

Update Q

end for

until the end

Figure 3 Q-Learning algorithm

The more straightforward way for storing the values of the value function is to group them in the Q-table, even if this method has some limitations.

If the state space of the problem is vast, it is impossible to save all values in the table format. The reason is very simple: the memory required to save all this data is too large. Moreover, even the research of a value in a particular state in the table could be computationally expensive.

Another limitation is due to the space of states: if the space is continuous, it will be impossible to use the tabular form unless the states themselves are discretized. For this reason, the adoption of the table format is applied only to environments with a reduced number of states and actions.

In order to overcome these issues, function approximators were introduced to save the value function. The value function is parameterized by a vector $\theta = (\theta_1, \dots, \theta_n)^T$, and it is indicated as $V(s; \theta^V)$. The function approximator can be thought of as a mapping between the vector θ in $\mathbb{R}^n$ with the space of the value function. As long as the number of approximator parameters is less than the number of state values, a given parameter's value changes the value function in multiple regions of the space of states. This helps the function approximators to make a better generalization in fewer training steps.

Various methods exist in Reinforcement Learning for function approximation, like Deep Q-Network (DQN) algorithm.

1.6.7 Deep Q-Network algorithm

Deep Q-Network is a reinforcement learning method of function approximation. It represents the evolution of Q-Learning, where a neural network substitutes the table state-action. Therefore, in this algorithm, learning does not consist of updating the table

but consists of adjusting the weights of the neurons that compose the network through backpropagation.

The learning of the value function in Deep Q-Network is thus based on changing the weights according to the loss function:

$$L_t = \left(\mathbb{E}\left[r + \max_a Q(s_{t+1}, a_t) \right] - Q(s_t, a_t) \right)^2$$

where $\mathbb{E}\left[r + \max_a Q(s_{t+1}, a_t) \right]$ represents the expected optimal reward while $Q(s_t, a_t)$ is the value estimated by the network.

Then, the errors computed by the loss function are backpropagated in the neural network through a backward step (backpropagation), following the gradient descent logic. Indeed, the gradient indicates the direction with the greatest growth of a function. Thus, moving in the opposite direction reduces the error. The behavior of the policy is given by an ϵ – greedy approach, to ensure a sufficient exploration. *Figure 4* shows the first version of the algorithm.

Algorithm 2 Deep Q-Learning or Deep Q-Network (DQN)

Initialize $\boldsymbol{Q(s, a)}$ with random weights

Repeat

Observe an initial state $\boldsymbol{s_1}$

for $\boldsymbol{t = 1}$ **to** $\boldsymbol{T}$ **do**

Choose an action $\boldsymbol{a_t}$ from the state $\boldsymbol{s}$ using the $\boldsymbol{\epsilon}$ – greedy policy derived from $\boldsymbol{Q}$

Carry out the action $\boldsymbol{a_t}$

Observe the reward $\boldsymbol{r_t}$ and the new state $\boldsymbol{s_{t+1}}$

Compute the target $\boldsymbol{T}$

if $\boldsymbol{s_{t+1}}$ is the terminal state **then**

$\boldsymbol{T = r_t}$

Else

$\boldsymbol{T = r_t + \gamma \max_a Q(s_{t+1}, a_t)}$

end if

Train the network $\boldsymbol{Q}$ minimizing $\left(\boldsymbol{T - Q(s_t, a_t)}\right)^2$ as loss

end for

until the end

Figure 4 Deep Q-Learning algorithm

When using the algorithm in *Figure 4*, it can be seen that the approximation of the function is not stable. So, in order to obtain convergence, it is appropriate to modify the basic algorithm by introducing techniques to avoid fluctuations and divergences.

The most important technique is called experience replay. With this technique, the experience of the agent $e_t = (s_t, a_t, r_t, s_{t+1})$ is taken at each time step t and stored in a dataset $D = e_t, e_{t+1}, \ldots, e_n$ called replay memory. The training is performed using a mini-batch technique, i.e., by taking a sub-set of experience samples randomly extracted from this replay memory. The use of this technique allows the exploitation of past experiences in more than one network update. In addition, the randomly chosen subset from the replay memory allows breaking the strong correlation present between successive experiences, thus reducing the variance between different updates.

Algorithm 3 Deep Q-Network (DQN) with Experience Replay

Initialize the Replay Memory $\boldsymbol{D}$
Initialize $\boldsymbol{Q(s, a)}$ with random weights
Repeat
  Observe an initial state $\boldsymbol{s_1}$
  for $\boldsymbol{t = 1}$ **to** $\boldsymbol{T}$ **do**
    Choose an action $\boldsymbol{a_t}$ from the state $\boldsymbol{s}$ using the $\boldsymbol{\epsilon}$ – greedy policy derived from $\boldsymbol{Q}$
    Carry out the action $\boldsymbol{a_t}$
    Observe the reward $\boldsymbol{r_t}$ and the new state $\boldsymbol{s_{t+1}}$
    Store the transition $(\boldsymbol{s_t, a_t, r_t, s_{t+1}})$ in $\boldsymbol{D}$
    Sample transitions $(\boldsymbol{s_j, a_j, r_j, s_{j+1}})$ from $\boldsymbol{D}$
    Compute the target $\boldsymbol{T}$
    if $\boldsymbol{s_{j+1}}$ is the terminal state **then**
      $\boldsymbol{T = r_j}$
    else
      $\boldsymbol{T = r_j + \gamma\, max_a Q(s_{j+1}, a_j)}$
    end if
    Train the network **Q** minimizing $\boldsymbol{\left(T - Q(s_i, a_i)\right)^2}$ as loss
  end for
until the end

Figure 5 DQN algorithm with Experience Replay

1.6.8 Double DQN algorithm

Double Deep Q Network (DDQN) was created to address a known problem with the original algorithm, which is the overestimation of move values (Q-values). To do this, it uses two distinct evaluation functions learned from two separate neural networks, respectively, the online network with parameters θ and the target network with parameters θ^-. Afterward, set the calculation to identify the best action by decomposing it in two different operations:

- Evaluation of the $\boldsymbol{\epsilon}$ – greedy policy using the online network
- Estimation of the values using the target network

Finally, the scheme in *Figure 6* represents how it works.

Algorithm 4 Double DQN with Experience Replay

Initialize the Replay Memory $\boldsymbol{D}$

Initialize the online function $\boldsymbol{Q}$ with random weights $\boldsymbol{\theta}$ and the target function $\widehat{\boldsymbol{Q}}$ with weight $\boldsymbol{\theta^- = \theta}$

Repeat

Observe an initial state $\boldsymbol{s_1}$

for $\boldsymbol{t = 1}$ **to** $\boldsymbol{T}$ **do**

Choose an action $\boldsymbol{a_t}$ from the state $\boldsymbol{s}$ using the $\boldsymbol{\epsilon}$ – greedy policy derived from $\boldsymbol{Q}$

Carry out the action $\boldsymbol{a_t}$

Observe the reward $\boldsymbol{r_t}$ and the new state $\boldsymbol{s_{t+1}}$

Store the transition $(\boldsymbol{s_t, a_t, r_t, s_{t+1}})$ in $\boldsymbol{D}$

Sample transitions $(\boldsymbol{s_j, a_j, r_j, s_{j+1}})$ from $\boldsymbol{D}$

Compute the target $\boldsymbol{T}$

if $\boldsymbol{s_{j+1}}$ is the terminal state **then**

$\boldsymbol{T = r_j}$

else

$\boldsymbol{T = r_j + \gamma\, \widehat{Q}(s_{j+1},\ arg\, max_a Q(s_{j+1}, a_j; \theta);\ \theta^-)}$

end if

Train the network **Q** minimizing $\left(\boldsymbol{T - Q(s_i, a_i; \theta)}\right)^2$ as loss

end for

until the end

Figure 6 Double DQN algorithm

2. ARTIFICIAL NEURAL NETWORK

This section will present an overview of the main concepts of artificial neural networks. It is imperative to have an idea of how they work in order to understand the behavior of Deep Reinforcement Learning algorithms. In fact, as mentioned above, Deep Q Network and Double Deep Q Network algorithms use neural networks for Q-table representation.

2.1 From the biological neuron to the artificial neuron

Neural Networks represent a point of contact between different disciplines such as neurology, psychology, and artificial intelligence. The idea is to simulate the behavior of synaptic transmissions that occurs in the biological brain during the learning and information transmission phases through hardware and software devices. Neural networks are a branch of Machine Learning, and because of this, they have a self-learning approach that, as evidenced by the name itself, is inspired by the primary element by which the brain processes information: the neuron (Figure 7).

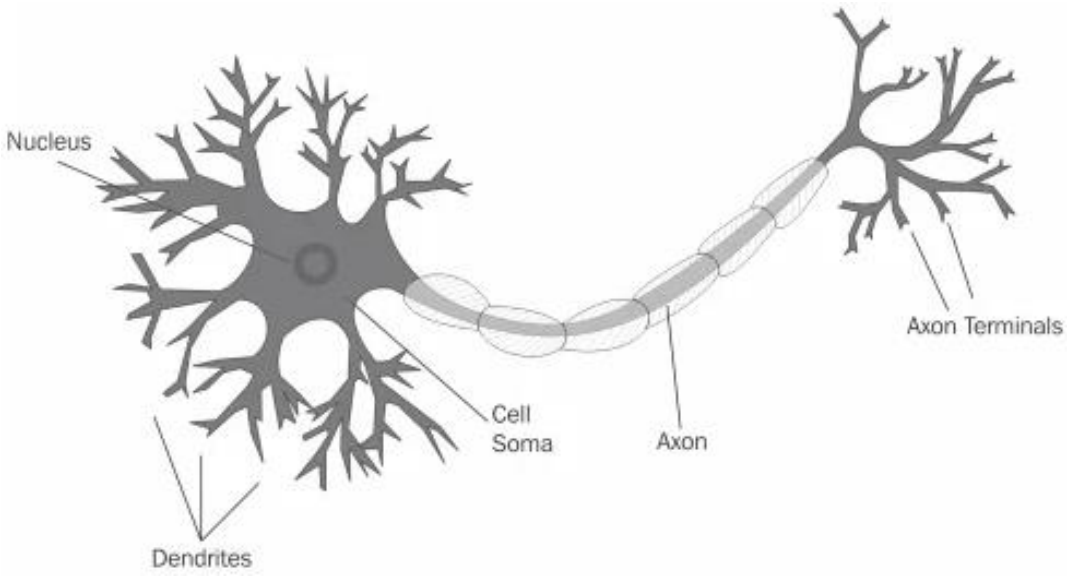

Figure 7 Biological Neuron

A neuron is characterized by a simple structure consisting of 3 essential elements:

- Cell body, in which is the nucleus that conducts all the neuron's activities.
- Axon, a single long fiber that transmits messages from the cell body to the dendrites of other neurons.
- Dendrites, fibers that receive messages from other neurons and send them back to the cell body.

Neurons receive electrical input from dendrites, absorb energy and release it via axons, the output channels, to other nearby neurons, transmitting the information in the form of an electrical impulse through specialized structures called synapses. Whether or not the signal favors activation of the receiving neuron, the synapse can be excitatory or inhibitory.

Only after a certain period of time, called refractory time, can the neuron generate a further impulse, showing the binary nature of this important yet simple biological element.

Similarly, an artificial neural network is formed by the interconnection of simple processing units, also known as artificial neurons, that have the propensity to extract knowledge from data and store it through weights, also called synaptic weights. Artificial neural networks differ from other methods in two important characteristics:

- They are designed with the ability to approximate any function, whether linear or non-linear. Linearity or non-linearity depends solely on the learning process and the activation functions, which can be either linear or non-linear.
- They can update synaptic weights during training. Samples are sent to the network, and the weights are changed. The modification of the weights is such that it reduces the minimum distance between the target (goal of the network) and the actual output. In this way, the network constructs an input-output mapping of the system from the samples it has trained with.

2.2 Model of the neuron

The neuron can be mathematically written as

$$y_k = \varphi\left(\sum_{j=0}^{p} x_j w_{kj} + b_k\right)$$

or as

$$y_k = \varphi(v_k + b_k)$$

where $x_1, \ldots, x_p$ are the input signals, b_k is the bias, $w_{k1}, \ldots, w_{kp}$ are the synaptic weights of the neuron k, w_{k0} is the weight related to the input $x_0 = 1$, i.e. the bias, $\varphi(\cdot)$ is the activation function, v_k is the action potential, and y_k is the output of the neuron.

The model of a neuron is formed from three base elements:

1. Weights and bias: connections (synapses) are made through weights, while the biases modify the effect of the activation function. In particular, the input signal x_j, where j represents the synapses associated with the neuron k, is multiplied by the weight w_{kj}, where the first subscript k refers to the neuron to which synapses are connected. The second subscript refers to the neuron from which the synapses originate. There is also a bias $b_k = w_{k0}$ (associated with the fixed input $x_0 = 1$), whose effect increases (if positive) or decreases (if negative) the net input of the activation function.
2. The sum operator computes a weighted sum of the input signals times the weights.
3. The activation function is used for processing the neuron's input and modulation of the output.

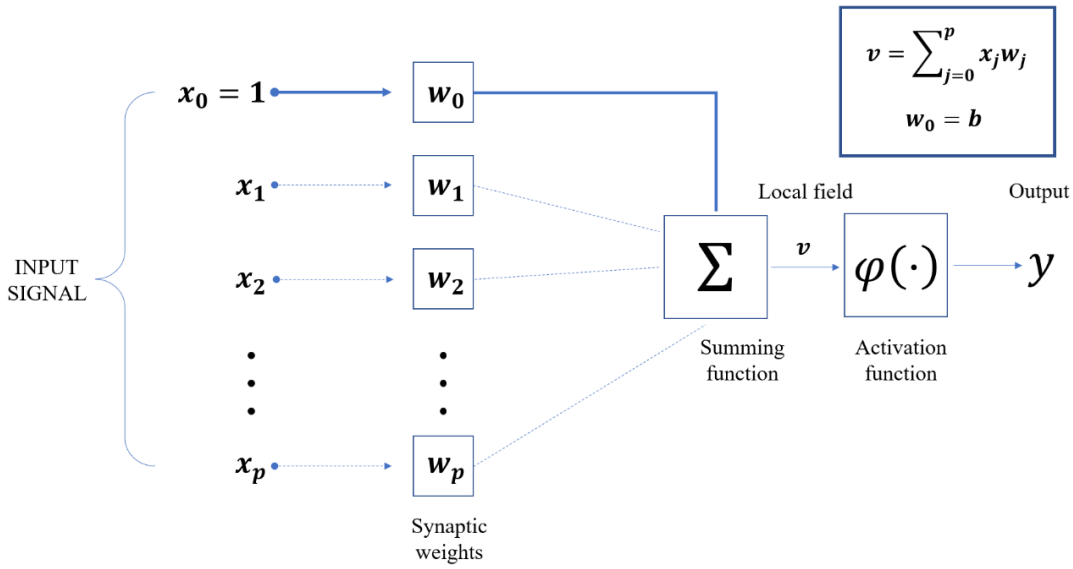

Figure 8 Example of Neural Network with input, output, weights, bias, sum operator, and activation function

2.3 Activation functions

The activation function $\varphi(v)$ are mathematical equations that determine the output of the neuron to which it is bound and contribute to the model's linearity and non-linearity. The function is attached to the neurons in the neural network. It indicates whether a neuron should be activated or not, based on whether each neuron's input is relevant for the model prediction. This function can also be used to normalize the output in the range $[-1, 1]$ or in the range $[0, 1]$.

The most commonly used activation functions are the following:

- Binary step function: it is described by

$$\varphi(v) = \begin{cases} 1, v < 0 \\ 0, v \geq 0 \end{cases}$$

  The advantage of this function is its simplicity and efficiency; however, it does not allow multi-value outputs, which means that it does not work very well for categorization.

- Linear step function in which the amplification is unitary in the linear region. It is described by

$$\varphi(v) = \begin{cases} 1, & v \geq +\frac{1}{2} \\ v, & -\frac{1}{2} > v > +\frac{1}{2} \\ 0, & v < -\frac{1}{2} \end{cases}$$

- The sigmoid function of which the best known and most widely used is the logistic function. In the logistic function, there is a parameter c that affects the slope, and the equation has values in the range $[0, 1]$.

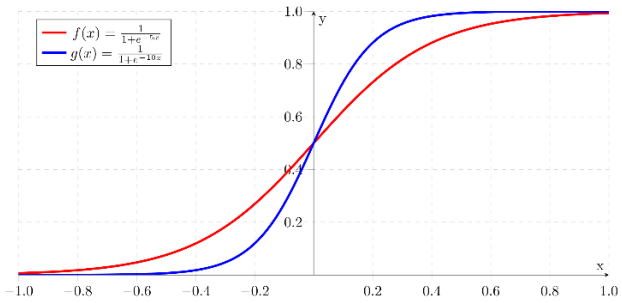

Figure 9 Sigmoid function for different c

It is described by

$$\varphi(v) = \frac{1}{1 + e^{-cv}}$$

The advantage of this activation function is that its smooth gradient manages to avoid jumps in output values. In addition, the output values are between 0 and 1, normalizing the output of each neuron. Finally, it provides clear predictions indeed, as can be seen in *Figure 9*, for X greater than 0.2 or less than -0.2, it tends to bring the Y value (the prediction) to the edge of the curve, i.e., very close to 1 or 0. However, it is very computationally expensive and its gradient, for very high or very low values of X, hardly changes at all in the prediction, causing a vanishing gradient problem.

- Hyperbolic tangent function or tanh is a function similar to the sigmoidal function and differs from it in being antisymmetric with values between -1 and 1. The equation is described by

$$\varphi(v) = \frac{1 - e^{-2cv}}{1 + e^{-2cv}}$$

- ReLU (rectified linear unit) is a smooth function defined as

$$\varphi(v) = \begin{cases} 0, v \leq 0 \\ max(0, v), v > 0 \end{cases}$$

The strengths of ReLu are that it is computationally efficient, thus allowing the network to converge very quickly and that it is non-linear. However, it looks like a linear function has a derivative function and allows backpropagation. One problem with the ReLU is that when the inputs are close to zero or are negative, the gradient of the function becomes zero, and therefore the network cannot perform the backpropagation and cannot learn.

2.4 Architecture

Neural networks are organized into layers. Each network has an input layer, an output layer, and a series of hidden layers. The purpose of the first layer is simply to send the inputs to the first hidden layer without any kind of processing. Neural networks are said to be fully connected if they are composed only of fully connected layers. A layer is said to be fully connected if all inputs of one layer are associated with all activation units of the next layer. Layers of this type are as generic as possible: given S neurons and N inputs, all inputs are sent to all neurons; there will be then S outputs. Each neuron is associated with N weights plus a bias because of the number of inputs it receives. *Figure 10* shows the schema of the network with this type of layer.

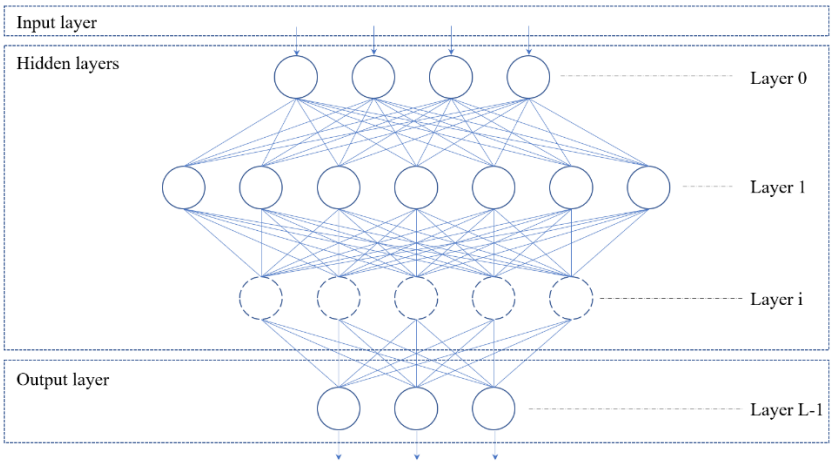

Figure 10 Fully connected neural network scheme

The weights associated with the inputs of each neuron are calculated through techniques such as gradient descent, which uses a cost function to minimize the error of the output values.

2.5 Convolutional Neural Network (CNN)

One of the most widely used neural networks for image processing is the convolutional neural network. It is generally composed of layers of different types, but the main characteristic of these networks is the presence of at least one convolutional layer. Whereas fully connected layers connect all inputs to all neurons, and thus all outputs depend on all inputs, convolutional layers use the so-called local connection. Each output only depends on the part of the inputs.

Convolutional layers work on data organized in several matrices of equal dimensions: this is ideal for images, for example, where each value corresponds to a pixel, and for each color (red, green, and blue), there is a different matrix, as shown in *Figure 11.* The data within these three-dimensional tensors are identified by a triplet of indices i, j, and k where i denotes the row, j the column, and k the matrix. Let us define H as the height of the volume (along i), W as the width (along j), and D as the depth (along k).

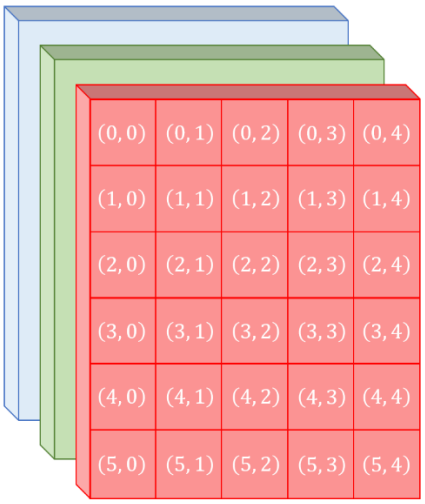

Figure 11 Convolutional layer image representation

The receptive field is a parameter that describes the extent of connectivity of neurons with inputs, thus indicating the size of the input portion that is used. The neuron contains as many weights as the amount of data in the receptive field. Its functioning is equal to one of the fully connected layers: it takes the inputs, multiplies them by weights, adds a bias, and finally applies an activation function. Each input is associated with a weight. Then, the weights within the neuron can be organized as a volume of values of the same size as the receptive field.

An assumption can be made: if the recognition of a specific feature occurs in a certain portion of the volume, then it is reasonable to assume that it can also occur in another portion. This implies that it is possible to move the receptive field along the height and width of the volume using the same neuron and the same weights. Clearly, these operations reduce the number of weights to be used. During output evaluation, the receptive field covers the entire volume. Each position corresponds to an output.

The parameter that indicates how far the field moves at a time is called stride. The shift along index i is not necessarily equal to the shift along j, so the stride parameter consists of two values.

In addition, there is another parameter called padding: it allows a frame of zeros to be added around the input matrices, which is very useful for detecting the characteristics sought even on the edges of the data volume. It consists of 4 values which indicate in order how many rows of zeros to add at the top, bottom, left, and right.

There are types of layers akin to convolutional layers that allow the reduction of the number of samples from one layer to another without the need to store additional weights. These layers are called pooling layers.

The operation of this type of layer is simple. As with convolutional layers, there are the receptive field, stride, and padding as parameters. The maximum value (max-pooling), minimum value (min-pooling) or average value (average pooling) is calculated independently for each of the input tensor matrices among those highlighted by the receptive field. The output will then have a vector composed of D values for each position of the receptive field. The dimensions of the output matrices are calculated in

exactly the same way as for the convolutional layers, while the volume depth is equal to that of the input volume ($H_{out} \times W_{out} \times D$).

3. EXPLAINABLE AI

Explainable AI (in brief, XAI) is a field of Artificial Intelligence aiming at explaining Machine Learning models to humans by using a variety of methods. These methods try to answer questions like: What is the model learning? Which part of the input is most important for a prediction?

Nowadays, AI systems and Machine Learning algorithms are widespread in many areas. Indeed, data is used almost everywhere to find solutions to problems and help humans. An essential factor for this success is the progress in the area of deep learning, but also, in general, the development of new creative ways of using data. As a result, the complexity of these systems becomes incomprehensible even to AI experts. That is why the models are usually referred to as black boxes. However, machine learning is also being applied in safety and health-critical environments such as autonomous driving or healthcare.

Therefore, humans need to understand what is going on inside the algorithms. This is precisely where explainable AI comes in. This research field provides techniques to understand better and validate how the machine learning models work.

Over the past two years, interest in this area has increased; understanding models is not only relevant for data scientists to build them, but especially for end users who expect explanations of why certain decisions are made. Therefore, transparency and interpretability can also be seen as a kind of user experience.

Usually, when building models with data, a trade-off between accuracy and interpretability can be observed. Neural networks, for example, often have millions of parameters that simply exceed human ability to understand. Therefore, there are generally two options which are: build interpretable machine learning models or derive human-understandable explanations for complex machine learning models such as for reinforcement learning models. In the literature, the former option is also called model-based, while the latter is called post-hoc.

Post-hoc methods can be further divided into black-box approaches and white-box approaches.

Black-box approaches mean that everything about the model is unknown, so it is focused on the relationship between input and output, and the response is to derive explanations.

For white-box approaches, there is the possibility of accessing model internals, that is, the possibility of accessing gradients or weights of a neural network.

The field of explainable AI involves the whole area of psychology about what makes good explanations and what kinds of explanations are best for humans.

3.1 Terminology

A few properties help distinguish the different types of methods in this field. Foremost, it is possible to differentiate between model-agnostic and model-specific explainable AI methods.

Model agnostic means the explainable AI algorithm can be applied to any kind of model, such as random forests, neural networks or support vector machines. On the other hand, model-specific means that the method was designed for a specific type of machine learning models such as only for neural networks.

Regarding the scope of explanations provided, methods can be classified into global and local approaches. This refers to the goal of explaining the entire model or only parts of the model, such as individual predictions. To understand this better, remember that the decision boundary of a complex model can be highly non-linear. In such cases, it does not make sense to provide explanations for the global model, so many approaches zoom in on a local area. They then explain individual predictions at that decision boundary.

In addition to the agnosticity and scope of a method, one can further differentiate based on the data type a method can handle. In fact, not all explainability algorithms can work with all types of data.

Finally, models can be distinguished by the type of explanation that is provided. There can be visual methods, e.g., returning correlation plots, but there can also be methods

that give information about the importance of variables. This is sometimes also referred to as feature summary.

Instead, other methods return data points that help understand the model better.

Finally, there are also approaches that build a simple model around a complex one. This simple model can then be used to derive explanations. These models are generally called surrogates.

3.2 LIME method

LIME is one of the most popular methods for explaining black-box models. LIME stands for a Local Interpretable Model agnostic Explanation. Since LIME is a model agnostic post-hoc method, it assumes no access to the underlying model. It treats the model as a black-box. This means it makes no assumptions on how the model is working, but it extracts the logic of the model by observing the outputs of the black-box in response to a large number of inputs. All the information the line explainer gleans from the black-box is established by querying the model and observing outputs.

The basic idea of LIME is to enlarge the local area of the individual prediction and then create a simple explanation that makes sense in that local region. In this way, it is possible to provide an explanation in a particular case without worrying about the explanations of the whole model. So, given an input X, LIME generates new input samples around it and observes the output of the model in response to these artificially created inputs. Then, LIME constructs a linear model that is locally faithful to how the model behaves in the sampled neighborhood.

The key idea of what LIME is trying to do is the following: the classification decision boundary can be globally complex and squiggly, but it is simple and linear in the local region.

Domain experts play an important role when dealing with explainable artificial intelligence algorithms. In fact, they are assumed to have prior knowledge of the problem, so if LIME's explanation is contrary to what the experts know, then it means that maybe there is something wrong with the developed model.

The paper introducing LIME is titled "Why should I trust you?" (Ribeiro, Singh, & Guestrin, 2016). The paper also reports that providing explanations improves the acceptance of a predictive algorithm. For LIME, the only requirement is that the explanations are locally faithful, but they might not make sense globally.

The mathematical idea behind LIME is to create a local approximation of the complex model for specific input.

$$\xi(x) = \underset{g \in G}{arg\,min}\, \mathcal{L}(f, g, \pi_x) + \Omega(g)$$

The formula denotes the complex model as f and the simple model as g (local model). The small model g is derived from a set of interpretable models that is denoted by G. The family G in the paper is set on sparse linear models.

The first loss term, $\mathcal{L}(f, g, \pi_x)$, in the optimization function wants to approximate the complex model f by the simple model g in the neighborhood of the data point x. In other words, the goal is to obtain a good approximation in that local neighborhood. The term π_x denotes the local neighborhood of data point x and is a kind of proximity measure.

The second loss term is used to regularize the complexity of the simple surrogate model.

In summary, the purpose is to look for a simple model g that minimizes the two loss terms: thus, it should approximate the complex model in the given local area and also remain as simple as possible.

In order to calculate the first loss term, it is necessary to start by randomly generating new data points in the neighborhood of the original one. These new points will be weighted by distance from the original point since the interest is only in the neighboring local area. The generation of the new points is done by perturbations, that is, by slightly changing some values of the original input. Afterward, these new data points are used to obtain a prediction using the complex model f finally obtaining a new dataset. This new dataset can be used to fit a classifier with the values obtained from the predictions of the complex model f as labels and the new data points as features.

Thus, minimization of the first loss term consists of obtaining the highest accuracy on the new data set using a simple linear model. For linear regression, for example, this

simply consists of minimizing the sum of the squared distances between the predictions and the ground truth. In the paper, the authors use the following loss function to optimize a linear model:

$$\mathcal{L}(f, g, \pi_x) = \sum_{z,z' \in Z} \pi_x(z) \left(f(z) - g(z')\right)^2$$

It represents the sum of square distances between the predictions of the complex model f (labels) and the predictions of the simple model g. In addition, proximity π_x is added to weight the loss function by how close a data point is to the original input. Points closer to the original data point are weighted more. In the paper, the authors use an exponential kernel as a distance metric that can be thought of as a heatmap.

The second term of the loss function $\Omega(g)$ is used to ensure that the model remains simple. LIME indeed uses a sparse linear model for g. The usage of sparse linear models allows the second loss term to be taken care of. Indeed, the sparse linear model aims to produce as many zeros as possible. In practical terms, this goal can be achieved by using some regularization techniques such as lasso linear regression. In this way, it is guaranteed to produce simple explanations with only the most relevant variables.

3.2 Shap method

The idea behind shap comes from the area of cooperative game theory. To understand the methods of shap, let us first understand the idea behind the area from which it comes.

A cooperative game is a competition between groups of people, also called coalitions. Coalitions participate in the cooperative game and receive a payout at the end. Since each member of the coalition contributes differently, dividing the payoff into the same parts may be unfair to some. The problem that cooperative game theory should solve is how to fairly allocate the gains to each player in the winning coalition. The solution is the Shapley values introduced in 1951 by Lloyd Shapley. Shapley values indicate the average contribution of each player in winning.

Explainable AI shap algorithm uses shapley values. In fact, cooperative game theory can also be applied in machine learning models. Just think of features as players in the game, prediction as the payoff, and the game as a machine learning model.

Thus, the basic idea is based on treating each feature as a player in a game and computing shapley values to find out their contribution in the black-box model. The central intuition behind the shapley values is to compare how the coalition would behave with versus without a specific player. In this manner, it is possible to find out how the excluded player contributed to the game. The idea behind Shapley values is to calculate a player's contribution for each subset and then average over all contributions; this gives a player's marginal contribution to the team.

Artificial intelligence techniques shap stand for additive shapley explanations and is generally a local explanation technique. Thus, it aims to explain individual predictions of black-box models. However, it is also possible to obtain valid global explanations by aggregating individual predictions.

Mathematically, the shapley values are computed using the following formula:

$$\phi_i(f,x) = \sum_{z' \subseteq x'} \frac{|z'|!\,(M - |z'| - 1)!}{M!} [f_x(z') - f_x(z' \setminus i)]$$

$\phi_i(f,x)$ represents the shapley value for feature i and requires the black-box model f and an input data point x (e.g., a single row in a tabular dataset) to be computed.

The first thing to do to get the values is to iterate over all possible subsets $z' \subseteq x'$, i.e., a combination of features, in order to account for interactions between individual feature values. The reason the sampling space is denoted x' is because for more complex inputs such as images, not every pixel is used as a feature, but they are summarized in some way (x $\longrightarrow x'$).

Then, it is necessary to obtain the output of the black-box model for each subset with and without the feature of interest, which is the $i-th$ feature. The differences between the two outputs indicate how the $i-th$ feature contributed to the prediction in the subset. These calculations are repeated for each possible permutation of subsets. Each of these is also weighted by how many players there are in the coalition or, in other words,

how many of the total number of features there are in the subset. Thus, M in the formula stands for the total number of features.

The intuition is that the contribution of feature i addition should be weighted more if many features are included in that subset so that the specific feature gives a significant change in the prediction even if many other features are already included.

On the other hand, more weight should also be given to small coalitions because there, the features are isolated, and their effect on the predictions is directly observable.

However, there is another problem to analyze. It concerns how to exclude features from a machine learning model. Typically, inputs have a fixed size, and removing only part of them is impossible because the shape would change. Therefore, the idea behind shap is to impute random values from the training dataset to exclude features. If this operation is done for all subsets when calculating the inputs, the relevance of these features is essentially sampled, which means shuffling the features by making them random. Thus, since random features have no predictive power, this procedure is like exclusion.

Computing all permutations necessary to obtain shapley values is computationally expensive. More precisely, obtaining all possible subsets is an exponential term.

$$total\ number\ of\ subsets\ of\ a\ set\ = 2^n$$

For example, when having 10 features, there are 1020 possible combinations for the subsets; therefore, many calculations should be performed to get the average contribution of a single feature.

The basic idea is to approximate the shapley values instead of computing all the combinations. Kernel shap, for instance, samples subsets of features and fits a linear regression model based on these samples. The variables in the linear regression model indicate whether a feature is present or absent in the subset. The output value is the prediction. After training, the coefficients of the linear regression model can be interpreted as an approximation of the shapley values. This is quite similar to LIME. However, here proximity is not considered. Furthermore, the samples are weighted according to how much information they contain.

Besides kernel shap, other approximation techniques exist for shapley values, for example, tree shap or deep shap, which are used for tree-based or deep neural networks, respectively. These techniques are not model agnostic, but they can use model internals to boost the performance when calculating shapley values.

3.3 Counterfactual explanations and adversarial attacks

So far, several interpretability techniques have been analyzed to understand black-box machine learning models better. These methods mainly produce feature importance and show how inputs affect predictions.

A counterfactual explanation is another type of explanation. A counterfactual refers to the smallest change in input characteristics that changes the prediction of another predefined output. Counterfactual explanations are also called contrastive explanations in literature.

Counterfactuals have long been present in psychology. David Lewis initially presented a theoretical definition in 1973. The idea for using them in machine learning was first presented in 2017 in a paper called "Counterfactual explanation without opening the black-box". The basic idea for calculating counterfactuals comes from a different AI field which is also called AI safety. The goal in this field is to make machine learning models more secure against manipulation.

Adversarial samples are well-designed input samples that use the weaknesses of machine learning algorithms to generate false predictions. The approach to generating adversarial samples is quite similar to generating counterfactuals since both look for minimal changes in the input to generate different output.

In both cases, the goal is to solve the following optimization problem:

$$arg\min_{x'} d(x, x') \ \ such\ that\ f(x') = c$$

That is, it finds x' which is the counterfactual or adversarial sample that changes the prediction of the black-box model to a target class c. In this optimization problem, the distance function $d(x, x')$ ensures that it stays as close as possible to the original input.

In both cases, when explaining and attacking black boxes, the goal is to find similar input data points that modify the prediction.

There are several approaches to computing counterfactuals for a prediction. The approaches can be divided into white-box and black-box.

White-box approaches are called model-specific; in these, the developer has access to the model. Suppose it is possible to access the internal elements of the model, such as the weights in the neural network. In that case, this information can be used to find the minimum in the optimization problem faster.

In contrast, the model cannot be accessed in black-box approaches, so the developer would have to rely on the relationship between input and output. Thus, a solution can be found by querying the model many times, which is the model agnostic way of computing counterfactuals.

Many different ideas exist for both categories. CertifAI is a method that uses a genetic algorithm to create counterfactuals and can be implemented when the model cannot be accessed. It uses selection, mutation, and crossover, typical operations in evolutionary algorithms. Another method is written in the paper "Counterfactuals without Opening the Blackbox" by (Watcher, Mittelstadt, & Russel, 2018). It is model-specific (white-box approach) and uses an atom optimizer and gradient descent to find the optimal counterfactual.

All approaches perform some perturbation on the input, meaning that they change feature values randomly or are guided in some way.

Finally, it is crucial to consider that when explaining a model using counterfactuals, there may be more than one explanation option. So, it is up to the developer to select the most appropriate explanations for the client.

4. TECHNOLOGIES

In this thesis, reinforcement learning will be used to train a model capable of moving forward and playing the well-known Super Mario Bros game. The model will be able to collect coins and make it to the end of the levels.

In order to do it, there are some important steps to follow.

Firstly, the Super Mario environment must be set up, so this means being able to make Mario interact with python code.

Then, the environment should be pre-processed for applied AI. Finally, the reinforcement learning technique is used to go ahead and teach it to play Super Mario.

In the following subsection, there is a summary of some useful libraries used to solve the problem.

4.1 Open AI Gym

Open AI Gym is a toolkit for reinforcement learning research written in Python code. The toolkit includes a growing collection of environments that expose known problems in reinforcement learning with a single common interface and a website in which any user can publish results obtained in a given environment in order to compare the performances of different algorithms with the community.

In addition, the users are also encouraged to share the source code that allowed them to obtain the uploaded results, with detailed instructions on how to reproduce the results they obtained.

Open AI Gym provides abstractions and interfaces helpful in creating new environments and can manage the rendering, avoiding the developer's worry about it. Given that the framework was created specifically for studying reinforcement learning algorithms, the interfaces proposed between the environment and the agent are in line with the elements required by the problem.

Open AI Gym presumes that the environment is episodic and that the agent interacts with it at each time step by performing one of the allowable actions. By interacting with the environment, the agent obtains information such as state, reward, and a flag indicating the eventual completion of the episode, which the agent uses to understand when it is appropriate to reset the environment and start a new episode.

Open AI Gym makes available the env class, which contains the environment and possible internal dynamics. The class has several methods and attributes to implement to build an environment. The most relevant methods are called reset, step, and render.

The reset method is designed to reset the environment, initializing it to its initial state.

The step method is responsible for advancing the environment by a one-time step. It requests the action to be performed in input and returns the new observation to the agent. The method's handling of movement dynamics, calculation of state and reward, and episode completion checks must be defined.

The third and last method is the render method which must define how the elements at each step must be represented. The method provides different types of rendering, such as human, rgb_array, or ANSI. With human type, the rendering occurs on the screen or terminal, and the method returns nothing; with rgb_array type, invoking the method returns an n-dimensional array representing the RGB pixels of the screen; in choosing the third type, the render method returns a string containing a textual representation. To perform the rendering, Open AI Gym provides the viewer class through which it is possible to draw the elements of the environment in the form of a set of polygons and circles.

Concerning the attributes of the environment, env provides the definition of action space, observation space, and reward range. The action space attribute represents the set of possible actions that the agent within the environment can perform. By means of the observation space attribute, it is defined the number of parameters that make up the state and the range of values that can be assumed for each of them.

The reward range attribute contains the minimum and maximum reward obtainable in the environment, by default set to $]-\infty, \infty[$.

4.2 PyTorch

PyTorch is an open-source framework developed initially by the AI group of Facebook; it is written in Python, C++, and CUDA, and it is used in machine learning and Deep Learning. PyTorch is based on torch, a library that provides a wide choice of Deep Learning algorithms.

One of the most interesting aspects of PyTorch is the use of tensors. Tensors are a particular type of vectors and n-dimensional numeric arrays. This aspect is one of the strengths of this framework: PyTorch integrates well with Python (or, to use an ad hoc term, is very Pythonic), an aspect that is reflected, to give an example, in the similarity between tensors and NumPy objects and the ease with which they can interact.

5. IMPLEMENTATION

5.1 Environment

The first step in setting up the Super Mario Bros environment is initializing it. Thanks to the make method, it is easily possible to load it from the Gym library.

```
env = gym_super_mario_bros.make("SuperMarioBros-1-1-v0")
```

In this case, the environment is the first level of the first world of Super Mario Bros Game. The game features are some obstacles, such as pipes, blocks, and enemies, but also some objects that give Mario more points, such as mushrooms and coins.

Figure 12 Super Mario Bros environment

In *Figure 12* can be visualized the environment. An important step is defining the action space, which consists of the possible movements Mario can take. Generally, the action space of the game comprehends the 12 following moves:

- NOOP: do nothing
- Right: walk right
- Right + A: jump right

- Right + B: run right
- Right + A + B: run and jump right
- A: jump
- Left: walk left
- Left + A: jump left
- Left + B: run left
- Left + A + B: run and jump left
- Down: duck, enter a pipe, or climb downwards on a beanstalk
- Up: climb upwards on a beanstalk

However, if the action space includes all these movements, then the algorithm needs more episodes to teach Mario to complete the model. To simplify, only a subset of these movements has been selected. The gym library already has some pre-set action spaces defined that are:

- ONLY_RIGHT: includes the possibility of doing nothing and all movements to the right.
- SIMPLE_MOVEMENT: includes the possibility of doing nothing, jumping, going to the left, and all movements going to the right.
- COMPLEX_MOVEMENT: includes all twelve movements listed above.

In order to define the action space, the following line should be included in the code. The variable possible_actions consists of a list containing all the allowable actions Mario to take.

```
env = JoypadSpace(env, possible_actions)
```

Each input state is an array of sizes $[3, 240, 256]$. However, to train the model, it is not strictly necessary to keep them that way. For example, Mario learns how to do the level even if the environment is grayscale, or again it can learn even if the width and height are reduced to 84×84.

Then, the following function has been written in order to pre-process the environments:

- GrayScaleObservation: it is a function that transforms an RGB image into a grayscale. In doing this modification, the size of the image reduces. It goes from $[3, 240, 256]$ to $[1, 240, 256]$.
- ResizeObservation: it down samples each observation into an image of dimensions 84×84. Then the new array size is $[1, 84, 84]$.
- SkipFrame: This function is introduced because consecutive frames are not that different. Thus, it is possible to skip n intermediate frame without losing so much information.
- FrameStack: This function allows consecutive frames of the environment to be stacked together, causing multiple frames to represent a single observation. Thus, by feeding the learning model with the stacked frames, it is possible to identify whether Mario was landing or jumping based on the direction of movement in the previous frames.

```
env = SkipFrame(env, skip=4)
env = GrayScaleObservation(env)
env = ResizeObservation(env, shape=84)
env = FrameStack(env, num_stack=4) # 4 frames stacked together
```

Each observation is converted into a torch tensor within one of these functions. In the implementation, the performance of the model was analyzed both with FrameStack equal to 4 and equal to 1. The reasons connected with this choice will be explained more in the section dealing with the implementation of explainable AI.

If FrameStack is equal to 4, then the final size of the state is $[1, 4, 84, 84]$. While if its value is equal to 1, then the final size is $[1, 1, 84, 84]$.

5.2 Agent

In implementing the agent, it is necessary to define the Mario class with all the methods to be used for training the model. The methods needed are:

- Act: given a state, it chooses an action and updates the value of the step. This function takes as input a single observation of the current state of dimensions $[1, 4, 84, 84]$ (or equal to $[1, 1, 84, 84]$ based on FrameStack), and it gives as the

outcome an integer number representing the action that Mario will perform. The act function is used to select an action Mario will perform, and in order to do this, two methods are helpful for the learning process. Firstly, one can explore the environment by taking random actions. Secondly, one can exploit what the agent has learned by exploring the environment. Here, it is necessary to consider the trade-off between exploitation and exploration. When deciding whether to explore or exploit, the function samples a number from a uniform distribution. Random action is executed if the sampled number is less than the exploration rate. On the other hand, if the sampled number is greater than the exploration rate, the exploitation step is executed. The exploration rate is updated at each step and decreases as the number of actions taken increases.

- Cache and recall: The cache function stores experience in memory while the recall function retrieves a batch of experience from the memory. These two functions have to be implemented in order to be able to use the Double DQN algorithm.
- TD estimate and TD target: these two values are involved in the learning procedure. TD estimate represents the prediction of the optimal Q^* for a given state s, while TD target is the sum of the current reward and Q^* evaluated in the next state s'. Then, mathematically they can be written as:

$$TD_e = Q^*_{online}(s,a) \qquad a' = \underset{a}{arg\,max}\, Q^*_{online}(s',a)$$

$$\text{then} \quad TD_t = r + \gamma Q^*_{target}(s',a')$$

  Since a' is not known, it is chosen in such a way it maximizes Q^*_{online} evaluated in the successive state s'.

- Update Q online: While Mario samples experiences from his memory, this function computes TD_e and TD_t, back-propagating the loss function down Q_{online} to update the parameters θ_{online}.

$$\theta_{online} \longleftarrow \theta_{online} + \alpha\nabla(TD_e - TD_t)$$

  The loss function implemented for this procedure is taken from the PyTorch library, specifically the SmoothL1Loss.

  For a batch of size N, the loss can be described as $\ell(x,y) = mean(L)$ where:

$$L = \{l_1,\dots,l_n\}^T \text{ with } l_n = \begin{cases} 0.5\,(x_n - y_n)^2/beta, & if\ |x_n - y_n| < beta \\ |x_n - y_n| - 0.5\,beta, & otherwise \end{cases}$$

- Sync target: Loads model's parameter dictionary using a deserialized state_dict.
- Save checkpoints: it is used to save the model.
- Learn: the learn function brings all the previous methods together. Specifically, it samples from memory, obtains the TD target and TD estimate, and then back-propagates the loss through Q_{online}.

Moreover, the MetricLogger has been created. This class is used to create a log file that saves all metrics at each step.

Finally, in the Mario class within the `__init__` values it is set:

```
self.net = MarioNet(self.state_dim, self.action_dim).float()
```

MarioNet is a class that inherits the properties of nn.Module. In this class is implemented the deep neural network used to teach the agent to complete the layer.

The network takes as input a state which passes through three convolutional layers and two fully connected layers, returning as output a tensor with the scores of each action. This tensor will then be used to select an action; the one with the highest score will be selected.

The first convolutional layer has a kernel of size 8×8 with a stride of 4. The second convolutional layer has a kernel of size 4×4 with a stride of 2. In contrast, the third has a kernel of size 3×3 with a stride of 1. The kernel is a matrix that defines the convolutions of the net. Such an array is generally of a smaller size than the image and is applied by translating the kernel onto it. Instead, the stride represents the kernel translation speed on the image, measured in horizontal and vertical pixels.

After the last convolutional layer, there are two fully connected layers. A layer is said to be fully connected because each of its inputs is connected to its outputs by means of matrix-vector multiplication.

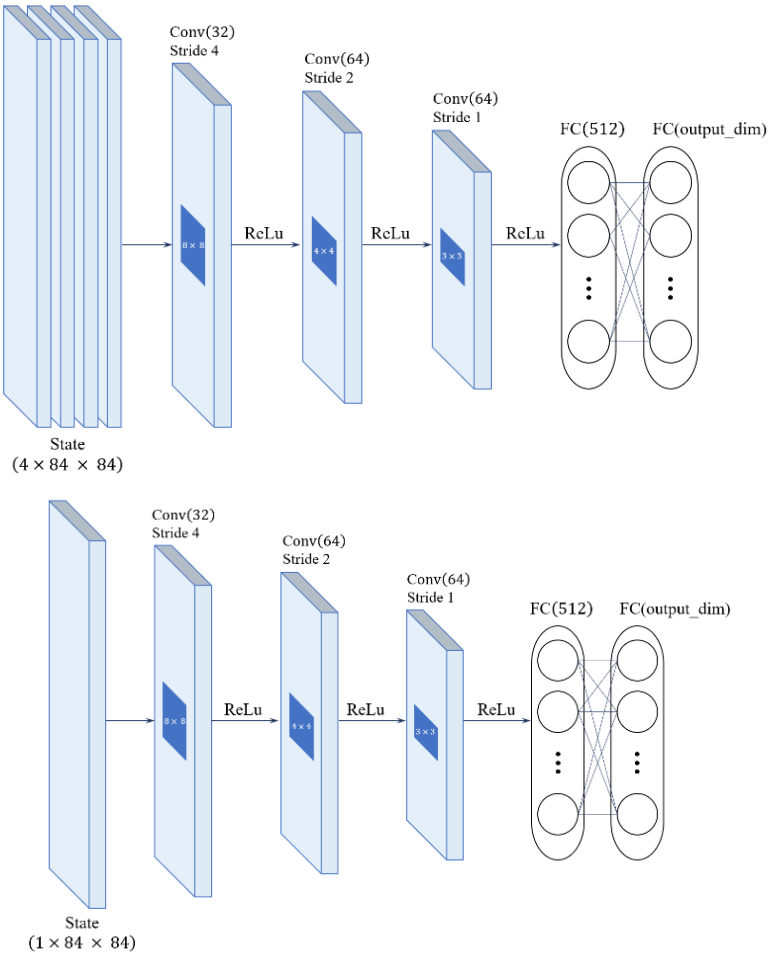

Figure 13 Implemented Neural Network scheme

The input state of the Neural Network also changes if the parameter FrameStack changes (*Figure 13*).

5.3 Train the model

After pre-processing the environment and defining the agent in all its characteristics, let us proceed with the actual training of the model. In doing this, it is first necessary to define the number of episodes to be executed. The choice of this number is not trivial; in fact, using the DDQN algorithm to train the Neural Network could lead to an overfitting problem. This problem occurs mainly because of the use of experience replay; a well-known problem related to learning algorithms is having a high correlation between successive experiences.

After defining the number of episodes, a for loop is implemented to iterate over them. For each episode, the environment is reset. This is due to the fact that for each episode, the game is restarted.

Within the for loop, there is a while loop that iterates until the end of the game, which occurs upon the realization of two possible events: the death of the agent or the end of the time available to complete the game.

Based on the current state, the agent selects the first action for each step of the while loop. After that, it passes the selected action to the env.step function. At this point, the env.step function outputs the next state, the reward, a flag indicating whether the episode is finished or not, and some information related to the performed action. In saving the agent's experience, the cache function, defined in the agent class, stores all the information obtained from the env.step function (agent experience) to memory (a deque type list).

Next, the learn function is called, which implements the DDQN algorithm used to select the actions to be performed by Mario.

Finally, the episode ends when the "done" flag in the env.step function becomes equal to True. At this point, the while-loop stops, and the for-loop switches to the next episode.

At the end of the procedure, the results consist of the trained model.

```
episodes = 100

for e in range(episodes):
 state = env.reset()

 while True:

         # Make the agent select an action
         action = mario.act(state)

         # Agent performs the action
         next_state, reward, done, info = env.step(action)

         # Store the experience - Remember
         mario.cache(state, next_state, action, reward, done)

         # Learn step
         q, loss = mario.learn()
```

```
        # Logging
        logger.log_step(reward, loss, q)

        # Update state
        state = next_state

        if done or info["flag_get"]:
break

logger.log_episode()
logger.record(episode=e, epsilon=mario.exploration_rate, step=mario.curr_step)
```

5.4 Test the model

In reinforcement learning, the testing phase is usually used in order to evaluate the quality of the policy learned by the agent and to compare different algorithms. However, unlike supervised learning techniques, this type of evaluation can also be done during model training. In this section, therefore, the term test does not properly refer to evaluation but rather to the use of the model.

In fact, the main goal is to use the model in order to save frames (as shown in *Figure 14*) that will later be used to try to explain the choices made by the agent.

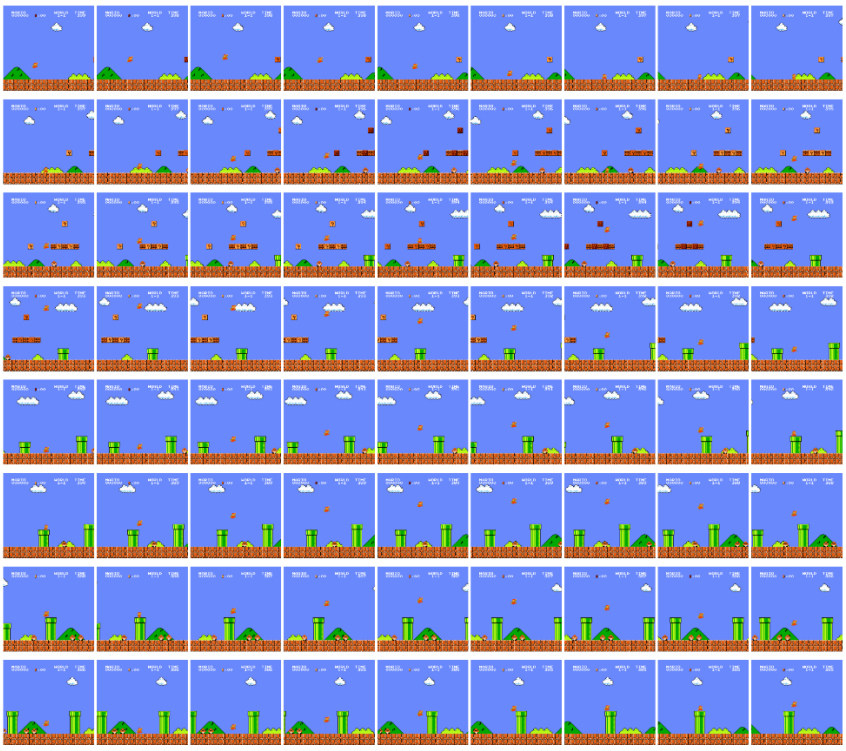

Figure 14 Plot of test frames

5.5 Explainable AI

Explainable AI algorithms try to find explanations for the results of black-box models. In Deep Reinforcement Learning, these algorithms can be particularly useful.

One of the most popular algorithms in this field is LIME. In implementing it, it is necessary to import the LIME package, which provides many helpful functions for the purpose of analysis.

When getting explanations of the game frames, the procedure is quite simple. First, initialization of the explainer is necessary. Then, this class will be used to get the explanations, explicitly using the explain_instance method.

```
explainer = lime_image.LimeImageExplainer()

explanation = explainer.explain_instance(concat_im, net_lime,top_labels=3,
hide_color=0, num_samples=1000)
```

The function takes the following components as input:

- An RGB image
- A classifier function (classifier_fn): takes a numpy array and outputs predictions. In this function, it is also included some pre-processing operations on the image.
- An iterable object with the labels to be explained.
- top_labels: if not None, it produces explanations for the K labels with the highest prediction probabilities.
- The maximum number of features present in the explanation (num_features)
- The size of the neighborhood to learn the linear model (num_samples)
- batch size
- The distance metric to use for weights (distance_metric).
- The regressor to use in the explanation, the default value is Ridge regression in LimeBase (model_regressor).
- A segmentation function (segmentation_fn).

- A random seed (random_seed).

After that, it generates neighborhood data by randomly perturbing the pixels (features) of the image (instance). Then, locally weighted linear models are applied to these neighborhood data to explain each class in an interpretable way.

The algorithm is applied to two different scenarios: the first scenario is when the state given as input to the network is of size $[1, 4, 84, 84]$, while the second scenario is when the neural network takes as input a state of size $[1, 1, 84, 84]$.

5.5.1 First scenario (input state is composed of 4 stacked images)

In this scenario, the implementation is not straightforward because LIME takes as input only an RGB image. In order to obtain the desired results, two different procedures have been used.

The first procedure consists in creating a function to merge four images into a single image (since LIME only takes one image as input) and then creating a function to split them again before feeding them to the network.

```
def concatenate_img(im1, im2, im3, im4, plot_flag=False):
 vertical = np.concatenate((im4, im2), axis=0)
 vertical1 = np.concatenate((im3, im1), axis=0)
 horiz = np.concatenate((vertical, vertical1), axis=1)
 if plot_flag == True:
 Image.fromarray(horiz).show()
 return horiz

def separate_img(image):
 hsep = np.hsplit(image, [256])
 hsep0 = hsep[0]
 hsep1 = hsep[1]
 vsep0 = np.vsplit(hsep0, [240])
 vsep1 = np.vsplit(hsep1, [240])
 images = [vsep0[0], vsep1[0], vsep0[1], vsep1[1]]
 return images
```

The `concatenate_img` function receives as input four images to be merged and a flag indicating whether or not to plot the four images after merging them. The `concatenate`

function from the numpy package has been used to put together the four arrays of dimensions $[3, 240, 256]$. This function also requires specifying on which axis it is wanted to concatenate the arrays. The arrays are concatenated on the vertical side by setting the axis parameter to zero. In contrast, by setting the axis parameter to one, the matrices are concatenated on the horizontal side.

The `separate_img` function uses the `hsplit` and `vsplit` functions from the numpy package for splitting up the image created with the first function in the chunk of code.

After defining functions to merge and separate images, it is necessary to implement a function that, from a list of images, applies all necessary transformations to obtain the state that can be taken as input by the neural network. This task is implemented in the `image_to_state` function below.

```
def image_to_state(list_img):
 four_frames = []
 for i in range(4):
 img = list_img[i]
 img = np.resize(img, (1, 256, 512, 3))
 img = img.squeeze(0)
 img = np.transpose(np.asarray(img), (2, 0, 1))
 img = torch.tensor(img, dtype=torch.float64)
 bw_transform = T.Grayscale()
 img = bw_transform(img)
 img = transforms(img)
 four_frames.append(img)
 state_lime = torch.stack((four_frames[0], four_frames[1], four_frames[2],
four_frames[3]))
 state_lime = state_lime.squeeze(1)
 state_lime = state_lime.unsqueeze(0)
 state_lime = state_lime.expand(10, -1, -1, -1)
 return state_lime

def net_lime(image):
 list_images = separate_img(image)
 state_lime = image_to_state(list_images)
 return net(state_lime, model='online').detach().numpy()
```

To recapitulate, the first procedure adopted in this first scenario follows these steps:

1. Concatenate four images;

2. LIME perturbs the image to see which pixels contribute most to the choices made by the agent;
3. Separate the perturbed image to obtain the four frames;
4. Transforms the four frames to obtain the state that can be taken as input by the neural network;
5. Since the function wants the image and the model as input, it is necessary to create a function that applies the transformations in 3 and 4.

The function that does this in step 5 is called `net_lime`.

The second procedure uses a different logic since proceeding as above could give some problems. In particular, when images are merged into one, the image that LIME analyzes looks like the one in *Figure 15*. However, in reality, the state that the network takes as input does not have that form. Thus, the idea applied in the paragraph above may not be the best one.

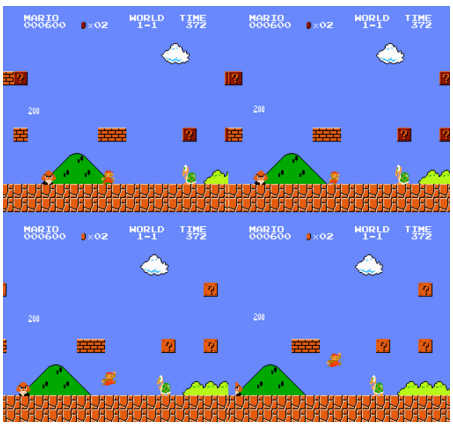

Figure 15 Merged frames

So, the idea is to feed LIME the original image so that the perturbed image is the one of interest. An alternative approach to implementing this idea might follow these steps:

1. Prepare a function that, given four images, applies all the necessary transformations and creates the right-sized state.
2. Create a net_lime function that takes the three frames preceding the image that will be given input to LIME and applies the function in step 1. Then, having obtained the state, feed it to the neural network.
3. Launch LIME with as input the image and the net_lime function.

5.5.1 Second scenario (input state is composed of one stacked image)

In the second scenario, the network was modified so that the states consisted of a single frame instead of being an envelope of four images. In this case, the implementation is simpler and fairly straightforward. In fact, since the state to be given as input to the neural network consists of only one frame, it is no longer necessary to use the function that concatenates the image. Now the only thing that should be done is the function that applies transformations, such as Resize, and GrayScale, to the image that is given as input to LIME.

```
def image_to_state(image):
 img = np.resize(image, (1, 240, 256, 3))
 img = img.squeeze(0)
 img = np.transpose(np.asarray(img), (2, 0, 1))
 img = torch.tensor(img, dtype=torch.float64)
 bw_transform = T.Grayscale()
 img = bw_transform(img)
 img = transforms(img)
 state = img.unsqueeze(0)
 state = state.expand(10, -1, -1, -1)
 return state

def net_lime(image):
 state_lime = image_to_state(image)
 return net(state_lime).detach().numpy()
```

Finally, the upside of the first scenario is that using four frames stacked together as a state allows the network to have a wider view. However, this first approach then causes difficulties in the application of files.

The positive side of the second approach, on the other hand, is that considering a state as a frame makes it easier to implement LIME.

6. RESULTS AND DISCUSSION

6.1 Trained model

Agent training is conducted in several ways. *Table 1* shows all the ways in which the training was performed, along with a rating of how it seems to perform in the test. The performance column of *Table 1* contains indications of how the agent is performing at the level, and the words used have the following meanings:

- Stuck: it indicates when, after advancing a little, the agent gets stuck in front of the obstacle, and the level ends due to the expiration of time.
- Dead: it is related to the death event of the agent (collision with an enemy, for example).
- Complete: it indicates that the agent has successfully reached the end of the level.

episodes	*FrameStack*	*Performance*	*Action space*
2k	4	Stuck	All right movements and all left movements
20k	4	Dead	All right movements and all left movements
40k	4	Complete	All right movements and all left movements
40k	1	Dead	Simple_Movements
40k	1	Dead	Right_only without NOOP
150k	1	Stuck	Right_only without NOOP

Table 1 Summary of training executions

In particular, the table shows that the agent seems to require at least 40k episodes to be trained.

However, it is necessary to check whether there is overfitting in the model. When talking about overfitting, one is referring to the situation where, instead of learning the general pattern recognition rules, the model seems to learn example patterns by heart, recognizing those already seen well but making many errors in patterns not yet viewed.

In a reinforcement learning model, and specifically in the case of the videogame Super Mario Bros the phenomenon of overfitting can be noticed by having the agent play another level. Let us examine the model trained as in the third row of *Table 1*. If the agent, trained on the first level, plays another level, it fails very quickly. This is a sign of overfitting.

In order to overcome this problem, one possible solution would be to change levels for each episode, thus making the agent learn all possible obstacles. Some graphs related to this approach will be shown at the end of this section and explained in a bit more detail. The adoption of this strategy is not directly part of the purpose of this thesis, which instead aims to show explanations of the decisions made by the network used in the Deep Reinforcement Learning model.

Another interesting aspect to note in training the model is the choice of the action space. The choice of actions has an impact on training. Introducing certain actions can allow the agent to increase rewards, but doing so could cause a slowdown in the learning process. Removing actions relevant to the game can cause the agent to lose the game. Therefore, it is essential to choose actions in such a way as to streamline the agent's training.

Let us analyze in detail each scenario in which the agent is trained.

The scenario in the first row of the table can be omitted in the explanation as 2000 episodes are few for training. It is considered better to move on to the explanation of the training that lasts 20k episodes.

This first attempt shows that after 20k episodes, the agent fails due to crashing into an enemy after only 74 actions. Let us then proceed by increasing the number of episodes, leaving the possible actions Mario can do unchanged.

With 40k episodes of training, the agent finishes the game successfully. *Figure 16* shows graphically which actions the agent chooses most frequently. It can be seen that the most commonly chosen action is to go right by running. This is reasonable as the other actions are only chosen when obstacles or points are reached.

On the other hand, it is interesting to understand the situations where Mario goes left as the action involves going backward away from the end of the game. However, the agent chooses such actions only 5% of the time.

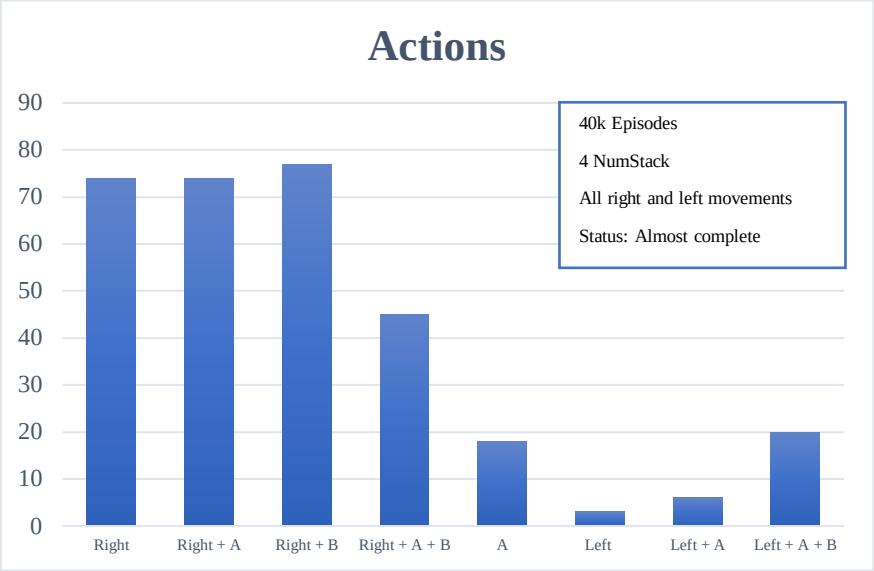

Figure 16 Bar chart of the actions in the scenario reported in the third row of Table 1

Another test was carried out based on an environment where the state is not represented by four frames stacked together but rather by a single frame.

Another difference is that the agent is programmed to do actions included in the list called SIMPLE_MOVEMENT specified above. After training the network for 40k episodes, it is tested, and it is verified whether the agent has acquired the ability to complete the level. In this case, the agent is not able to complete the level but gets stuck after a limited number of actions.

In *Figure 17*, the number of actions chosen by the agent before failing can be analyzed. In this case, the game ends because the agent runs into an enemy. The agent fails shortly after the start of the episode, so the total number of actions is considerably lower than that shown in *Figure 16*. An important thing to note is that the agent often chooses to do nothing (NOOP). In particular, the agent decides not to do actions 11 times.

Therefore, it is possible that giving the agent the option to perform no action decreases his ability to complete the pattern and increases the number of episodes required for training, making the process more time-consuming.

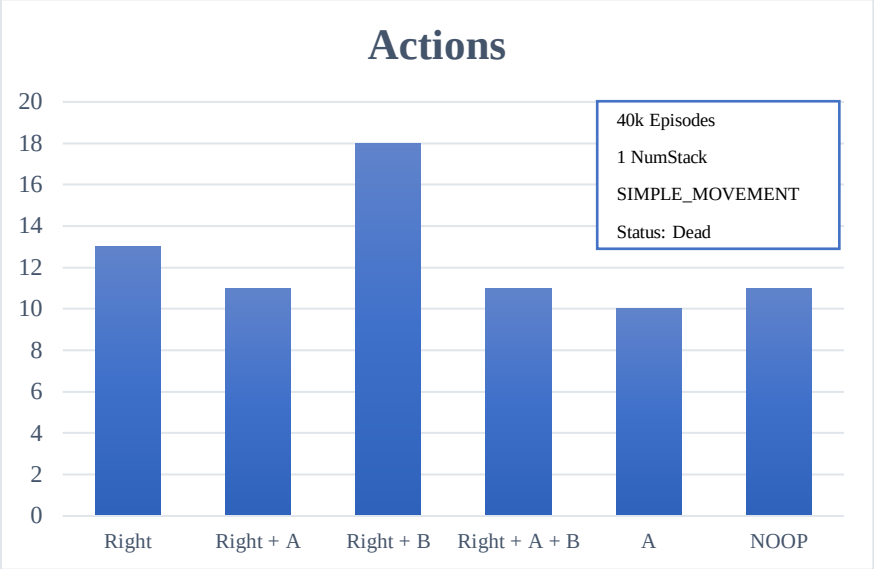

Figure 17 Bar chart of the actions in the scenario reported in the fourth row of Table 1

Finally, one last analysis can be performed. Since some measurements for each episode are saved in a log file during training, it is possible to track them by comparing the algorithm's behavior in different executions. Each metric represents a moving average of the last 100 observations, and they are as follows:

- MeanReward: the average of the last 100 observed rewards.
- MeanLength: the average length of the last 100 episodes.
- MeanLoss: derived by the loss between TD target and TD estimate in this application, the loss measure implemented is the SmoothL1Loss.
- MeanQValue

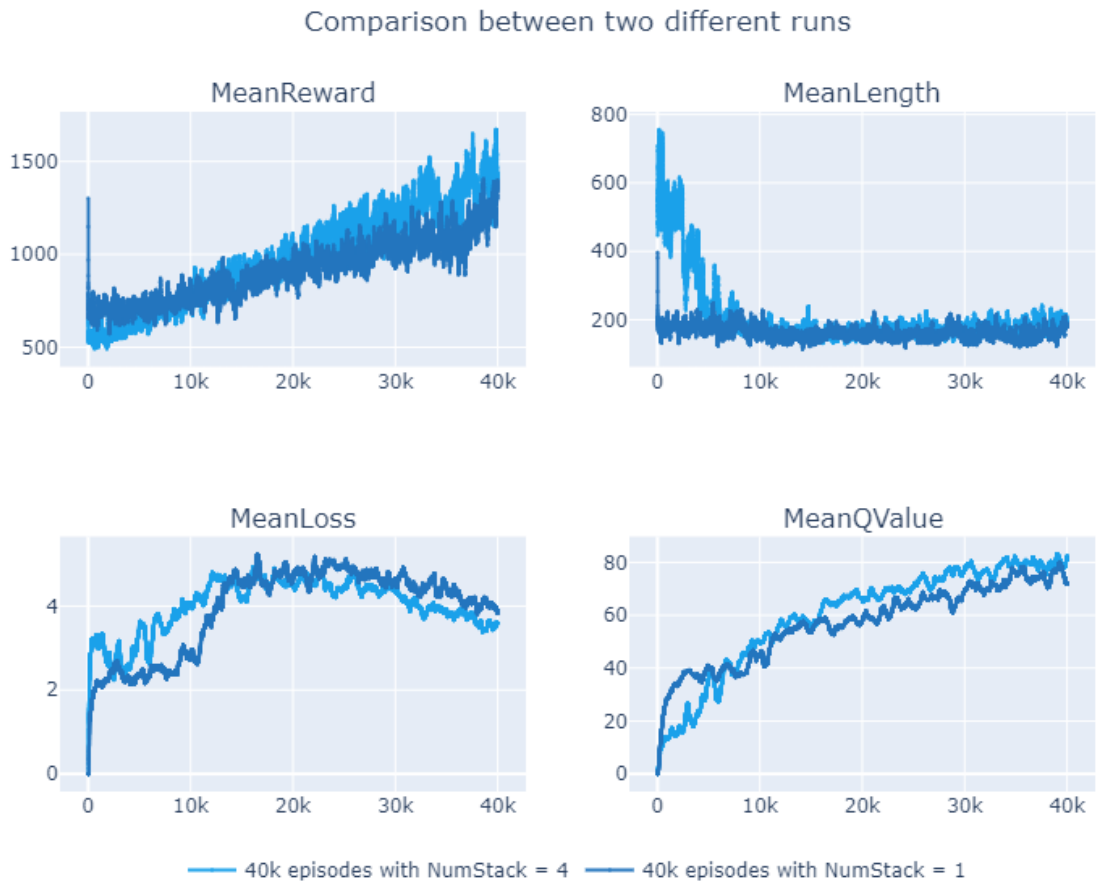

Figure 18 Comparison between two runs through logging information

In general, looking at *Figure 18*, the execution seems better when NumStack is set to four. Specifically, mean rewards tend to grow faster and, in the end, achieve a higher reward. Regarding the mean length, which represents the episode's length, it can be seen that the scenario with $NumStack = 4$ at the beginning has a longer duration, dropping dramatically after about 8k episodes. This means that probably at the beginning, the game ends due to running out of time, and then as the agent is trained, he reaches the finish line more and more often before time runs out.

Instead, looking at the graph of the mean loss, it can be seen that in the middle part, it seems to show an increase which then falls back towards the end. Also, in this plot in the last episode, the lower loss is related to the scenario where $NumStack = 4$.

Let us make two interesting final tests.

First, it is interesting to note that the models analyzed so far are never trained for more than 40k episodes. This happens because it was observed that when running the model for 150k episodes, the result is as shown in *Figure 19*.

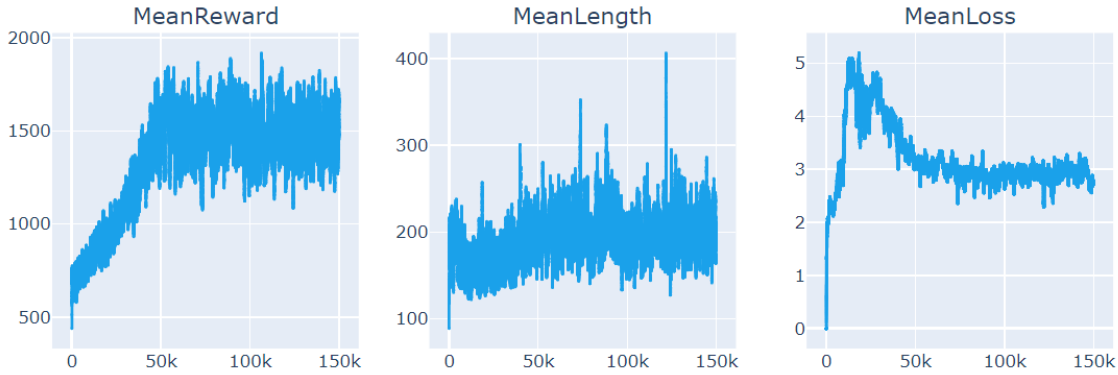

Figure 19 Model trained for 150k episodes

The MeanReward plot and the MeanLoss plot clearly show that after 50k episodes, the algorithm does not progress significantly. Therefore, it is not deemed necessary to use a model trained for more than 50k episodes.

Second, it is interesting to train the model by changing levels for each episode. This test is essential to avoid overfitting as much as possible. In reinforcement learning models, the phenomenon of overfitting occurs when the agent trained in a particular environment learns to behave perfectly in that environment but, by slightly changing it, starts to misbehave.

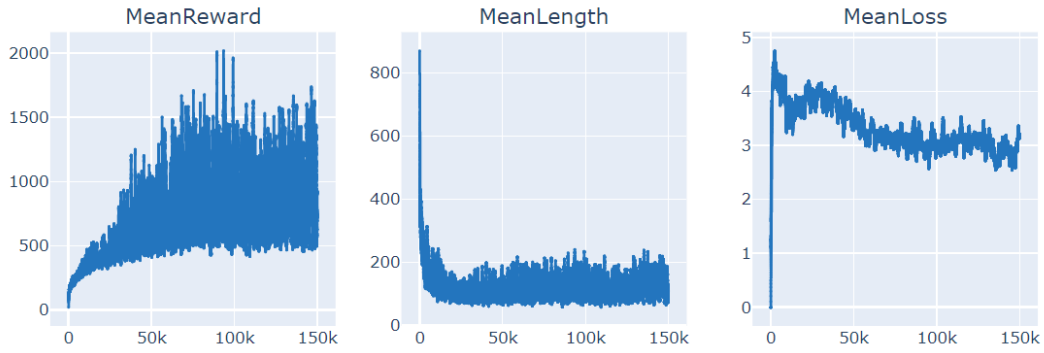

Figure 20 Model trained on random levels for 150k episodes

Figure 20 shows the results obtained when training the model on random levels.

When trying this different approach to train the agent, it can be seen that there are some problems. The MeanReward graph clearly shows that the variance increases dramatically after 50k episodes. The other two graphs also show some variability. In addition, the agent does not perform very well across levels.

The increase in variance could be related to the fact that when the level is changed, the agent's skill also changes a lot.

In addition, it should be underlined that this approach was tried in the case where the environment consists of states characterized by only one frame ($NumStack = 1$). This aspect could affect the agent's ability to learn.

Therefore, training could be improved by trying to set $NumStack = 4$.

6.2 Explainable AI

It is now essential to focus on the results obtained from the application of the explainable AI algorithm, which in the specific case of the analysis is LIME. LIME stands for Local Interpretable Model-agnostic Explanations, and as the name suggests, it attempts to give a local interpretation of the model. This means that it breaks down a complex algorithm into many smaller local points, where the prediction can be explained in a simple (and usually linear) way. In particular, the principle is to perturb the model input data and check how the model output changes, repeating this process

until a graphical representation of how the model works with respect to specific features (e.g., phonemes in an audio signal or contours in an image) is constructed. In general, existing agnostic techniques do not allow for understanding the interactions between different features of the model. On the other hand, LIME tries to improve other agnostic algorithms by trying to overcome this limitation through a multi-featured model perturbation focused on a specific model prediction.

6.2.1 *Results first scenario (input state is composed of four stacked images)*

The implementation of explainable AI algorithms can be complicated when the state contains four stacked frames.

As explained above (Chapter 5.5), two different methodologies were implemented to obtain the results with files.

The first procedure is based on juxtaposing four images into a single image (see *Figure 15*), perturbing pixels in the merged image, and re-separating the four images by creating the right size state. Then the neural network takes this state as input.

On the other hand, the second procedure uses the original image as input, perturbs the pixels in that image, draws the three previous images to create the state, and feeds it to the neural network.

The outputs of the two procedures are logically different. In the first procedure, it is possible to see the most important pixels scattered across all four frames, while the second procedure returns only the image of interest. In order to obtain explanations, the frames of the test phase in which the trained neural network is used must first be saved.

The first explanations obtained concern the run of the trained neural network for 20k episodes. As explained above, when training the agent for 20k episodes, it fails by crashing into an enemy.

Figure 21 Two explanations (First procedure) with a model trained for 20k episodes

Figure 21 shows the explanations obtained using the first procedure. In fact, in each image, it is possible to identify four frames side by side. In the image, the areas of pixels highlighted in blue/green are those that contribute to the decision made by the agent. In contrast, the areas highlighted in pink are the pixels that contribute most to the choice of any other action.

The images turn out to be quite complex to interpret. In fact, it is possible to see several areas highlighted as positive and as many as negative.

Observe first the image on the left. This image is at a point in the game where the agent makes a correct choice. Despite the confusion, it can be seen that LIME highlights several points as positive. Some of them are important in the choice of action, e.g., the cubes in the upper right or the enemy in the lower left.

Look now at the image on the right. The image represents the four frames used by the network to output the action that causes the agent to fail. In this case, the explanation seems to give conflicting information. In particular, let us look at the top right and bottom right frames. In the former, it appears that the area after the tube contributes positively to the decision, while in the latter, it appears that the same area is not important at all.

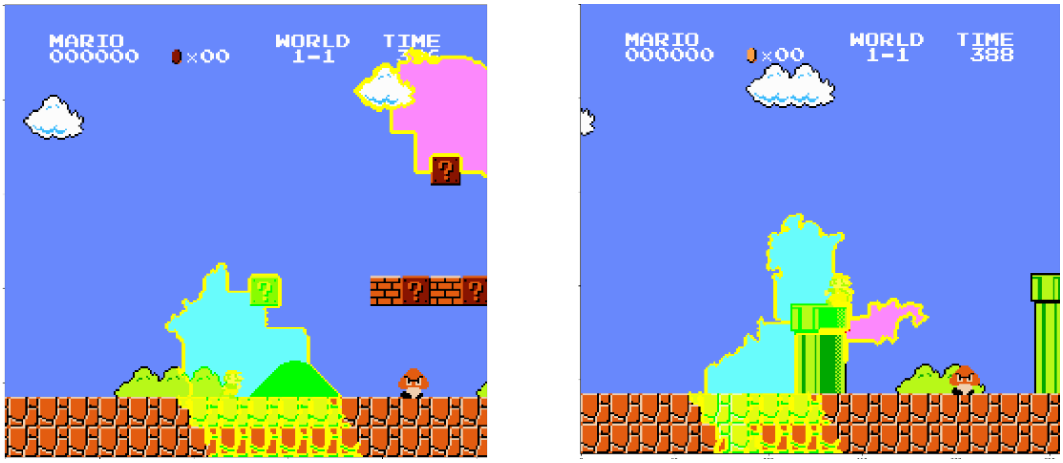

Figure 22 Two explanations (Second procedure) with a model trained for 20k episodes

When using the second procedure for the same model trained for 20k episodes, the explanations result as in *Figure 22*.

In this case, the explanations are less fragmented than before and clearer. Identification of important areas is now easier.

The first image in *Figure 22* clearly shows that the part of the pixel that contributes positively to the choice of action to be taken is the part around Mario. The area also includes a cube that gives additional points. Also, it can be seen that the least influential zone is a spot far from the agent.

The second image provides a different explanation. As mentioned above, this frame is the one just before the agent's failure (wrong action). Also, from the explanation, it can be seen that the part that affects the agent's action the most is actually not important at all. Moreover, the part that, according to LIME, affects the least should be the most influential.

Consider now the model trained for 40k episodes where the agent can go either left or right. Again, the results of the explanations for both procedures were provided.

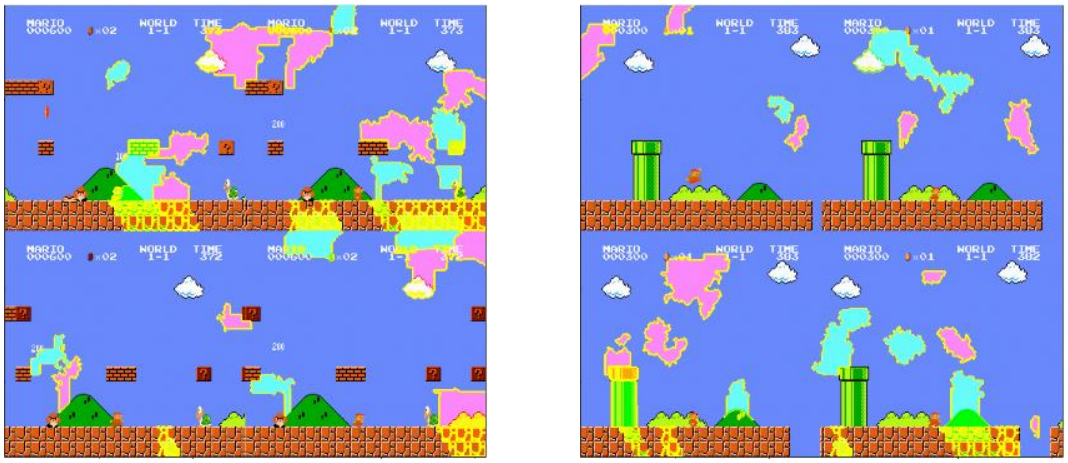

Figure 23 Two explanations (First procedure) with a model trained for 40k episodes

Let us look at the first image. Although difficult to interpret, the algorithm seems to make meaningful explanations. In fact, it highlights the agent and cubes in the upper left frame and the enemy's head, and a cube in the upper right frame.

In contrast, in the second image on the left, the network seems to make choices based on background details.

After seeing the behavior of this approach to obtain results in both training cases, it is possible to see that something is wrong. Previously, one might have thought that the poor LIME behavior was related to training the neural network for a few episodes. Instead, the poor LIME behavior is caused by the way the images are provided as input.

Examine now the behavior of the second procedure.

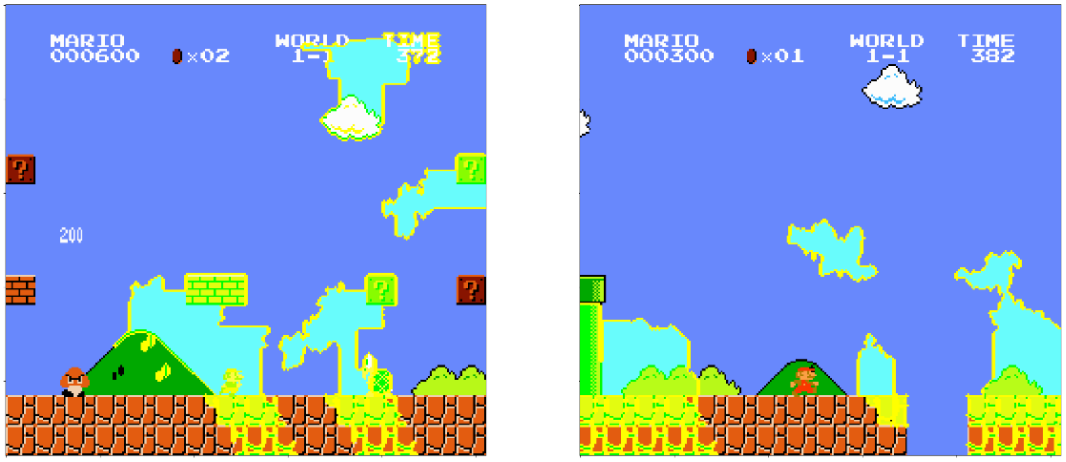

Figure 24 Two explanations (Second procedure) with a model trained for 40k episodes

In *Figure 24*, the explanations obtained by the second procedure show the following results and insights.

Let us look at the first image.

The frame illustrates a very clear explanation. First, it shows only features that make a positive contribution to the decision. The features highlighted by LIME in the first image are as follows:

- the blocks of bricks that can potentially increase the reward
- two blocks that not only increase the reward but also boost the strength of the agent
- an enemy

This explanation is one of the best of those displayed so far. In fact, it highlights all important factors to consider in making the decision of what action to take. Note that the only highlighted feature seemingly useless for Mario's advancement is the little cloud located at the top. So, thanks to LIME, if someone asked why Mario, at that moment in the game, chooses to jump to the right, the analyst would be able to answer that the reason is that in that area, there are objects that bring advantage and an enemy that could cause a failure. So the agent, by jumping, is able to avoid the danger and take the points.

On the other hand, the frame on the right shows that the most significant area consists of the part in front of Mario. Specifically, the algorithm highlights the edges that draw the hole into which the agent might fall. Also, in this frame, the algorithm returns an area of the background (behind Mario) as relevant.

In this case, the explanation for the next action could be that since the neural network does not perceive enemies in front of Mario and perceives the absence of the pattern of bricks where the hole is, the action that is chosen is to jump.

Finally, some observations need to be made. LIME seems to provide adequate explanations when the neural network that decides the actions to be performed by the agent is trained well.

Furthermore, after seeing in both training cases the performance of LIME with the two different implemented procedures, the following observations can be made.

The first procedure allows LIME to perturb the pixels not only of the image to be explained but also of the previous ones to be stacked to create the state. This operation confuses the algorithm. Indeed, at each iteration of the explainable AI algorithm, the network sees obscured pixels of both the image to be analyzed and the previous ones, causing the highlighting of influential areas to be confusing. In addition, having to create a new image with four frames, one cannot be sure that the arrangement is read and understood by the algorithm. In conclusion, if one wants to implement files on a model where the state format consists of four stacked frames, it is recommended to use the second procedure for implementation.

6.2.2 *Results second scenario (input state is composed of one single image)*

As discussed in the implementation chapter, the neural network was modified to take a state that consists of a single image as input. The reason this approach was also tried is to be able to get the explanations with LIME without having to involve modifications or stringent strategies before feeding the image to the explainable Artificial Intelligence algorithm. In fact, according to what has just been analyzed, the size of the original state (i.e., the one with $NumStack = 4$) gives some problems or needs more attention.

Therefore, consider the trained models with $NumStack = 1$. Specifically, two training attempts were made. Both attempts were run for 40k episodes, but in one, the agent could perform all the actions in the SIMPLE_MOVEMENT list, while in the second attempt, Mario could perform all the movements in the RIGHT_ONLY list except for the NOOP action, i.e., it was removed the opportunity to do no action. Consider that in both cases, the training of the algorithm ends up in agent failure. This means that in both cases, the agent does not learn how to finish the level.

The first explanation provided regards the first attempt.

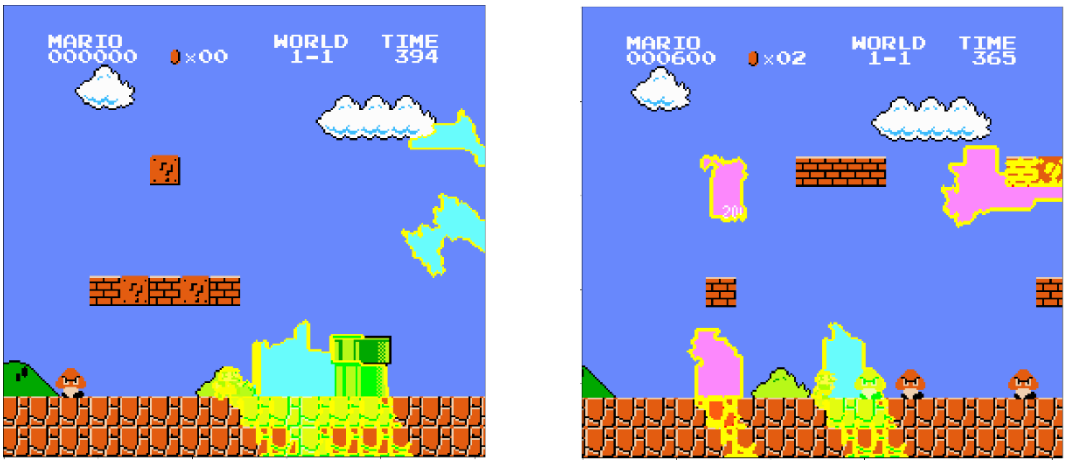

Figure 25 Two explanations with NumStack = 1 and SIMPLE_MOVEMENTS

The two explanations shown in *Figure 25* are quite interesting.

First, the frame on the left shows that the most important pixels that affect the agent's action are those closest to Mario. One aspect of the explanation to consider is that it also highlights part of the pipe. Thus, this result indicates that the jumping action the agent takes at that point in the game is due to its proximity to an obstacle to be crossed.

Instead, let us look at the image on the right. It can be seen all the pixels that contribute positively to the choice of action are around Mario. Thus, the only part that provides an explanation for the action taken is the highlighted area near the agent that includes an enemy. In general, the explanation here is that the most important factor considered by the algorithm is proximity to the enemy. Finally, there are also highlighted areas in pink

in this frame. These areas are the ones that are least considered by the algorithm. One thing that may not be appreciated is the highlighting as non-influential of the cubes that give points to the agent. However, the not-perfect training of the model and the fact that the cubes are far away from Mario could be the cause of the low influence.

The explanations are probably not entirely accurate in this case due to the fact that the model trained with $NumStack = 1$ has issues related to the fact that the neural network has difficulty perceiving the agent's movements.

Consider now the trained model in which Mario can perform all movements to the right except NOOP, i. e., except the possibility of no movements.

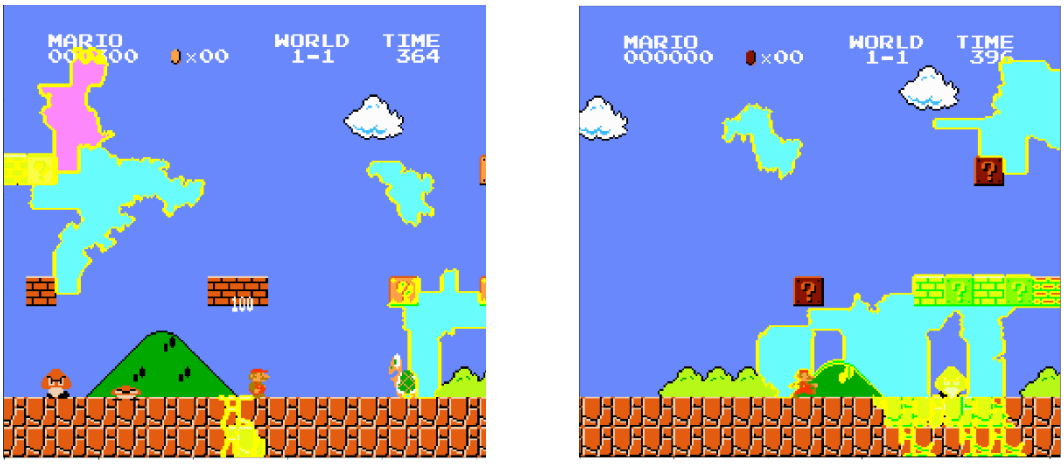

Figure 26 Two explanations with NumStack = 1 and RIGHT_ONLY

Figure 26 shows the explanations for the model with action space, which includes all movements to the right.

In the left image, not all areas marked as positive make sense for choosing actions. In fact, the algorithm marks part of the background behind the agent as relevant. However, the agent in this environment has only the option of going to the right. So, the algorithm should not consider the area to the left of the agent.

Even though this meaningless area is present in the image, an explanation of the other blue part underlined by LIME could be provided. In fact, it includes enemy and cube pixels that are relevant to the decision.

In the image to the right, the most important and significant part is located to the right of the agent so Mario can potentially reach it. In addition, the algorithm highlights one enemy and the cubes, features that certainly help to explain. However, even in this frame, scattered regions in the background are not influential.

Finally, when looking at the explanations that LIME returns, it is necessary to pay attention and evaluate whether what is displayed is possible or not.

6.3 Final comparison

The purpose of this section is to discuss and compare both the trained models and the methods used.

During the exposition of the results of the trained models, it was possible to acquire a general overview of all the attempts made. However, the question to be answered is which attempt among those analyzed best fits the problem.

The answer is not easy. In fact, if evaluation metrics are available in supervised learning when dealing with reinforcement learning, the testing phase does not directly provide an evaluation.

So to prefer one model over another, other methods must be used. In this case, one can use *Figure 18*, in which the measures represented are what is closest to evaluation metrics.

These quantities are collected during training but give information such as convergence and its speed.

Based on the data shown in *Figure 18*, the model in which the parameter corresponding to the number of selected stacked frames (NumStack) is equal to four is preferred.

The reasons related to this choice are not only about the results in the plots but also about the logic behind the parameter. The number of stacked images represents essential information for the neural network, i.e., it represents the fact that the agent is moving through a video. Taking four frames as a state helps the neural network to understand the action the agent is taking, such as the beginning of a jump or the end of a jump.

After expressing a preference on which model is deemed the best, consideration needs to be made. Preferring the model with the NumStack parameter equal to four brings complications in the context of explainable AI.

As a reminder, the procedures adopted for implementation are:

- First procedure: it is based on juxtaposing four images into a single image (see *Figure 15*), perturbing pixels in the merged image, and re-separating the four images by creating the right size state. Then the neural network takes this state as input.
- Second procedure: it uses the original image to be analyzed as input, perturbs the pixels in that image, draws the three previous images to create the state, and feeds it to the neural network.

The resulting explanations of the two procedures applied to the best model are the following.

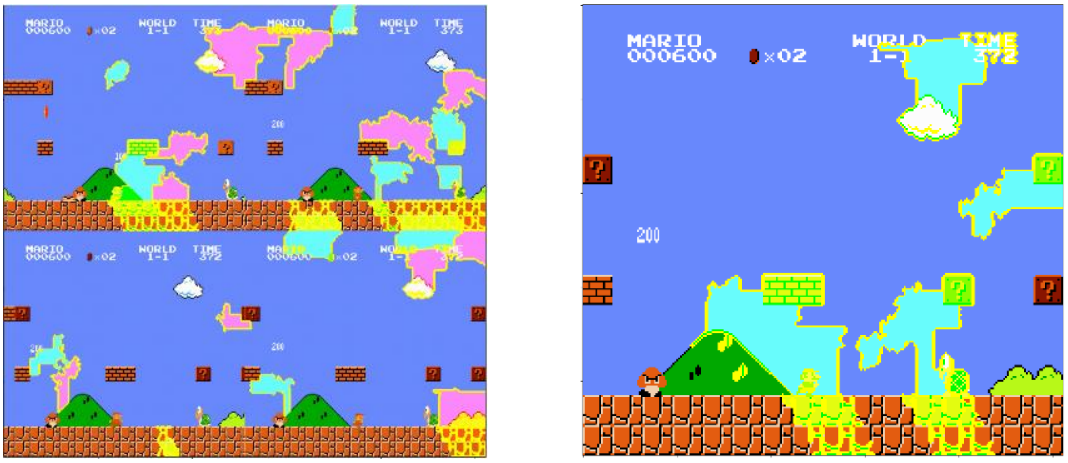

Figure 27 Comparison procedure explainable AI

Figure 27 shows the substantial differences between the two approaches.

Although they are performed with the same parameters in the first image, the highlighted areas are small, scattered, and unclear. In contrast, in the second image, it remains more obvious which areas of the image make a more significant contribution and which areas do not. Also, as mentioned in the implementation, there is no certainty

that the explanation based on the concatenated image is correct. In fact, based on the LIME operation of perturbing the input data in order to understand how the model output changes, it is possible that giving a “distorted” input image may somehow hinder the explanation provided by the explainable AI algorithm.

So, in this case, after comparing the two procedures, it is believed that the best course of action is to use the second procedure articulated as follows:

- give LIME the image to analyze as input
- LIME perturbs the image
- create a net_lime function that takes the three frames preceding the image given as input, creates the state, and feeds it to the network
- then the output obtained looks like the one on the right in *Figure 27*.

CONCLUSION

This thesis aimed at training an agent through the use of reinforcement learning. Reinforcement learning is a branch of machine learning that, through simulation of the environment, trains an agent to act autonomously.

The training is based on the Super Mario Bros video game. Thanks to Open-AI Gym, it was possible to train Mario to win the first level of the game.

Open-AI Gym provides the simulated environment, and thanks to it, it was possible to do several experiments.

The main models were trained on the classical version of the environment. However, since all experiments resulted in some sort of overfitting, an attempt was also made to train based on the random selection of game levels.

Finally, the best-resulting model is the one in which the agent was trained for 40k episodes. Moreover, in the best attempt, the neural network setup is such that it takes as input a state consisting of four stacked frames and returns a vector with scores for each action. In this part, the most compelling point is that, in general, training the model with the above setting (four stacked frames) is better in terms of learning speed than the alternative settings tried. The reason for this assertion is that by giving the model four frames, it can understand the agent's movements. Instead, giving only one frame as a state does not allow the same understanding to the network.

An additional goal of the thesis was to provide frame-based explanations of the actions returned by the neural network. In particular, problems have emerged related to the fact that existing explainable AI algorithms are not adequately adapted to reinforcement learning.

In fact, in this field, working with unlabeled data and simulated environments in video mode could cause obstacles. Specifically, in the paper, the neural network used to train the agent took as input a state consisting of four stacked frames, such that it sensed the agent's movements.

This setup created problems for the use of the explainable AI algorithm, LIME. LIME takes a single image as input and obscures pixels at each iteration to figure out which ones affect the agent's choice the most and which ones affect all other possible actions the most.

Two different strategies have been tried to solve these problems. The procedure that provides the most straightforward explanation is the one that eventually gives a single image as output with the relevant areas highlighted. It consists of giving LIME the original image as input and, instead of the model, giving input to a function that draws the three frames preceding the one of interest, creates the state, and feeds it to the network. Because of this strategy, the explanations received in the output are accurate and understandable.

To conclude, this field of study is still largely under development. Possible development of the topic in the future could be the investigation of explainable AI algorithms and their adaptation to reinforcement learning algorithms. In particular, the thesis provides insights into the application of explainable AI algorithms to reinforcement learning but perhaps some more specific methodology could be developed.

REFERENCES

Knapic, S., Malhi, A., Saluja, R., & Främling, K. (2021). Explainable Artificial Intelligence for Human Decision Support System in the Medical Domain. *Machine Learning & knowledge extraction*, pp. 740 - 770.

Baird, L. C. (1995). Residual algorithms: Reinforcement learning with function approximation. *Proceedings of the Twelfth International Conference on Machine Learning* (pp. pp. 30 - 37). San Francisco: Morgan Kaufmann.

Barto, A. G. (1992). Reinforcement learning and adaptive critic methods. In D. A. White, & D. A. Sofge, *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches* (pp. pp. 469 - 491). New York: Van Nostrand.

Barto, A. G., & Duff, M. (1994). Monte Carlo matrix inversion and reinforcement learning. *Advances in Neural Information Processing Systems: Proceedings of the 1993 conference* (pp. pp. 687 - 694). San Francisco: Morgan Kaufmann.

Choo, B., Crannel, G., Adams, S., Dadgostari, F., & Beling, P. A. (2020). Reinforcement Learning from simulated environments: an encoder decoder framework. *28th High Performance Computing Symposium* . Fairfax Campus, VA, USA.

Dethise, A., Canini, M., & Kandula, S. (2019). Cracking open the black box: what observations can tell us about reinforcement learning agents. *Proceedings of the 2019 Workshop on Network Meets AI & ML*, (pp. pp. 26 - 36). Beijing.

François-Lavet, V., Henderson, P., Islam, R., Bellemare, M. G., & Pineau, J. (2018). *An Introduction to Deep Reinforcement Learning.*

Fukuchi, Y., Osawa , M., Yamakawa, H., & Imai, M. (2017). Application of instruction-based behavior explanation to a reinforcement learning agent with changing policy. *International Conference on Neural Information Processing* (pp. pp. 100 - 108). Springer.

Garreau, D., & Mardaoui, D. (2021). What Does LIME Really See in Images? *Proceedings of the 38 th International Conference on Machine.*

Hailemariam, Y., Yazdinejad, A., Parizi, R. M., Srivastava, G., & Dehghantanha, A. (2020). An Empirical Evaluation of AI Deep Explainable Tools. *2020 IEEE Globecom Workshop.* Taipei, Taiwan: IEEE.

Holzinger, A., Goebel, R., Fong, R., Moon, T., Müller, K.-R., & Samek, W. (2020). *xxAI - Beyond Explainable AI.* Vienna, AU: Springer.

Ignatiev, A. (2020). Towards Trustable Explainable AI. *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence.*

Iyer , R., Li , Y., Lewis, M., Sundar, R., & Sycara, K. (2018). Transparency and explanation in deep reinforcement learning neural networks. *Proceedings of the 2018 AAAI/ACM Conference on AI, Ethics, and Society*, (pp. pp. 144 - 150). New York, NY.

Jakkola, T., Singh, S. P., & Jordan, M. I. (1995). Reinforcement learning algorithm for partially observable Markov decision problems. *Advances in Neural Information Processing Systems: Proceedings of the 1994 Conference* (pp. pp. 345 - 352). Cambridge MA: MIT press.

Lin, L.-J., & Mitchell, T. (1992). Reinforcement learning with hidden states. *Proceedings of the Second International Conference on Simulation of Adaptive Behavior: From Animals to Animats* (pp. pp. 271 - 280). Cambridge, MA: MIT press.

Littman, M. L. (1994). Markov games as a framework for multi-agent reinforcement learning. *Proceedings of the Eleventh International Conference on Machine Learning,* (pp. pp. 157 - 163). San Francisco: Morgan Kaufman.

Lundberg, S. M., & Lee, S.-I. (2017). A Unified Approach to Interpreting Model. *31st Conference on Neural Information Processing Systems*.

Lyu, D., Yang, F., Liu, B., & Gustafson, S. (2019). SDRL: interpretable and data-efficient deep reinforcement learning leveraging symbolic planning. *Proceedings of the AAAI Conference on Artificial Intelligence*, (pp. pp. 2970 - 2977). Honolulu, HI.

Madumal, P., Miller, T., Sonenberg, L., & Vetere, F. (2020). Explainable Reinforcement Learning through a Causal Lens. *The Thirty-Fourth AAAI Conference on Artificial Intelligence.* Victoria, Australia.

Nandy, A., & Biswas, M. (2018). *Reinforcement Learning with Open AI, Tensorflow and Keras using Puthon.* Apress.

Nowé, A., Vrancx, P., & De Hauwere, Y. M. (2012). Game theory and multi-agent reinforcement learning. In *Reinforcement Learning* (pp. pp. 441 - 470). Spring Berlin Heidelberg.

Pan, X., Chen, X., Cai, Q., Canny, J., & Yu, F. (2019). Semantic Predictive Control for Explainable and Efficient Policy Learning. *International Conference on Robotics and Automation.* Montreal, Canada.

Ribeiro, M. T., Singh, S., & Guestrin, C. (2016). "Why Should I Trust You?": Explaining the Predictions of Any Classifier. 1135-1144.

Strumbelj, E., & Kononenko, I. (2010). An efficient explanation of individual classifications using game theory. *Journal of Machine Learning Research*.

Sutton, R. S., & Barto, A. G. (2014). *Reinforcement Learning: An Introduction.* London, England: The MIT Press.

Tabrez, A., Agrawal, S., & Hayes, B. (2019). Explanation-based reward coaching to improve human performance via reinforcement learning. *14th ACM/IEEE International Conference on Human-Robot Interaction*, (pp. pp. 249 - 257).

Van der Velden, B. H., Kuijf, H. J., Gilhuijs, K. G., & Viergever, M. A. (2022). Explainable artificial intelligence (XAI) in deep learning-based medical image analysis. *Medical Image Analysis*.

Van Hasselt, H., Guez, A., & Silver, D. (2016). Deep Reinforcement Learning with Double Q-learning. *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence.*

Wang, H., Wang, Z., & Cui, X. (2021). Multi-objective Optimization Based Deep Reinforcement Learning for Autonomous Driving Policy. *Journal of Physics: Conference Series*.

Watcher, S., Mittelstadt, B., & Russel, C. (2018). Counterfactual explainations withou opening the black box. *Harvard Journal of Law & Technology*.

Wells, L., & Bednarz, T. (2021). Explainable AI and Reinforcement Learning - A Systematic Review of Current Approaches and Trends. *Frontiers in Artificial Intelligence*.

Yuan, S., Zhang, Y., Qie, W., Ma, T., & Li, S. (2019). Deep Reinforcement Learning for Resource Allocation with Network Slicing in Cognitive Radio Network. *Proceedings of the 38 th International Conference on Machine*

ACKNOWLEDGEMENTS

I would like to acknowledge and give my warmest thanks to my supervisor Marco Della Vedova who has been a guide with his valuable advice and his constant encouragement, allowing me to accomplish this thesis.

My sincere thanks also go to professor Jianyi Lin for sharing his knowledge and supporting my work.

I would like to express my heartfelt thanks and gratitude to Università Cattolica del Sacro Cuore for these two years because it has been for me a place of growth not only from an academic perspective but mainly from a personal point of view.

In addition, I am grateful to all my friends and fellow here in Milan and Rimini; I would like to thank them all for the great moments we had and those we will have in the future. Thanks for the late-night studying session and the moral support.

Last but not least, I would like to thank my family for providing me with such an opportunity to pursue this master's degree program and for always believing in me and unconditionally loving me.