# Explainable Reinforcement Learning via Reward Decomposition

Lorenzo Mattia Edoardo Montecchiani

May 7, 2021

## 1 Introduction

The project has been developed starting from a reference paper, providing an implementation of the contents proposed and adding some new proposals. The original paper analyses the application of the algorithm drdQN to two gym environments (Lunar Lander and Cliff World) focusing in particular on some ways to explain the policy learned by the agent. The drdQN algorithm is a version of the dQN algorithm in which the reward is decomposed in significant components making possible to understand which has been the most relevant factor in determining the agent behaviour. Three are the instruments proposed to explain, given two actions *a1*, *a2* and a given state *s*, why the agent has preferred one action with respect to the other :

- Reward Difference Explanations (RDX) defined as the difference of the decomposed Q vectors of two actions in a given state
- Positive Minimal Sufficient Explanation ($MSX^+$) defined as the smallest set of positive RDX components needed to outweigh the summation of negative RDX components
- Negative Minimal Sufficient Explanation ($MSX^-$) defined as the smallest set of negative RDX components to outweigh the $MSX^+$ minus the smallest component.

Once the implementation of the paper's concepts was done, two have been the main focuses of the research. First, a study of the agent's learning per-

formances aimed at their improvement, then the introduction of a new explanation tool, named Minimal Necessary Explanation, trying to increase the explainability of the agent's behaviour.

# 2 Implementation

Starting with no code implementation proposed by the authors of the original paper, this phase has been quite challenging. The first step has been the implementation of a working version of the dQN algorithm, to have a solid base on which it was possible to work properly. Then, it had to be adapted to the reward decomposition algorithm drdQN, and this consisted in two main steps:

- The adaptation of the environment's rewards.
- The adaptation of the policy and the training step.

The environments had a single reward composed by the summation of significant components and this have been changed to an array of these components. For example, in Lunar Lander eight are the main components identified: crash, live, main fuel cost, side fuel cost, angle, contact, distance, velocity.
Following this approach, has been necessary to introduce, in place of a single couple of neural networks (main and target) predicting as output a single Q-value, a couple of networks for each component, predicting one different Q-value per component. So, the training step was organized aiming to minimize the loss of all the main networks, computed as the squared difference of the Q-value predicted by the respective target network ($Q_{t,c}$) in the next state ($s$') considering a new action ($a$') chosen by the policy, plus the reward ($r_c$) for moving to this state and the value predicted by the main network in original state $Q_c(s, a, \theta_c)$.

$$L(\theta_c) = (y_{c,i} - Q_c(s, a, \theta_c))^2$$

$$y_{c,i} = \begin{cases} r_c & \text{for terminal } s' \\ r_c + \gamma Q_{t,c}(s', a', \theta'_c) & \text{for non terminal } s' \end{cases}$$

The network architecture is the same for each component, made by two dense

layer of 32 neurons and a final one with a number of neurons equal to the number of actions.
The training is done every step, based on a batch of touples *(s, a, r, s')* of fixed length (for instance, in this work, 32). The drdQN policy in a given state chooses the action that maximises the summation of the Q-values of all the components.

$$argmax_a \sum_c Q_c^t(s, a)$$

# 3 Research

Once obtained a working version of the drdQN algorithm on both the environments the attentions of the project has moved to a research objective, focused on two different perspectives: efficiency and explainability.

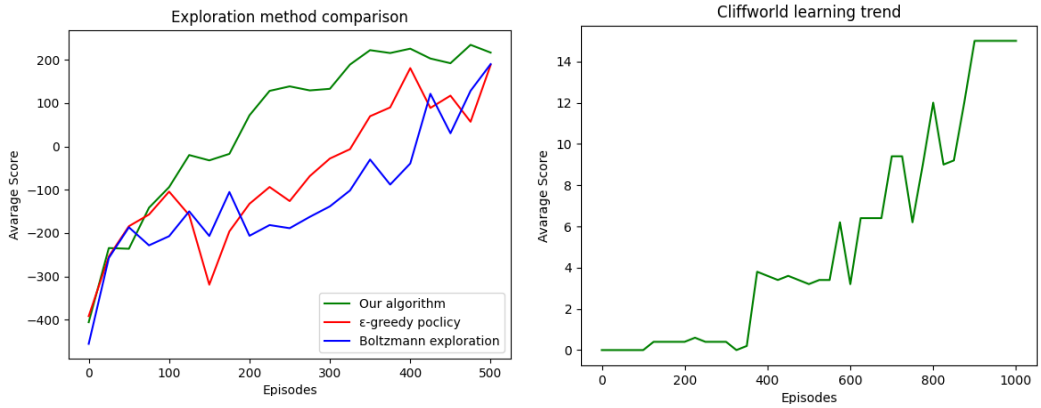
## 3.1 Efficiency: our algorithm proposal

In first place the goal of this phase of research has been a new exploration approach aimed at improving the learning trend of the agent with respect to the $\epsilon$-greedy policy proposed in the original paper. The doubts moving the research interests towards this field were linked to the way of choosing the next action, while the agent had to explore. In fact, according to the $\epsilon$-greedy policy, when the agent has to explore, all the actions have the same probability to be chosen. A better approach could be to weight the probability of each action according to its Q-value. Higher the Q-value, higher the probability. There are in literature some other algorithms approaching the exploration phase in this way, one of this is the Boltzmann one, and on it is based the new proposed approach. It is a sort of mix between the Boltzmann policy and the $\epsilon$-greedy one.
So, there is an hyperparameter $\epsilon$, decaying with the growing of the number of episodes, representing the probability of exploration (like in $\epsilon$-greedy approach) and while exploring, the choice of the action is made with a probability weighted on the action's Q-value. In particular:

**Algorithm 1 our algorithm proposal:** Executed starting from a state $s$ and a value for $\epsilon$

$Qvals \leftarrow ComputeQvalues(s)$
$p = random()$
**if** $p < \epsilon$ **then**
  **for** a in Actions **do**
    $P(a) = \frac{e^{Qvals(a)}}{\sum_{i=1}^{n} e^{Qvals(i)}}$
  **return** random action with probability $P(a)$
**else**
  **return** argmax(Qvals)

In a first phase we have done some experiments using the learning rate proposed in the reference paper, 0.01, but it yielded to a very oscillating learning trend. For this reason we have changed it to a smaller value, 0.0005, giving to the learning trend more stability and as a consequence a faster convergence. After having performed some trials, using this learning rate, it has been noticed that this approach was quite satisfying both in terms of score reached and in number of episodes needed to reach it. But to have a confirmation that the proposed algorithm was good in general, we have performed a comparison between our proposal and two other algorithms, in particular, $\epsilon$-greedy policy and Boltzmann exploration, all applied to two different environments: Lunar lander and Cliff world.
Below the graphs representing the results obtained: (a) on the left the comparison of the three methods in Lunar Lander. (b) On the right our method results applied to Cliff world.
The left graph is computed as the average of 5 runs of the learning algorithm with each point giving the results of 10 test games.(N.B. score 200 means that the agent has been able to land correctly). As it is possible to see around episode 350 the agent trained with the algorithm proposed in this work, is the only one reaching the threshold of 200 of reward score; please note that in a higher number of episodes all the algorithms would have converged.
The right graph instead, represents the average score trend performed by the agent trained with the algorithm proposed in this work, in the Cliff world environment. In this picture it has been decided to not show a comparison with the other methods because with them the agent was never able to reach the "winning" reward threshold of 15, and even worst, to never reach a score higher then 1.




## 3.2 Minimum necessary explanation (MNX)

The reason why it has been chosen to introduce a new method to explain the agent behaviour, with respect to the ones proposed in the reference paper, is to obtain another point of view about the explainability. In particular the aspect that has been focused in the experiments of this work is the relevance of each component in the choice between two actions.
The new method introduced has been named Minimal Necessary Explanation. It is computed starting from the RDX (the difference of the decomposed Q vectors of two actions in a given state), and represents the smallest combination of components for which the remaining ones do not exceed the summation of the negative RDX elements ($d$).

$$MNX = \underset{M \in 2^c}{\operatorname{argmin}} |M| \; s.t. \sum_{c \notin M \wedge \Delta_c > 0} \Delta_c(s, a_1, a_2) < d$$

There may be multiple such sets M, and we break ties by preferring smaller sums $\sum_{c \notin M \wedge \Delta_c > 0} \Delta_c(s, a_1, a_2)$.

Practically, it tells which components, if removed, would change the result of the choice between the two actions. This allows to do extremely interesting considerations about the reward components, making possible to determine approximately which are the most relevant for the success of a policy execution.

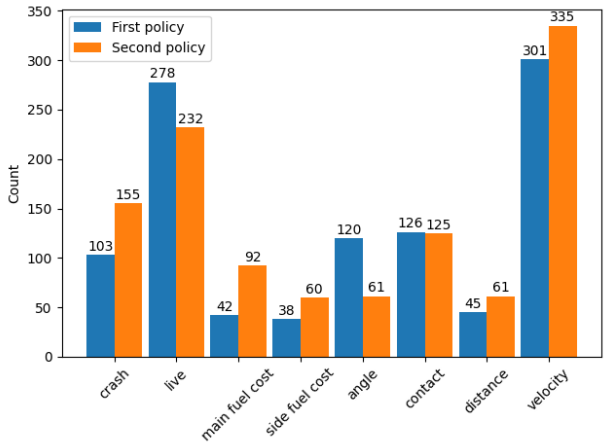



Figure 2: The histogram shows the number of times each component has appeared in the MNX computed executing, with the same seed and a similar number of steps, the two policies taken into account in the examples below.

Consider now, as an example, two situations occurred during tests made in the Lunar Lander environment. Take into account two different learned policies, in the first one the 'angle' component (representing the Q-value obtained for minimizing the angle with respect to the vertical axis) appears in the total MNX, computed with respect to all the actions in all the steps, 120 times; while in the second one it appears just 61 times. Trying to execute again the two policies making this component irrelevant, just not taking it into account during the choice of the best action, in the first one, in which the 'angle' was quite much present in the total MNX, the agent is not more able to land, and crashes. Instead, in the other one, the agent is still able to complete successfully the policy.
A very similar case has occurred for the 'crash' component (representing the Q-value obtained for being far from crashing). It appeared 103 times in the total MNX of the first policy and as a result, removing it in a following execution of the policy the performance has worsened but the agent has been still able to land correctly. Instead, in the second policy, it appeared 155 times and in the following execution without taking into account the 'crash' component, the agent crashed. Thus it is possible to conclude that the MNX allows to have an idea about which are the most relevant components for a correct execution of the policy, even if not always the component appearing more often are the most important.