ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DEPARTMENT OF COMPUTER SCIENCE
AND ENGINEERING

ARTIFICIAL INTELLIGENCE

MASTER THESIS

in

Autonomous and Adaptive Systems

PERFORMANCE ANALYSIS OF THE DESIGN OF EXPERIENCE REPLAY BUFFERS BASED ON INTRINSIC MOTIVATION IN DEEP REINFORCEMENT LEARNING

CANDIDATE	SUPERVISOR
Giulio Vaccari	Chiar.mo Prof.
	Mirco Musolesi
	CO-SUPERVISOR
	Dott. Giorgio Franceschelli

Academic year 2021-2022

Session 3rd

To my family

Abstract

In Reinforcement Learning, an intelligent system, usually referred to as an agent, must learn to maximize a cumulative reward by correctly behaving in an initially unknown environment. In order to improve, the agent must collect feedback from its interactions with the surrounding world, which guides the agent in adapting its actions to achieve better scores. However, there are some environments where feedback is not constantly provided, thus making learning more difficult. In these circumstances, we say that the reward is sparse, and including additional modules in the learning framework can be necessary to improve agent performance.

Methods based on intrinsic motivation try to address the problem of sparse feedback by introducing an additional reward that incentives the agent when its behavior leads it to explore interesting regions of the environment. For example, this reward could be proportional to the novelty of the states visited by the agent during its exploration. In this way, the agent learns to better explore the problem state space, without being blocked by the absence of feedback.

This thesis aims to implement and analyze a new framework for dealing with sparse reward environments. To this end, three different models based on as many intrinsic motivation techniques are implemented. Each model makes use of a prioritized experience replay buffer in which transitions priorities are given by intrinsic motivation scores.

Analysis of the results shows that prioritization based on temporal difference errors remains the best performing approach, but it also revealed an interesting potential in certain categories of intrinsic motivation techniques,

capable of achieving higher scores than those obtained from a uniform priority model.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

The advent of *machine learning* has marked the beginning of a new era in computer science, where programs have the capacity to solve problems for which a solution is difficult or impossible to state formally, using standard programming techniques. In this context, different branches of machine learning have been developed in order to solve problems in different areas.

However, there are still situations in which we are far from fully exploiting the potential of the subject, and among them, there is the increasingly important field of *reinforcement learning*. Reinforcement learning concerns the development of intelligent agents with the task of maximizing a predefined notion of "reward" in a given environment. In practice, this simple definition may include problems that range from simple Atari games to autonomous driving and complex robotic control. Among these, some are more challenging than others: in some games, for example, our agent can be easily guided to the goal by frequent feedback provided by the game itself; in others, our agent may be required to proceed for a long time without any kind of indication about the correctness of its choices, making learning more difficult.

In order to allow a better exploration of the state space in these sparse reward environments, several techniques have been proposed based on the notion of intrinsic motivation [1, 11, 34]. This family of methods introduces an additional reward, i.e the intrinsic reward, that is able to provide feedback also in scenarios where the classical reward is rare. This new feedback rewards the agent for exploring new states that are substantially different from the ones previously visited, in practice rewarding its curiosity.

This research thesis is about implementing and testing a new approach to improve the efficiency of agents used in sparse reward environments, by exploiting the notion of intrinsic motivation to focus the learning of the agent toward its most "curious" past experiences.

To this end, a variant of the classic prioritized experience replay [26] is implemented, in order to analyze the effect that a prioritization based on intrinsic motivation has on agent training.

1.2 Contributions

This thesis aims to investigate a new approach to the resolution of problems in sparse reward environments, by studying what happens when the rewards produced by intrinsic motivation modules are used in the prioritization of transitions inside a prioritized experience replay (PER) [26]. Experience replay [19] allows agents to learn from accumulated experiences and, in its prioritized version, can be used to give some experiences more importance than others.

For the purpose of this thesis, an off-policy deep reinforcement learning model is implemented and tested using prioritized replay buffers based on three intrinsic motivation techniques. These are the count-based method presented in [30], the prediction-based intrinsic curiosity module introduced in [22], and the random network distillation approach of [8]. For each technique, a model is trained by means of an experience replay that uses priority values obtained from the corresponding intrinsic reward function. The performances of the agent with the three replay buffers are evaluated in the MountainCar [21] sparse reward environment, together with two baseline models which use a uniform replay buffer [19] without prioritization and the original prioritized experience replay introduced in [26].

1.3 Structure of the Thesis

The discussion is organized as follows. First, chapter 2 will start introducing some fundamental concepts of Reinforcement Learning. Then, we will deepen into the theory behind the specific agent used to conduct this research. At the end of the chapter, we will discuss the use of intrinsic motivation in the context of reinforcement learning and sparse reward environments.

Chapter 3 will present the core idea of this work. It will introduce the methods that will be compared and analyzed in the following chapters, together with the learning framework used in this thesis.

In chapter 4, we will see the actual implementation of the modules presented in the previous sections. In particular, we will see the implementation details of the agent, the prioritized experience replay, and the three different prioritizers.

Chapter 5 will first discuss the modalities in which the experiments of this thesis will be conducted. It will introduce the environment chosen for the testing of the models, and the definition of the baselines. Then, we will see a comparison of the performances achieved by the tested models, by analyzing their reward curves over the training. At the same time, we will conduct an analysis of the priority values produced by the prioritizers, in order to understand how they affect the agent.

Finally, chapter 6 will summarize the main contributions of this work, discussing the strengths and the limitations of the approach. The thesis will then be concluded with some final considerations and perspectives on possible future work to extend this research.

Chapter 2

Background

2.1 Reinforcement Learning

2.1.1 Introduction to Reinforcement Learning

Reinforcement learning is a branch of machine learning that concerns the development of intelligent agents capable of adapting to an unknown environment in order to maximize a notion of cumulative reward. The agent can obtain rewards from the environment by interacting with it, as shown in figure 2.1.

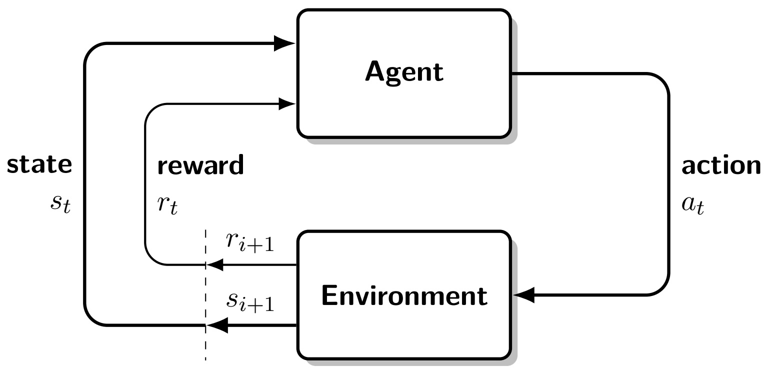

Figure 2.1: RL interaction process in a schematic way

At each time step, the agent is provided with an observation, which is a representation of the current state of the environment. Starting from that observation, the agent decides which action to perform. In response to it, the environment returns a reward signal together with a new observation. That reward will represent the feedback that will be used by the agent to correct its behavior until reaching a good performance. This cycle is repeated several times until the agent terminates its execution.

The agent goal is to learn a way of acting that allows it to maximize the sum of the rewards obtained in a training episode.

2.1.2 Markov Decision Processes

In order to approach the resolution of these agent-environment problems, we first need to define them formally. In reinforcement learning these interactions are stated as Markov Decision Processes (MDP).
We report the MDP formalization presented in [29], where it is defined by the combination of four components:

- $\boldsymbol{S}$ is the set of *states* which represent the possible configurations of the environment

- $\boldsymbol{A(s)}$ is the set of *actions* that the agent can perform from state $s \in S$

- $\boldsymbol{R}$ is the set of *rewards*, usually expressed as real numbers, which are given to the agent as a consequence of its actions

- $\boldsymbol{p(s_{t+1}, r | s_t, a_t)}$ is the function that determines the *dynamics* of the MDP. It gives the probability of leading to a new state $s_{t+1} \in S$ and receiving a reward $r \in R$ by performing action $a_t \in A(s_t)$ from state $s_t \in S$

At each time step t, the agent starts from a state s_t and chooses an action $a_t \in A(s)$ to perform. Then, the environment moves to a new state s_{t+1} and receives a reward $r_t \in R$ following the probability distribution defined by p.

The agent's goal is the maximization of the cumulative reward defined as the sum of all the rewards obtained during its interactions with the environment.

2.1.3 Episodic and Continuing Tasks

Agents can be used in two different types of tasks, based on whether their interactions with the environment can or cannot be broken down into finite sequences of steps.

In *episodic tasks* each agent-environment interaction has an end after a finite amount of time steps. This finite sequence of interactions is called *episode*, and in this context, we can define the cumulative reward achieved by the agent simply as the sum of the rewards obtained at each time step within the episode:

$$G \doteq \sum_{t=0}^{T} r_t \tag{2.1}$$

where T is the length of the episode.

In *continuing tasks* the interactions between the agent and the environment do not necessarily lead to a terminal state. For this reason, we need to introduce an additional term in the definition of the cumulative reward, called *discount rate* $\gamma \in [0, 1]$. The discount rate is used as a coefficient for the rewards achieved at each time step, and it ensures that a possibly infinite summation of rewards leads to a finite result. From an intuitive point of view, the discount rate can be seen as a way to weigh the importance of later rewards against earlier ones. For example, by decreasing γ it is possible to give more importance to rewards obtained earlier in the episode compared to the ones achieved later.

The cumulative reward for continuing tasks is defined as:

$$G \doteq \sum_{t=0}^{\infty} \gamma^t r_t \tag{2.2}$$

Because of the positive effect that the discount rate has on the agent's perception of reward, the use of γ is also extended to episodic tasks.

2.1.4 Policies and Value Functions

In the setting of a Markov decision process, we need a way to formalize the current behavior of the agent, in order to be able to measure and improve its performance. In reinforcement learning, a *policy* π is a probability distribution function defined over the set of possible actions $A(s)$ conditioned by a state s of the Markov decision process. In practice, a policy π encodes the behavior of the agent by associating to each action available from a given state a probability value that indicates how much is probable that the agent would take that action starting from that state.

Given the policy that represents the behavior of an agent, we can now define a function to estimate its performance. The *value function* $v_\pi(s)$ gives the expected cumulative return obtainable starting from the state v and following the policy π afterward. $v_\pi(s)$ is also called *state-value function*, and it can be formalized as follows:

$$v_\pi(s) \doteq E_\pi[G_t|s_t = s] = E_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s] \tag{2.3}$$

where $E[\cdot]$ denotes the expected value of a random variable assuming of following the policy π at each time step k greater than t.

In a similar way, it is possible to define another very useful function called *action-value function* $q_\pi(s, a)$, which represents the expected return starting from state s and immediately taking action a, then following the policy π:

$$q_\pi(s, a) \doteq E_\pi[G_t|s_t = s, a_t = a] = E_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s, a_t = a] \tag{2.4}$$

These value functions are particularly important because they are strictly connected with the concept of *optimal policy*. In order to solve a reinforcement learning problem, we need to find a policy that maximizes the amount of reward collected over the long run, and in a Markov decision process, there is always at least one policy that is optimal compared to the others. We denote a generic optimal policy with π^*, and we define its associated optimal value function v^* as follows:

$$v(s)^* \doteq \max_{\pi} v_\pi(s) \quad \forall s \in S \tag{2.5}$$

Note that every optimal policy π^* is characterized by the same value function v^*. In the same way, we can define the optimal action-value function q^* as:

$$q^* \doteq \max_{\pi} q_\pi(s,a) \quad \forall s \in S \quad and \quad \forall a \in A \tag{2.6}$$

As can be easily observed, once we have an optimal value function, it is possible to obtain an optimal policy by simply taking in each state the action which maximizes the value function. Thanks to this result, we can reduce the resolution of a reinforcement learning problem to the one of obtaining an optimal value function to follow. But how can we compute an optimal value function for our problem?

The first possibility is to use the *Bellman optimality equation*, which gives us a way to compute it exactly. First of all, let express a generic state-value function v_π of a policy π as follows:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')] \tag{2.7}$$

where $a \in A(s)$ for each state s, and s' are possible successor states of s with rewards r associated to the transitions.
We can then rewrite the previous equation under the optimal policy, which assumes the following form:

$$v^*(s) = \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma v^*(s')] \tag{2.8}$$

Bellman's equations allow expressing the value of $v^*(s)$ recursively in a very elegant form, whose application, however, is not always feasible. In fact, in order to use Bellman's equations to compute $v^*(s)$, we would need to know the full dynamics of the environment, which however are usually unknown at the start of the MDP. In addition to this, if the MDP is very complex, the use of Bellman's optimality equations is computationally infeasible.

The second possibility for the computation of v^* consists in estimating it using past experiences collected by the agent during its training inside the MDP. There are several ways to accomplish this task, but in this thesis we will concentrate on a class of techniques that allows the agent to learn a policy from past transitions between pairs of states, which is called *temporal difference learning*.

2.1.5 Temporal Difference Learning

Temporal difference learning refers to a family of reinforcement learning methods that are able to learn a policy starting from a set of MDP transitions in the form $(\boldsymbol{S_t}, \boldsymbol{A_t}, \boldsymbol{R_t}, \boldsymbol{S_{t+1}}, \boldsymbol{A_{t+1}})$, where:

- $\boldsymbol{S_t}$ is the starting state
- $\boldsymbol{A_t}$ is the action performed from the starting state $\boldsymbol{S_t}$
- $\boldsymbol{R_t}$ is the reward received in the transition
- $\boldsymbol{S_{t+1}}$ is the state reached after the transition
- $\boldsymbol{A_{t+1}}$ is the successive action performed from $\boldsymbol{S_{t+1}}$

Generally, these transitions are used to learn an action-value function $q(s, a)$ in an iterative estimation process. Learning q_π for the current policy π is useful because q_π can then be used to improve the policy π itself. Since a change in π is reflected in a change in q_π, we need to keep updating q until the training converges. The repetition of this cycle should lead to the improvement of the policy and therefore to the improvement of the behavior of our agent.

The simplest method of this family of techniques is called *TD(0)*, introduced by Sutton in [28]. TD(0) is designed to solve the easier problem of estimating the value function v_π associated with a given policy π. In order to do so, it uses the following formula to compute the next approximation of V:

$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) \tag{2.9}$$

where γ is the discount rate and α is the learning rate.
The TD(0) update of $V(S_t)$ involves the use of *bootstrapping*, where current estimates of V are used to update V itself. As we can see, the update requires information that can be obtained in a single transition from time step t to time step $t+1$ together with the action that will be selected in the subsequent time step, thus making this method especially useful in continuing tasks or in general when we don't want to wait for the end of an episode to perform an update.

Looking closely at the previous formula, we can individuate a term which represents the discrepancy between the current estimate of $V(S_t)$ and its new updated estimation after observing the transition:

$$\delta_t = \overbrace{R_{t+1} + \gamma V(S_{t+1})}^{Target} - V(S_t) \tag{2.10}$$

This is called the *Temporal Difference (TD) Error* of the transition and consists of the difference between a target and the current estimate $V(S_t)$. The target in TD(0) can be seen as an approximation of the real value of $V(S_t)$, inferred from the transition.

For any fixed policy π, it is possible to prove that this method converges to v_π under some simple assumptions concerning the learning rate, as stated in [29].

Finally, here is the full pseudocode for TD(0):

Algorithm 1: TD(0)

```
Input: policy $\pi$, learning rate $\alpha \in (0, 1]$
Output: value function $V$
Initialize $V$ arbitrarily, except that $V(s) = 0$ if $s$ is terminal
foreach episode do
    Initialize $S_0$
    foreach step t in episode do
        Choose action $A_t$ using policy $\pi$ from state $S_t$
        Take action $A_t$ and observe $R_t, S_{t+1}$
        $V(S_t) \leftarrow V(S_t) + \alpha(R_t + \gamma V(S_{t+1}) - V(S_t))$
        $S_t \leftarrow S_{t+1}$
    end
end
return $V$
```

2.1.6 On-Policy and Off-Policy Algorithms

As seen in temporal difference learning, it is possible for an agent to learn a policy starting from a set of experiences, i.e. the transitions, that can be obtained by exploring the environment. In order to explore, the agent must follow a policy that determines its behavior when deciding which actions to perform from each visited state. We can refer to this policy as the *behavioral policy* of the agent.

In reinforcement learning, there is a distinction between agents whose behavioral policy coincides with the current policy being trained, i.e. the *target policy*, and agents for which the two do not necessarily coincide.

Formally, we say that a learning algorithm is *on-policy* when the exploration of the states follows the target policy. Otherwise, we say that the learning algorithm is *off-policy*.

Algorithms that perform training on-policy usually do so because the learning process is based on successive improvements of the current policy that follows a local search approach. In this setting, it is necessary for the transitions used to construct the updates to be obtained considering the current state of the policy and not an old version of it. On the other hand, algorithms that perform training off-policy do not require this constraint on the transitions. Off-policy methods try to make the agent converge directly to an optimal policy using experiences that can be obtained also without following the actual policy. For example, an off-policy agent can be trained also using transitions that are explored some time before in training, while for an on-policy agent this could lead to wrong policy updates.

2.1.7 Experience Replay Buffers

Instead of directly using the obtained experience to perform a policy update, off-policy algorithms make use of a data structure called *experience replay buffer* [19].

An experience replay buffer is a memory where transitions collected by the agent are stored in order to be used later for training the policy. Each time a new transition is collected, it is stored in the buffer instead of being immediately used for computing an update. Then, at regular intervals, a batch of transitions is sampled from the buffer and an update is performed using those experiences. The main utility of a replay buffer comes from the fact that it helps reduce the temporal correlations between transitions used to compute updates, which helps the training of the agent.

Experience replay buffers can be safely used only when the training algorithm expects transitions that can be obtained with a different version of the policy. For this reason they are generally used with off-policy algorithms, while they are usually avoided with on-policy approaches.

2.1.8 Prioritized Experience Replay

In a classic experience replay buffer, transitions are sampled with uniform probability to construct the policy updates. Vice versa, a *Prioritized Experience Replay (PER)* [26] is a variant of the classic experience replay buffer in which transitions stored in the buffer are not sampled with uniform probability, but instead with a probability given by their priorities.

According to the original paper, each time a new transition is obtained from the environment and inserted in the buffer, its priority is set to a maximum value to ensure that it is used for training the agent at least one time. Then, after a batch of transitions are sampled for computing an update, their priorities are updated with values proportional to how much the agent has still to learn from those experiences. Specifically, the updated priorities are given by the temporal difference (TD) errors computed in the last policy update.

When sampling a batch of experiences from the buffer, each transition has a probability of being chosen which is monotonic in its priority, according to the following formula:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{2.11}$$

where $p_i > 0$ is the priority of transition i. The hyper-parameter α decides the smoothing of the probabilities: it determines how much prioritization is used.

The use of prioritized sampling, however, presents an issue: it introduces bias in the training because it alters the distribution of the data used to construct the policy's updates. This could have a negative impact on the final performance of the agent, and therefore it needs to be corrected. To solve this problem, PER uses an additional mechanism to control the importance given during training to each transition sampled, which is the *weighted importance sampling* (weighted IS). With this approach, the contribution of a sampled transition is modulated by an IS weight calculated as follows:

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^\beta \tag{2.12}$$

When constructing the updates, these weights will be multiplied by the temporal difference errors associated with the transitions in question with the aim of modifying how much they affect the policy. The value of β determines the compensation factor for the data distribution, which fully compensates for the non-uniform probabilities when $\beta = 1$. Conceptually, at the beginning of training this value should be low, in order to exploit the benefits of prioritization. Then, it should gradually increase until reaching a value near 1. In fact, in the early stages of training the alteration of the data distribution is not so problematic, while it is very important to correct it toward its end.

As stated in [26], the introduction of PER allows to speed up learning by a factor of 2 on the Atari benchmark [4] compared to the classical replay buffer. This is possible thanks to the fact that during the training of the policy, precedence is given to those experiences where the agent obtained larger errors, from which it can therefore learn more.

2.2 Agents

2.2.1 Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) comes from the combination of classic reinforcement learning with deep learning. The introduction of deep networks inside agents' architectures was necessary to deal with complex environments that can be seen in real-life problems. In particular, deep networks become very useful every time we need to process high-dimensional state representations. In classical reinforcement learning, a temporal difference algorithm would use a simple table for implementing the action-value function $q(s, a)$. The problem is that if the state space cardinality is too vast, the size of the table would be too large to be manageable. For this reason, we need to replace the tabular implementation of $q(s, a)$ with an *approximation* of it that exploits the capabilities of deep learning.

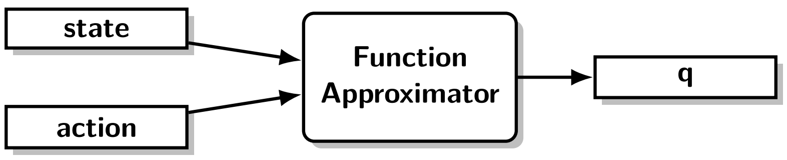

Figure 2.2: Computation of $q(s, a)$ using a function approximator

Using a deep network as a function approximation of $q(s, a)$ will provide a mapping between states (or action-state pairs) to state (or action-state) values, as seen in figure 2.2. The training of the DRL agent will then consists in training its deep networks, which can be performed by integrating reinforcement learning algorithms with gradient descent techniques.

2.2.2 Value-Based and Policy-Gradient Methods

Historically, deep reinforcement learning algorithms can be divided into two main categories, which are the *value-based methods* and the *policy-gradient methods*. What characterizes the two approaches is the way they learn a policy from the experiences collected by the agent.

Value-based methods rely on learning an action-value function $q(s, a)$ that will be used to derive it. The value function can be estimated in different ways, for example using a temporal difference approach as seen previously with the TD(0) algorithm, or also with other techniques like Monte Carlo methods [29]. In addition, the estimation of a value function during the training of a policy π can be performed following either an on-policy or an off-policy methodology, resulting in methods like Semi-Gradient SARSA [24, 29] and Deep Q-Learning [20].

In DRL the current estimation $\hat{q}$ of the action-value function is implemented using a neural network which is trained by updating its weights vector θ. The general gradient descent update for value-based algorithms has the following form:

$$\theta_{t+1} = \theta_t + \alpha[U_t - \hat{q}(s_t, a_t; \theta_t)]\nabla\hat{q}(s_t, a_t; \theta_t) \tag{2.13}$$

where α is the learning rate, $\nabla\hat{q}(s_t, a_t; \theta_t)$ is the gradient of $\hat{q}$ with respect to the state and the actions selected at time step t, and U_t is the target of the update. U_t represents the value that $\hat{q}(s_t, a_t; \theta_t)$ should assume, and the difference between the two quantities is the error that will determine the intensity and the direction of the gradient update. In general U_t is unknown, so an estimation of it is used instead. The way this estimation is constructed differs among the various value-based algorithms.

Starting from the actual estimate $\hat{q}$ of the action-value function, value-based methods implement their policy following the actions with the highest value from the current state s_t:

$$a_t = \underset{a}{\operatorname{argmax}}\, \hat{q}(s_t, a) \tag{2.14}$$

Unfortunately, the above method typically does not lead to a sufficient exploration of the problem space, making learning very difficult. A simple solution to this problem consists in using the so-called *ϵ-greedy search*, which introduces a stochastic component in the agent's decision process. At each time step, the agent has a probability equal to $\epsilon \in [0, 1]$ to perform an action chosen at random instead of performing the best one according to the value function. Usually, the ϵ value is high at the start of the training and then is decreased until it reaches a value near 0.

On the other hand, policy-gradient approaches eliminate the necessity of learning a value function and directly train a policy using the experiences collected by the agent. As stated in the previous sections, a policy is represented as a mapping π between states and action probabilities, and it can be used to determine how much is probable that the agent at time step t will perform a certain action from the given state s_t:

$$\pi(a, s; \theta) = Pr(a_t = a | s_t = s, \theta_t = \theta) \tag{2.15}$$

θ here is the vector that represents the weights of the neural network that encodes π, which is called *policy network*. In policy-gradient methods, the policy network is trained directly from transitions (or from full trajectories of transitions), by using gradient ascent updates. A basic example of this approach can be seen in the classic REINFORCE algorithm [35].

Although the classification introduced in this section is certainly important from a conceptual point of view, there are also algorithms that do not belong to either category, or that derive from a combination of the two. Actor-Critic algorithms are the best example of an architecture that manages to exploit the best of both approaches, combining the direct training of a probabilistic policy, with that of a value function that is used to drive the improvement of the previous one. For further discussion of the above methods, we recommend reading [29].

2.2.3 Q-Learning and Extensions

Q-Learning [33] is a classical value-based reinforcement learning algorithm belonging to the family of temporal difference methods. The training of a Q-Learning agent consists in learning an action-value function $Q(s, a)$, which will be used within an ϵ-greedy exploration strategy. The update of the current Q function is performed off-policy, making Q directly approximate the optimal value function q^*:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha(R_t + \gamma \max_A Q(S_{t+1}, A) - Q(S_t, A_t)) \tag{2.16}$$

As we can see, the algorithm is off-policy because the update's target uses the best action from S_{t+1} (according to the current Q) instead of choosing the one actually used in the transition. Thanks to this, it is possible to train the agent using an experience replay buffer, increasing its convergence speed.

Q-Learning has been adapted to be used with deep learning, resulting in the *Deep Q-Learning* algorithm presented in [20] by Mnih *et al.*, usually abbreviated in DQN for *Deep Q-Network*. In this version, a deep neural network

Algorithm 2: Q-Learning

```
Input: $\epsilon > 0$, learning rate $\alpha \in (0, 1]$
Output: action-value function $Q$
Initialize $Q(S, A)$ arbitrarily, except for terminal states
foreach episode do
    Observe $S_0$
    foreach step t in episode do
        Choose action $A_t$ at random with probability $\epsilon$
        otherwise $A_t = \max_A Q(S_t, A)$
        Take action $A_t$ and observe $R_t, S_{t+1}$
        $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_t + \gamma \max_A Q(S_{t+1}, A) - Q(S_t, A_t))$
        $S_t \leftarrow S_{t+1}$
    end
end
return $Q$
```

is used to implement the action-value function $\hat{q}(s, a)$, which is trained by gradient descent using the following adaptation of formula 2.16:

$$\theta_{t+1} = \theta_t + \alpha[R_t + \gamma \max_A \hat{q}(S_{t+1}, A; \theta_t) - \hat{q}(S_t, A_t; \theta_t)]\nabla\hat{q}(S_t, A_t; \theta_t) \tag{2.17}$$

where θ_t is the weight vector of the network at time t.
In [20], a DQN extended with an experience replay buffer was used for the first time to solve a large number of Atari games. The observations provided by the game environment were in the form of tensors of pixels, representing a concatenation of visual frames from the game screen. For this reason, the neural architecture of the agent included also a first set of convolutional layers that allowed the processing of those frames.

Over the years, the classic DQN model has undergone some structural changes with the aim of improving its performance. Next we will look at the main ones, analyzing how they affect the basic model.

In 2015, Wang *et al.* introduced in [32] a modified version of DQN called *Dueling DQN*. In particular, this variant changed the final layer of the neural network, in which the computation of $\hat{q}(S, A)$ values was broken down into two sub-problems. In figure 2.3 we can see this new architecture compared to the classic one.

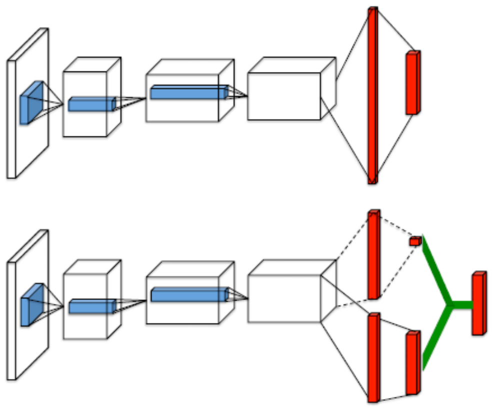
Figure 2.3: Dueling architecture from [32]. A classic DQN (**top**) and the dueling DQN (**bottom**).

In a dueling DQN, $q_\pi(s, a)$ is decomposed as the sum of:

- $v_\pi(s)$: the state value of s
- $A_\pi(s, a)$ the *advantage* of taking action a in state s

by doing so, we have:

$$q_\pi(s, a) = v_\pi(s) + A_\pi(s, a) \tag{2.18}$$

The computation of the q-values is divided into two streams, which are then combined to produce an action-value for each possible action.
As explained by the authors:

> "Intuitively, the dueling architecture can learn which states are (or are not) valuable, without having to learn the effect of each action for each state. This is particularly useful in states where actions do not affect the environment in any relevant way."

Unfortunately, if we combine the two quantities by simply summing them together, we will not be able to correctly distribute the error among the two streams during backpropagation. The authors proposed a solution to this problem which is to force $A_\pi(s, a)$ to be equal to zero when a is the best action according to the current policy.

$$q_\pi(s, a) = v_\pi(s) + (A_\pi(s, a) - \max_a A_\pi(s, a)) \tag{2.19}$$

Experimentally, replacing max with a $mean$ proved to be a better solution, as it increases stability during training.

$$q_\pi(s, a) = v_\pi(s) + (A_\pi(s, a) - \operatorname*{mean}_a A_\pi(s, a)) \tag{2.20}$$

The idea behind the second upgrade we are about to see tries to solve an inherent problem of Q-Learning through the introduction of an additional estimator, i.e. an additional neural network in the context of DRL. Precisely because of this two-network architecture, the resulting algorithm takes the name of *Double DQN*.

In classic Deep Q-Learning, the approximation of the target in the update formula is computed with a maximum over all the possible actions from the state s_{t+1}. Since this approximation is noisy, it is possible to prove that this approach leads to an overestimation of the correct action value. To solve the problem and eliminate this bias, Double Q-Learning introduces an additional network Q', named *target network*. The target network has the same structure as the *primary* one Q, and it is used to decouple the choice of the action from the estimation of its value.

In the computation of the update target, a Double Deep Q-Network model (DDQN) first uses its primary network to choose the action with maximum value from state S_{t+1}:

$$target\_action = \underset{A}{\operatorname{argmax}}\, Q'(S_{t+1}, A; \theta'_t) \tag{2.21}$$

Then, the target network is used to evaluate that action in the construction of the network update (instead of using the value obtained from the primary networks). So, the target used in the DDQN update formula will have the following form:

$$Y_t = R_t + \gamma Q'(S_{t+1}, \underset{A}{\operatorname{argmax}}\, Q(S_{t+1}, A; \theta_t); \theta'_t) - Q(S_t, A_t; \theta_t) \tag{2.22}$$

As we can see, Q selects the action, while Q' evaluates it.
While the weights θ of the primary network are updated using gradient descent with the target above, the weights θ' of the target network are updated periodically by copying the ones of the primary network every τ training steps.

The technique we just discussed comes from the paper [31] by Hasselt *et al.*. This, in turn, is an adaptation of the idea introduced in [14] by Hasselt, in the context of the classic Q-Learning. There is also an additional version introduced by Fujimoto *et al* [13] in the context of actor-critic methods.

Dueling DQN and Double DQN are two independent variants of the classic DQN, but as we will see in later chapters, our experiments are based on an agent that makes use of both upgrades. In fact, ours will be a Double DQN architecture in which each of the two networks has a dueling final layer.

2.3 Intrinsic Motivation

2.3.1 Challenges of Sparse Reward Environments

As observed in the previous algorithms, the role of the reward signal is crucial to guide the learning of the agent. The updates for the agent's policy are constructed using immediate and predicted reward values; without them, the agent simply could not learn which actions are good and which are not. These rewards are generated directly from the environment, and often they are sufficiently dense to allow the agent to learn with continuity. For example, in the famous ATARI game **Breakout** (figure 2.4), the agent is rewarded each time it increases its score, which happens every time it manages to hit a tile with the ball.

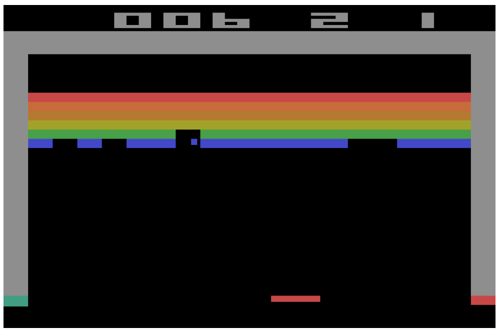

Figure 2.4: ATARI Breakout

However, in many real-life problems, the feedback provided by the environment is not frequent, or even rarely observable. In these situations, we say that the reward is *sparse*.

In sparse reward environments, learning becomes more difficult because the agent might need to try a very large number of interactions before obtaining a single useful feedback, and in that time it would be unable to learn anything. A famous example of this kind of environment is the **Mountain Car** problem, which first appeared in [21]. In Mountain Car the agent controls a little car positioned between two hills in a bi-dimensional environment (figure 2.5). The goal is to learn a way to strategically accelerate the car to reach the top of the right hill. In order to achieve sufficient momentum to escape the valley, the car needs to be pushed a bit in the opposite direction before moving toward the right hill. The difficulty of this problem lies in the fact that the only reward provided is a constant -1 value at each time step until the agent reaches the final position, or until the time runs out.

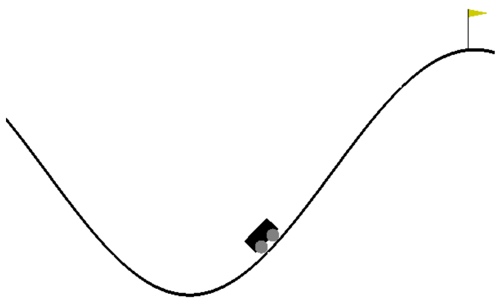

Figure 2.5: Mountain Car from OpenAI Gym

The agent won't be able to learn anything until a random combination of its moves takes it to the hilltop goal at least once. From that moment, it will be possible for the obtained positive feedback to be propagated throughout the agent's policy, allowing it to learn the effects of its own moves. If the state space of the problem is large, this kind of approach could not be feasible.

2.3.2 Curiosity as a Reward

To deal with the difficulties of sparse reward environments, an interesting solution has been proposed which draws inspiration from the notion of *intrinsic motivation* [11] used in the field of human psychology. Intrinsic motivation refers to the concept of internal gratification that drives people toward doing activities for inherent satisfaction, and not for achieving an external goal [25]. For example, human curiosity is a form of intrinsic motivation that rewards exploratory behaviors in unfamiliar environments. This kind of process can be seen easily in young infants that try to interact with every new object they encounter [1].

In reinforcement learning, intrinsic motivation can be embedded inside training in order to reward agents when they perform actions that lead them to discover new states of the environment or to try new kinds of interactions. As summarized in [34], there are several possible ways to integrate intrinsic motivation to guide learning, but generally, this is implemented as an additional reward that is given to the agent to encourage it to explore more. This additional reward is called *intrinsic reward* because it is generated inside the agent to reward its intrinsic behavior, independently from the environment's goal. Usually, this reward value is computed at each training step based on the characteristic of the current state that has just been discovered, along with other information related to past transitions and the current behavior of the agent. Intrinsic reward is opposed to the classic reward we refer to as *extrinsic*. Both intrinsic and extrinsic rewards are then commonly combined to obtain the feedback that will be given to the agent, which will be used to compute the policy updates. The introduction of this new form of combined reward is able to change the policy that is learned by the agent: the latter should find a balance between the maximization of the external reward (which remains the main goal) and fruitful exploration of the surrounding world.

In the following sections, we will look at some intrinsic motivation techniques used in reinforcement learning that differ in the way the intrinsic reward is computed.

A first family of techniques comprehends the *count-based methods* for the generation of intrinsic feedback. Count-based methods generate the intrinsic reward at each training step by looking at the state s_{t+1} just reached in the current agent's transition. The intrinsic feedback is then computed as an inverse function of the number of times a state similar to s_{t+1} has already been seen. In general, these methods try to estimate the novelty of the states visited by the agent in order to encourage the exploration of new states dissimilar from the ones already encountered. Certainly, there are many methods belonging to this category that would be interesting to delve into. An example of a count-based strategy to guide exploration can be seen in the classic UCB algorithm introduced in [18], in which the action selection is influenced by a bonus given by the number of times each action has already been chosen in past transitions. The approach presented by Bellemare *et al.* in [5] instead constructs a density model as a way of representing the information about the past states explored by the agent. This model is then used to derive pseudo-counts from which to obtain the intrinsic reward values.

The second category of techniques that we will discuss comprehends the *prediction-based methods*. These methods start from the idea that the intrinsic reward should encourage the agent to improve its knowledge about the dynamics of the environment. In order to do so, a prediction model is incorporated into the training algorithm. This will be used to define a prediction problem which should give us a measure of how familiar the agent is with the dynamics of the environment. Then, the intrinsic reward will be computed as a function of the error in the agent prediction. In fact, the goal of this family of methods is to encourage the agent to concentrate the exploration in those states where its predictive abilities are poorer, in which the agent has probably been trained less and has more to learn. Even in this context, there are several implementations of the idea underlying this category. Houthooft *et al.* in VIME [15] proposed a strategy based on variational inference in Bayesian neural networks [6], consisting in the maximization of *information gain* about the agent's belief of environment dynamics. Another interesting approach introduced by Pathak *et al.* in [23] involves the use of an ensemble of prediction models instead of just one. In this strategy, the intrinsic rewards are computed by measuring the disagreement between the predictions of the different models in the ensemble: higher the variance across the outputs of the models, higher the reward.

Among the algorithms we will use, there is one that could be considered as a hybrid approach between the two strategies previously introduced. Here the novelty of the states is taken into consideration to determine the intrinsic reward to assign in a transition, but on the other hand, a prediction problem is used in order to generate the feedback.

There are also other categories of intrinsic motivation techniques that, although not explored in depth in this thesis, deserve close attention. Among these we find *memory-based methods*, whose distinguishing feature is that they make use of an external memory to solve some of the problems encountered by other approaches. An excellent example can be found in the recent Never Give Up [2] method by Badia *et al.*, which makes use of an episodic memory-based intrinsic reward that combines per-episode and life-long novelty.

The are also other promising approaches to encourage an intelligent exploration of the search space which does not involve the use of an explicit intrinsic reward. For example in the Exploration via Empowerment Gain [3] presented by Becker-Ehmck *et al.* the agent exploration is driven by the notion of empowerment [17], which is a measurement of the perceived control over its environment. Finally, in [12] Ecoffet *et al.* propose a new approach for hard-exploration problems which involves a multi-step algorithm to improve the exploratory capabilities of the agent.

2.3.3 Count After Hashing

Classical count-based techniques used in the context of tabular reinforcement learning directly counted the number of times each state (or each state-action pair) has been visited during training in order to produce the intrinsic feedback [27].

In 2016 Tang *et al.* presented a count-based method [30] for the generation of intrinsic rewards which was capable of working with very large state spaces and with high-dimensionality observations, i.e., when tabular methods can not be used.

This approach also uses a table in which to store information about the past transitions of the agent; however, this is no longer associated with individual states, but rather with sets of states.

For this purpose, a hash function ϕ is introduced to map each state to an encoding in $\mathbb{Z}^k$:

$$\phi : S \to \mathbb{Z}^k \tag{2.23}$$

The main idea is to define ϕ as a locality-sensitive hashing (LSH) function, so it will map similar states to the same discrete encoding.

Each time a state s is visited during the training of the agent, ϕ is used to obtain its encoding $\phi(s)$. Then, this code is used as an index in a hash table $n(\cdot)$ that keeps track of the number of times a state with that code has already been seen.

The intrinsic reward $r^i(s)$ is computed as a function of this count, and it is simply added as a bonus to the extrinsic reward r^e to obtain the final reward given to the agent:

$$r^i(s) = \frac{\beta}{\sqrt{n(\phi(s))}} \tag{2.24}$$

$$r_t(s) = r_t^e + r_t^i(s) \tag{2.25}$$

where $\beta \in \mathbb{R}_{\geq 0}$ is the bonus coefficient. The higher the number of times a state similar to s has already been explored, the lower the intrinsic reward will be. Initially all the counts in $n(\cdot)$ are set to zero. Then at each time step t, the count $n(\phi(s_t))$ associated with the current state s_t is increased by one.

The hashing function used in the original paper is SimHash [9], a computationally efficient type of LSH:

$$\phi(s) = \operatorname{sgn}(Ag(s)) \in \{-1, 1\}^k \tag{2.26}$$

where $g : S \to \mathbb{R}^D$ is an optional preprocessing function and A is a $k \times D$ matrix with i.i.d entries drawn from a standard Gaussian distribution $\mathcal{N}(0, 1)$. The value k controls the granularity of the discretized state space.

The use of an additional function g can be very useful when working in complex environments, such as when observations are images. As mentioned by the authors, when working in an environment with visual observations such as those of Atari games, the direct use of SimHash gives poor results. For this reason, the encoding part of a neural autoencoder (AE) was used as g preprocessing function. In the middle of the AE architecture a layer whose output $b(s)$ is given by D sigmoid functions is considered. The g function is defined as the output $b(s)$ rounded to the nearest integer:

$$g(s) = \lfloor b(s) \rceil \in \{0, 1\}^D \tag{2.27}$$

We will not elaborate further on the implementation details of the AE, since no visual environments will be used in the experiments covered by this thesis, and therefore we will not make use of it. We recommend reading the original paper to delve deeper into this topic.

2.3.4 Intrinsic Curiosity Module

In 2017 Pathak *et al.* introduced in [22] a prediction-based technique of intrinsic motivation for reinforcement learning which determines the value of the intrinsic reward on the basis of the agent's current understanding of the environment. To do so, the agent learns to predict the most probable evolution of a given state given a possible action. Then, the intrinsic reward is computed as a function of the error in the agent prediction.

The method proposed in [22] generates intrinsic rewards using a module named *Intrinsic Curiosity Module (ICM)*, which formulates curiosity as the error of the agent in predicting the consequence of its own actions, by following the strategy explained above. The main contribution introduced by ICM is that before facing the prediction problem, the state representations are mapped in a feature space designed to give importance to only those characteristics of the observations relevant to the prediction task. In particular, by using this feature space, the states embeddings will contain only the information relevant to the action performed by the agent, thus focusing the prediction on the aspects of the observations that can be influenced by the behavior of the agent, ignoring the rest. This is useful to avoid a series of problems that can occur when working within the raw sensory space, ranging from the high dimensionality of the observations, which can make the prediction problem unnecessarily difficult, to the famous "Noisy-TV problem" (as explained in [22]). The feature space is learned directly during the training of the agent, using a self-supervised inverse dynamics model.

ICM is made of two neural sub-models, the *inverse dynamics model* and the *forward dynamics model*, which are trained together with the agent's policy. The inverse dynamics model consists in two components that are closely connected: the first is the *embedding network* ϕ, which maps a raw state s into a feature vector $\phi(s)$. The second component takes as input $\phi(s_t)$ and $\phi(s_{t+1})$ and predicts the action a_t causing the transition. The second component's purpose is to be used to train ϕ, starting with the idea that if the embedding function is optimized to solve the action-prediction problem, then it will produce good representations of the states which will be useful also for the original prediction problem.

The two combined components define the inverse dynamics model g:

$$\hat{a}_t = g(s_t, s_{t+1}; \theta_I) \tag{2.28}$$

where, $\hat{a}_t$ is the predicted estimate of the action a_t. The neural network weights θ_I are trained in order to minimize a loss function L_I which measures the discrepancy between $\hat{a}_t$ and a_t. The tuples (s_t, a_t, s_{t+1}) required to learn g are obtained while the agent interacts with the environment using its current policy $\pi(s)$.

The forward dynamics model consists of a neural network that implements the function f, which given the inputs $\phi(s_t)$ and a_t predicts the encoded next state $\phi(s_{t+1})$,

$$\hat{\phi}(s_{t+1}) = f(\phi(s_t), a_t; \theta_F) \tag{2.29}$$

where $\hat{\phi}(s_{t+1})$ is the predicted estimate of $\phi(s_{t+1})$. The weights θ_F are learned by minimizing the loss function L_F:

$$L_F(\phi(s_{t+1}), \hat{\phi}(s_{t+1})) = \frac{1}{2}||\hat{\phi}(s_{t+1}) - \phi(s_{t+1})||_2^2 \tag{2.30}$$

The overall optimization problem used to train both the agent and the ICM is a composition of the loss functions L_I and L_F, together with the maximization of the agent's policy expected sum of rewards:

$$\min_{\theta_P, \theta_I, \theta_F} \left[-\lambda E_{\pi(s_t;\theta_P)}[\sum_t r_t] + (1-\beta)L_I + \beta L_F \right] \tag{2.31}$$

where θ_P are the weights of the agent's policy, $\lambda > 0$ is a scalar that weighs the importance of the policy gradient loss against the importance of learning the intrinsic reward signal, and $0 \leq \beta \leq 1$ is a scalar that weighs the inverse model loss against the forward model loss.

Then, the intrinsic reward signal r_t^i is computed using the forward model as:

$$r_t^i = \frac{\eta}{2}||\hat{\phi}(s_{t+1}) - \phi(s_{t+1})||_2^2 \tag{2.32}$$

where $\eta > 0$ is a scaling factor.

Finally, the actual reward used to train the agent's policy is obtained by summing together the extrinsic and the intrinsic rewards:

$$r_t = r_t^e + r_t^i \tag{2.33}$$

where r_t^e denotes the extrinsic reward produced by the environment at time t.

2.3.5 Random Network Distillation

The method introduced in 2018 by Burda *et al.* in [8] is a simple but very effective solution for generating intrinsic rewards which can be placed somewhere between count-based and prediction-based methods. This approach is based on the introduction of two additional neural networks with the same structure, which will be used to define a prediction problem about the novelty of the visited states. The first network is called *target network*, and it is randomly initialized before starting the training to map states to vectors of random numbers. The target network is used to set the prediction problem that the *predictor network* will try to solve, by training on the data collected by the agent. Each time a new state is visited in a transition, first, the target network is used on the observation to obtain the random vector associated. Then the predictor network is applied to the same observation trying to predict the same output vector produced by the target. During the training of the agent, the predictor is trained by gradient descent to minimize the MSE loss between the two vectors, but at the same time, this error value is used as the intrinsic reward for the agent. By referring to f as the function computed by the target network, and $\hat{f}$ the function computed by the predictor network with weights θ, we have:

$$\begin{aligned} f &: S \rightarrow \mathbb{R}^k \\ \hat{f} &: S \rightarrow \mathbb{R}^k \\ r_t^i &= ||\hat{f}(s_{t+1}; \theta) - f(s_{t+1})||_2^2 \end{aligned} \tag{2.34}$$

The main idea behind this technique is that the prediction error is expected to be lower for states similar to the ones seen often by the agent during training. In fact, the predictor will be trained more on those observations. On the other hand, the error will be higher for those rarely seen states on which the predictor has trained less, which might be more interesting.

Despite its simplicity, this method was able to achieve excellent performance on hard-exploration games like Montezuma's Revenge. A crucial role in obtaining these good results was played by the pre-processing of the states before giving them as input to the networks, and by the normalization of the generated intrinsic reward values. The observations are normalized by whitening each dimension by subtracting the running mean and then dividing by the running standard deviation. The normalized observations are then clipped in $[-5, 5]$. The normalization parameters are initialized before starting the training by stepping a random agent in the environment for a small number of steps. This normalization scheme is used for both target and predictor networks, but not for the agent's policy network. Regarding the normalization of the intrinsic reward values, these are divided by a running estimate of the standard deviations of the intrinsic returns. This allows the rewards to be on a consistent scale across different environments and points in time during the training of the agent.

Chapter 3

Curiosity as Priority

3.1 Intrinsic Motivation for Efficient Learning

As seen in the previous chapter, some of the most popular methods for incorporating intrinsic motivation into the training of the agents are based on a modification of the classical reward that the environment provides. All these techniques define specific modules that are used to produce intrinsic reward values on the basis of the training history and of the last state just explored. These intrinsic reward values are then combined with the extrinsic reward to obtain the feedback used to guide the training of the agent. In the three approaches presented in the previous chapter, this combination is always performed by simply adding together the two kinds of rewards.

The aim of this thesis is to experiment a new way of using this intrinsic feedback that no longer relies on changing the reward provided to the agent. Instead, these same intrinsic values will be used to direct the agent's training on those past experiences which it considers more *interesting*. The main idea of this approach is to use intrinsic reward modules to evaluate how much the transitions collected by the agent are useful from an intrinsic motivation perspective. Then, in constructing the updates used to train the policy, higher priority will be given to those past experiences that received higher scores.

Conceptually, this method will train the agent more on those experiences considered useful by the intrinsic motivation modules, which could lead to an increase in sample efficiency during learning.

3.2 Curiosity as Priority

The approach outlined above finds a natural implementation when combined with the use of a prioritized experience replay. As we have studied in chapter 2, the use of a PER influences the transitions that are selected to construct the policy updates during the agent training. In fact, when we sample a random batch of transitions from a PER, the probability of extracting each transition is proportional to its priority. These priorities in the classic PER are defined as the temporal difference errors computed during the last update that used those transitions. In the context of our experiments, we will use a modified version of PER that uses intrinsic motivation values as priorities instead of classical TDs. In this way, transitions will be sampled with a probability proportional to their intrinsic motivation scores, resulting in policy updates more focused on the most “curious” past experiences.

3.3 Learning Framework

Our learning framework consists of four major components that interact with each other: the agent, the environment, the prioritized experience replay, and the intrinsic motivation module.

For this research, a Double Deep Q-Network has been chosen for the agent implementation. This model is naturally inclined to be used with a PER, and its not-so-complex architecture should make the comparison between the various configurations tested more transparent.

In figure 3.1 we can see a simplified schematic of how the framework works. The interactions between the agent and the environment produce the training transitions, which are stored in a prioritized experience replay with

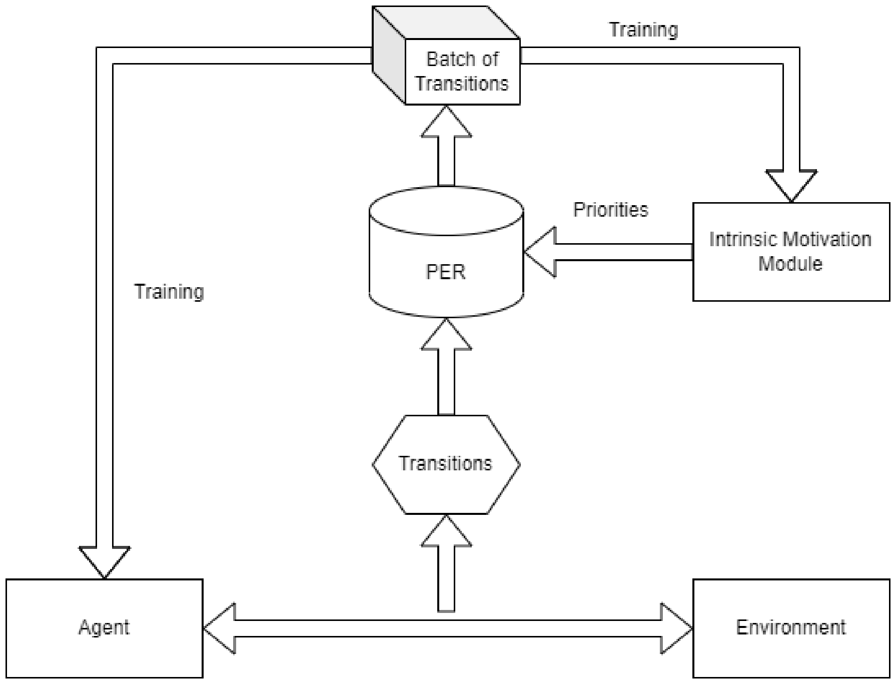

Figure 3.1: Interactions between the different components in the learning framework. The interactions between the agent and the environment produce collections of transitions that are stored in the PER buffer. Past collected experiences are sampled from the buffer and used to train both the policy and the intrinsic motivation module. Each time a batch of transitions is sampled, the current intrinsic motivation module is used to update those transitions' priorities.

an initial maximum priority level.

At regular intervals, a batch of transitions is sampled from the PER buffer and used to construct a policy update. The same transitions are also used to train the components of the intrinsic motivation module. Each time a transition is sampled from the buffer, its priority is updated using the intrinsic motivation module, and the training proceeds.

Due to the modularity of the framework, it is possible to easily test different modules for the generation of the priority values. In this thesis three intrinsic motivation techniques are implemented and tested in order to be used as intrinsic motivation modules. These are the three methods explained in the sections 2.3.3, 2.3.4 and 2.3.5.

The intrinsic reward modules are implemented following the same structure they have in their original papers, with small adaptations to our use case.

In the following chapter we will discuss the implementation details of the agent, the PER, and the three intrinsic curiosity modules which will be compared in the experiments together with two baseline models.

Chapter 4

Implementation

4.1 Double Deep Q-Network Agent

The agent used in the experiments is based on the Double Deep Q-Network model studied in section 2.2.3. Specifically, the algorithm uses a pair of two identical deep networks to compute the action values $\hat{q}(s, a)$, by following the approach described in [31]. Both networks present a final dueling layer as described in [32], using a mean over the advantages $A_\pi(s, a)$ in the combination of the output streams. The agent's policy is then implemented using an ϵ-greedy search with an ϵ value that decreases linearly over training.

Each deep neural network first processes the input through two dense layers with 128 units and rectified linear units ReLU as activation functions. Then each network presents a final dueling layer with linear activations and mean combination as described before.

The agent is trained with Adam [16] optimizer, using a learning rate equal to $1e-3$ and a discount rate γ equal to 0.99. The loss function used to train the model is the Huber loss. The value of ϵ used in the search starts at 1 and linearly decreases to $1e-2$ over the first 10% of training steps, after which it remains constant. In the double DQN architecture, the weights of the primary network are copied into the target network every 500 gradient updates of the former.

4.2 Prioritized Experience Replay

The implementation of prioritized experience replay follows the methods described in the original paper [26]. In particular, its variant with proportional prioritization is chosen.

Following the paper, the PER is implemented through the use of a data structure called “sum-tree”, which conceptually resembles a binary heap but with a different property that determines its structure: in a sum-tree, the value of a parent node is the sum of its children. Leaf nodes store the values and the internal nodes are intermediate sums, with the root node containing the total sum of all the values stored in the tree. The sum-tree is the primary data structure that constitutes the buffer, which is responsible for keeping in memory the indices and priorities of the transitions collected by the agent. The replay buffer is in fact the union of a sum-tree and a classic array in which the transitions, identified by the indices kept in the tree, are stored. The sum-tree provides an efficient way of calculating the cumulative sum of priorities, allowing $O(\log(n))$ updates and sampling [26].

To sample a mini-batch of k transitions, the range $[0; p_{total}]$ is divided into k equal ranges. Then, a value is uniformly sampled from each range. For each of these values, the sum-tree is traversed until reaching a leaf node determined by its priority value. The transitions that correspond to each of these sampled values are retrieved from the tree and will constitute the mini-batch.

The PER buffer used for our experiments has a capacity equal to $1/10$ of the number of steps for which the agent is trained. An update is performed every 4 transitions entering the buffer, using mini-batches of 64 transitions. The value of α to smooth the priorities of the transitions is equal to 0.6, while the initial value of β to regulate their importance starts at 0.4 and then increases linearly during training until reaching the value of 1 at the end of it.

4.3 Prioritizers

4.3.1 Count After Hashing

In [30], the authors explain how poor results are obtained when the hashing algorithm is applied directly to visual observations, as in the case of Atari games. For this reason, the authors introduced the use of an autoencoder for state preprocessing.

In the context of this thesis, the models will be tested on a nonvisual environment, which has low-dimensionality observations. It was therefore chosen to opt for the architecture that does not make use of AE, and thus no additional neural network is used.

The hash table is implemented with a Python dictionary, using string keys obtained from the output of the hash function $\phi(s)$, as explained in section 2.3.3.

The length k for the hash codes is set to 32, so the resulting A matrix is $32 \times n$, where n is the dimensionality of an environment's observation. Finally, the β value used in the generation of the intrinsic scores is equal to 1.

4.3.2 Intrinsic Curiosity Module

The module consists of three neural networks: the first two are used to represent the embedding model and the inverse model, while the last one is the forward model used to generate the intrinsic motivation scores.

The embedding network maps a state s to an embedding $\phi(s)$ with 32 dimensions. When used to evaluate a transition, the network is applied to both states s_t and s_{t+1}, in order to obtain the associated embeddings. The concatenated pair $(\phi(s_t), \phi(s_{t+1}))$ is then passed to the inverse dynamics network, which produces a soft-max distribution over all possible actions. The loss function L_I for the inverse dynamics system is computed as a categorical cross-entropy between the real and the predicted actions. Lastly, the forward network maps the pair $(\phi(s_t), a_t)$ to the predicted encoding $\hat{\phi}(s_{t+1})$ of s_{t+1}. The loss function L_F for the forward model is implemented as a mean squared error between $\phi(s_{t+1})$ and $\hat{\phi}(s_{t+1})$. The three networks are trained together by minimizing the following loss:

$$L = (1 - \beta)L_I + \beta L_F \tag{4.1}$$

where β is equal to 0.2.

The three neural networks present a similar structure with a single hidden dense layer comprising 128 units and ReLU activations. This architecture was chosen so that the composition of the embedding model and the forward model would result in having a similar structure to the policy network used by the agent. Both embedding and forward networks end with a dense layer with linear activations. The inverse dynamics network ends with a dense layer with soft-max activations. The networks are trained using Adam [16] optimizer and a learning rate equal to $1e - 3$.

4.3.3 Random Network Distillation

The module consists of a target and a predictor network with the same neural structure, together with a normalization procedure for the environment observations.

The states are preprocessed by subtracting the running mean and then dividing by the running standard deviation. These running estimates are computed from the states used to train the prioritizer. The normalized observations are then clipped between -5 and 5. The normalization parameters are initialized by means of trajectories collected using a random agent in the environment for a small number of steps before the training starts. The normalization is applied to the input of both target and predictor networks, but not for the agent's policy network. This follows the same preprocessing scheme proposed in [8].

Both the target and the predictor networks process a normalized input state s by means of two consecutive dense layers with 128 units and ReLU activations. Then both networks end with a dense layer with linear activations and a number of units equal to the number of actions available in the training environment.

The predictor network is trained using Adam [16] optimizer and a learning rate equal to $1e-3$. The minimized loss function is the same mean squared error used to generate the intrinsic motivation scores (see Section 2.3.5). In order to allow for a better comparison with the priority values generated by the other models, the intrinsic motivation scores were not divided by the running estimate of their standard deviation as done in the original paper.

4.4 Other Implementation Details

In all the experiments an update of the agent's policy is performed every 4 steps in the environment, thus for every 4 transitions entering the buffer. The components of the intrinsic motivation module are trained every 3 agents' updates, so every 12 steps in the environment. Each training step that involves both the agent and the intrinsic motivation module is performed using the same batch of experiences for both models.

All models are trained for a number of time steps equal to 399'000 plus 1000 initial steps used to insert the first transitions into the buffer and initialize the observation normalization parameters of the Random Network Distillation model.

Before using the scores produced by the intrinsic motivation modules as priority values, these are first clipped to 1 for stability reasons.

Chapter 5

Experiments and Results

5.1 Baselines

The following experiments aim to compare five different models to test the effect of curiosity-based transitions prioritization during the training of a reinforcement learning agent. The tested agent will be the same in all the experiments, based on a Double Deep Q-Network (DDQN) architecture as explained in the previous chapter. The models will differ only in the prioritization system used in their experience replay buffer.

The first three models use an intrinsic motivation module for the generations of priorities. The first one (**AESH**) is based on the count-based approach introduced in Section 4.3.1. The second one (**ICM**) is based on the prediction-based module presented in Section 4.3.2. Finally, the third one (**DIST**) is based on the hybrid approach seen in Section 4.3.3. Each model presents a DDQN agent which makes use of a prioritized experience replay [26] (see Sections 4.1 and 4.2) whose priorities are generated by the respective intrinsic motivation module.

The last two models represent the baselines for our experiments: the first is a DDQN agent with the original TD-based PER (**TD**), while the second is a DDQN agent with a classic experience replay with uniform probabilities [19] (**UNIFORM**). We remark that the DDQN agents used by the baselines have the same architectures as those used by the three experimental models.

5.2 Testing Environment

In order to carry out the tests of the three experimental models together with the two baselines, the research was directed toward an environment that presents a sparse reward. At the same time, it was necessary that such environment could be addressed by a Double Deep Q-network agent in a reasonable amount of training steps.

Under these premises, the MountainCar environment [21] provided by the OpenAI Gym library [7] is chosen as the benchmark for the five models.

MountainCar has already been introduced in the second chapter of this thesis, and as we have studied, it requires the agent to engage in pushing a car up a two-dimensional valley with the intent of reaching the top of the hill. The problem exists in two variants, one with a discrete action space, and the other with continuous actions. We selected the first setting, which was considered more appropriate for our agent architecture.

The discrete MountainCar environment allows the following action space:

MountainCar Action Space	
Num	**Effect**
0	Accelerate to the left
1	Don't accelerate
2	Accelerate to the right

The environment states are represented as a 2-dimensional vector of floating numbers as follows:

Observation Space	
Dimension	**Meaning**
0	Position of the car along the x-axis
1	Velocity of the car

The reward is sparse and it is equal to -1 at each time step, so the agent is encouraged to reach the top of the hill as fast as possible. The episode terminates when the time runs out, or when the car reaches the top of the hill, that is, when the position of the car is greater than or equal to 0.5. Each episode has a maximum duration of 200 time steps, which means the worst possible cumulative reward obtainable in an episode is equal to -200.

5.3 Evaluation Metrics

The models are evaluated by considering the total amount of reward collected during a full episode. Since one of the goals of this analysis is to measure the sample efficiency of the different replay buffers, the evaluations are performed by comparing models that have been trained with the same number of gradient updates. In particular, each model is trained for 399'000 steps plus 1'000 pre-train steps to fill the buffer and initialize the DIST observation normalization parameters. A policy update occurs every 4 steps, for a total number of ~100'000 updates. Every 4'000 training steps, the training process is paused and the model is evaluated by counting the total reward collected in a full evaluation episode. This ensures a fair comparison, since at each time all the models have undergone the same number of training updates. During evaluation episodes, the agent makes moves that always follow its policy, without using the ϵ-greedy strategy.

Together with the reward plots, we also perform an analysis of the priority values generated by the intrinsic motivation modules during training, to see how they impact the reward curves.

All the presented graphs are the result of averaging over 10 different complete model training sessions, for which different random seeds were used to ensure the statistical significance and the reproducibility of the experiments [10]. Along with the mean values, confidence intervals with 95% confidence level will also be shown.

5.4 Analysis of Priority Values Over Training

We will devote the next five subsections to analyze the performance of each of the five models. In addition to the reward plots, we will also consider the curves of the priorities generated by the different prioritizers during the training of the agents. Finally, we will compare all the models against each other (subsection 5.5).

5.4.1 AESH

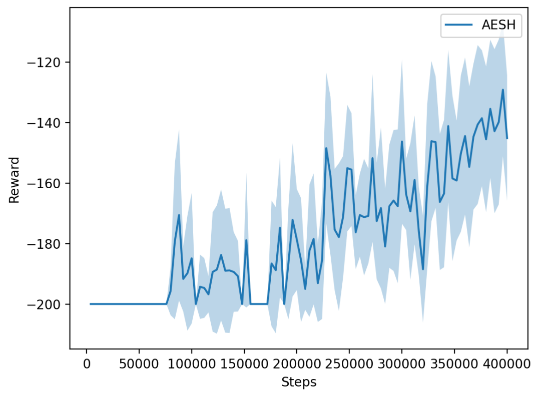

Figure 5.1: AESH Reward Plot
Cumulative rewards obtained in evaluation episodes every 4000 training steps, with mean and 95% confidence intervals across 10 different random seeds.

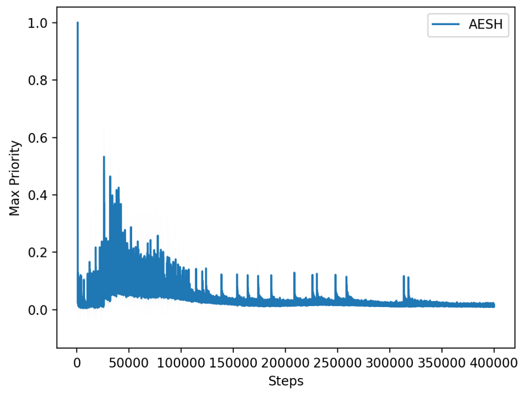

Figure 5.2: AESH Priorities Results
Max priority computed per-batch, with mean and 95% confidence intervals across 10 different random seeds.

We will start with the AESH model, i.e the one with the count-based prioritizer. Figure 5.1 gives us a measure of the increase in the model performance during training, by visualizing the cumulative rewards that the agent obtained in the evaluation episodes. Figure 5.2 aims at studying the max priority values generated per-batch over training by the intrinsic motivation module. Every time a batch of transitions is sampled from the PER and used to construct an update for the agent's policy, the priorities of those transitions are re-evaluated by the prioritizer. The priorities plot shows the maximum priority value of each batch at every agent's update.

As we can see, the generated priorities seem more relevant in the first half of the training process compared to the second. In the very first steps we can observe very high priority values due to the obvious novelty of the initial states. In the first 30% of training, we observe the maximum variance across priorities: here the model begins to explore the states of the environment for the first time, generating rather high intrinsic motivation scores. Instead, in the next steps we can see a kind of up-and-down pattern in the priority values.

One possible explanation for this phenomenon is that in the second half of training the model becomes capable of reaching the final states of the environment consistently. At the same time, however, the latter remains rarer than the initial states. This leads to a right imbalance between the counts of the more common intial states versus the rarer and more interesting final states. The priorities of the earlier states will tend to decline more rapidly than those of later ones, which, in proportion, are observed less often.

5.4.2 ICM

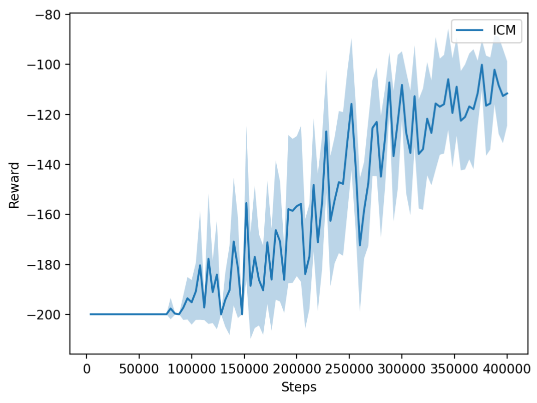

Figure 5.3: ICM Reward Plot
Cumulative rewards obtained in evaluation episodes every 4000 training steps, with mean and 95% confidence intervals across 10 different random seeds.

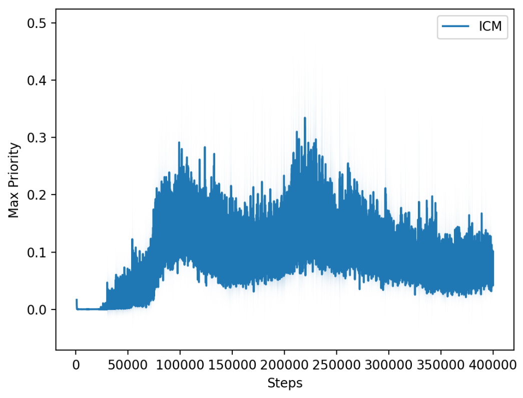

Figure 5.4: ICM Priorities Results
Max priority computed per-batch, with mean and 95% confidence intervals across 10 different random seeds.

Figure 5.3 and 5.4 show us, respectively, the reward scores obtained by the ICM model during its evaluation episodes and the maximum priorities generated during its training.

This time we can see a gradual increase in priority values that extends to about the first 20% of the training steps. Because of the ϵ-greedy exploration strategy of the DDQN agent, the latter will not be able to reach the final states of the environment from the very beginning. In these early steps, the ICM prioritizer soon becomes good at predicting the outcome of the agent's actions, since the agent is not yet able to deviate much from its initial state, because of its mostly random actions. This leads to very low errors in the predictions and therefore low priorities.

After the initial random exploration, the agent starts to reach the final states of the environment for the first time. In figure 5.3 this is visible only starting from around 25% of training, but it is referred to the evaluation episodes where the agent's behavior is more rigid in following its policy. Within the training episodes, it takes the agent less time to encounter the end states for the first time, as it explores more.

As we can see, the first encounters with final states lead to an increase in priority values. In fact, the appearance of such observations will increase the variability of states in the next-state-prediction problem faced by ICM, increasing the intensity of errors and thus priorities.

Compared to the count-based prioritizer, this time the trend of priorities seems to be more variable, with multiple times when priorities tend to rise and then fall. Probably this phenomenon stems from the changes undergone by the agent's policy, which changing during training leads to an alteration of the distribution of states observed by the prioritizer. In turn, this may result in a change in the encodings of the states obtained from the embedding network, requiring the forward network some time to readjust.

5.4.3 DIST

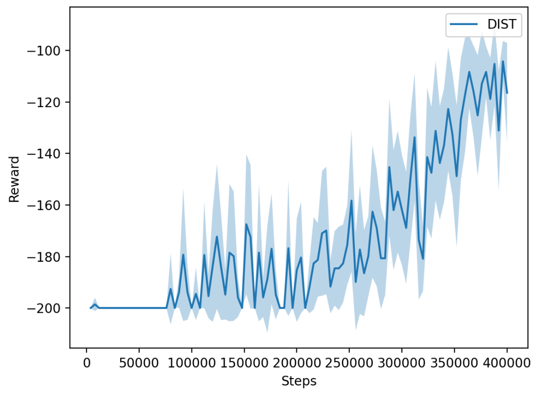

Figure 5.5: DIST Reward Plot
Cumulative rewards obtained in evaluation episodes every 4000 training steps, with mean and 95% confidence intervals across 10 different random seeds.

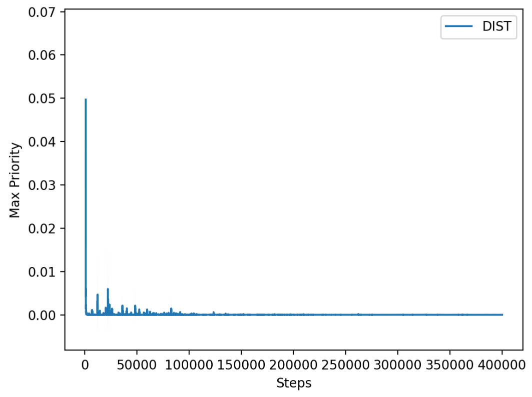

Figure 5.6: DIST Priorities Results
Max priority computed per-batch, with mean and 95% confidence intervals across 10 different random seeds.

Figure 5.5 and 5.6 show us, respectively, the reward scores obtained by the DIST model during its evaluation episodes and the maximum priorities generated during its training.

Unfortunately, The prioritizer based on the random network distillation has the least impact on the training of the agent. After an initial (small) spike in the priorities due to the novelty of the initial states, the predictor network soon becomes too good at predicting the output of the target, causing all the intrinsic values to decrease to values very close to 0.

Despite attempts to change the structure of the networks, the problem remained. Probably the low dimensionality of the observations made the application of this intrinsic motivation technique less effective.

Nevertheless, it is interesting to note that around the 50’000 step, there are some small peaks in the priorities generated. These are probably due to the first appearances of those states near the goal, which make the prediction problem more difficult. This increase in the priorities is caused by the errors made by the predictor network in predicting the target output. However, after a short time (i.e. once it learns the target outputs), this error vanishes, and the priority distribution becomes approximately uniform.

5.4.4 TD

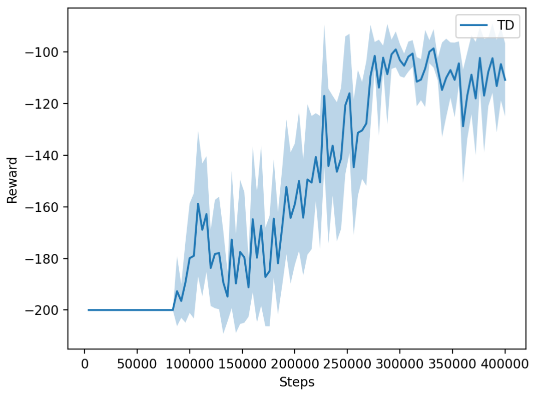

Figure 5.7: TD Reward Plot
Cumulative rewards obtained in evaluation episodes every 4000 training steps, with mean and 95% confidence intervals across 10 different random seeds.

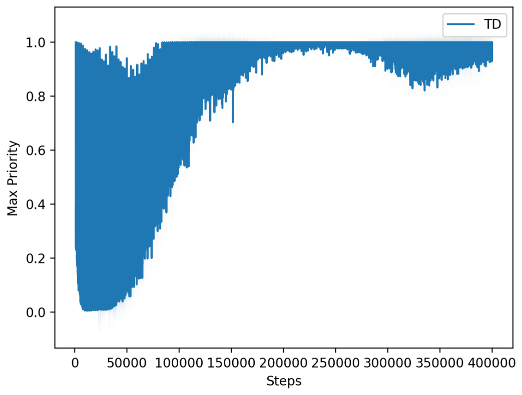

Figure 5.8: TD Priorities Results
Max priority computed per-batch, with mean and 95% confidence intervals across 10 different random seeds.

For comparison, we also report the priorities generated in the classic TD-based PER (figure 5.8), together with its evaluations (figure 5.7).

As in the original paper [26], each priority value is obtained from the TD-error of the related transition. The absolute value of TD-errors is taken, and then it is clipped to 1. Note that we are not including here the additional transformations performed by the PER to the priorities explained in section 2.1.8. Those are in fact applied in all prioritizers since they are related to the use of a prioritized replay buffer in general.

This time the priorities vary greatly in the first 20% of the training. Later, we can see how in the successive batches of transitions used to update the agent, there is usually at least a transition with a priority near 1, which means that it should be useful to improve the agent's policy.

5.5 Results Analysis

By looking at the overall picture, we can see that the prioritized experience replay with TD-based prioritization appears to be the best-performing model by a small margin, since it was the fastest to converge to a high and stable score. However, we can see that among all the experimental models, ICM performed the best, achieving a level of performance very close to that of TD.

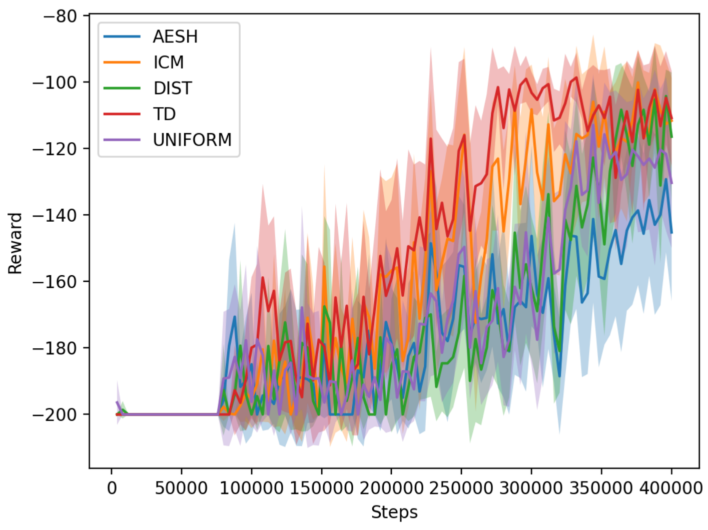

Figure 5.9: Reward plot for all models
Cumulative rewards obtained from all models in evaluation episodes every 4000 training steps, with mean and 95% confidence intervals across 10 different random seeds.

ICM not only managed to reach scores close to those obtained by TD, but it also achieved good sample efficiency. The similarity in the scores obtained by the two models could indicate that the criterion for selecting the best experiences is actually similar as well. The generation of ICM priorities is in fact based on the calculation of a prediction error which ideally should indicate how much the agent still has to learn about the dynamics of the environment. The error committed by the prediction model on a set of transitions should in fact decrease with the number of times the agent is trained on those same transitions. In the same way, TD-based prioritization favors those experiences in which the model has made the most mistakes in the past. In a certain sense, one could say that the prioritization carried out by ICM resembles a TD-based one applied to a proxy of the agent's knowledge of the environment. In TD this knowledge is contained in the agent's value function, whereas with ICM it is represented via the embedding and forward networks.

On the other hand, AESH was the worst model compared to the others. After starting out with good variability in the priority values generated, this then dropped dramatically compromising the quality of the prioritization. After the first 30% of training steps, we can observe how the number of transitions receiving high priority values drops drastically. This would not necessarily be a negative thing, since transitions with high priority might actually be the most useful for the purpose of agent training. Unfortunately, though, it seems that this strong discrimination in transitions led to worse results even compared with the model with uniform priorities. In addition, we can see how these priority peaks dropped over time, eventually leading to the generation of only near-zero priorities.

The last experimental model, DIST, was the one whose priorities had the least influence on the agent's training. In fact, the values generated were always very low, and even the smallest peaks soon disappeared, giving way to an almost uniform distribution. Precisely because of the low impact of prioritization, it is not surprising that the performance of DIST was very similar to that of the model with uniform priorities, although at the end of training its scores appeared to be slightly higher than those of UNIFORM. This approach could work better with high-dimensional observations, like in the context of visual Atari games, but it proved to be ineffective on the simplest representations of states present in MountainCar.

On a positive note, we can observe that all three models appeared capable of recognizing the appearance of later states, as visible in the priority curves.

	MountainCar Time Step	
	300k	**400k**
AESH	-146.3	-145.2
ICM	-108.2	-111.6
DIST	-162.0	-116.4
TD	**-103.2**	**-110.8**
UNIFORM	-165.1	-130.3

Table 5.1: Mean cumulative reward achieved by the models at different time steps of training. The scores are computed by an average across 10 random seeds.

The table 5.1 shows the average scores obtained by the models at time steps 300’000 and 400’000. Note how the models TD and ICM were able to achieve a good score earlier than the other models, as visible in their average rating at time step 300’000. Nevertheless, as visible in Figure 5.9, at time step 300’000 the variance in the scores obtained by ICM was much greater than that of TD, whose results were more stable. We recall how this variability can be observed because of the repetition of the tests on 10 different random seeds. In fact, each final score was obtained by averaging the performance of 10 randomly initialized models.

5.6 Performance and Time Complexity

Whenever a batch of transitions is sampled to make an update to the agent’s policy, the priorities of those transitions must be updated using the current prioritizer.

In the classical TD-based PER, this update requires only the time to change the priorities in the buffer’s sum-tree, which can be done for each transition in

$O(\log(N))$ time, where N is the number of transitions stored in the buffer. In the context of our three prioritizers, the new priority values need to be computed using a separate intrinsic motivation module, which also needs to be trained. We will now perform a very brief analysis of the most costly steps in using these prioritizers, focusing on the applications of the neural networks inside the models and their training.

In the ICM model, the priority calculation for a single transition requires two applications of the embedding network and one application of the forward network. Instead, its training requires two applications of the embedding network, one of the forward network, and one of the inverse dynamics network.

The DIST model, on the other hand, requires only one application of the target network and one of the predictor network in order to compute the priorities. The training involves only the predictor, and it requires a single application of both the predictor and the target. However, we need also to consider that this prioritizer requires normalization of observations before neural networks are applied.

The time required by the AESH model varies according to whether or not we use an autoencoder to encode states. When we use an AE, the priorities computation requires the additional encoding of the state s_{t+1} of the transition. The training of the AESH prioritizer concerns only its autoencoder when this is present, and in that case, it requires a single application of the full AE to each state s_{t+1} in the training transitions. However, our prioritizer did not include any autoencoder. Nevertheless, during our experiments, the AESH model proved to be the slowest due to the difficulty in parallelizing the hashcode search operation within the count table.

Chapter 6

Discussion and Conclusions

6.1 Contributions

In this thesis we have studied the effect of using intrinsic motivation for the prioritization of experience replay during the training of an off-policy reinforcement learning agent. Three different prioritizers have been implemented and evaluated, based on different curiosity-driven techniques [8, 22, 30], together with two baseline models. In addition, an analysis of priorities generated during model training was carried out in order to relate them to the performances of the agents.

The results of this research showed that TD-based prioritization currently remains the best approach, but we were able to investigate the main challenges and the potential of adapting some influential intrinsic reward generators in order to be effectively used for prioritization.

6.2 Limits of the Approach

Overall, there would seem to be two main aspects to consider. The first concerns an inherent problem in using intrinsic rewards as priority values. This can be observed mainly in the AESH and DIST methods, and it concerns the fact that the priority values generated, rightly, are high only for those observations that appear new to the prioritizer. The problem is that in the context of prioritization, one may have to work with transitions that are only slightly different from each other but equally important in order to construct useful policy updates. Thus, it becomes important to choose intrinsic motivation techniques capable of generating well-differentiated values over the course of training that do not drop drastically after a few steps.

The second aspect, on the other hand, is about the importance played by the normalization criterion chosen for priorities. In the experiments, the intrinsic reward values generated by the three models were used almost directly as priorities, without any special pre-elaboration. This does not seem to have been a major problem for ICM, but a more elaborate preprocessing strategy could help in models such as DIST.

Finally, from a more theoretical point of view, we must also consider how it is not obvious that an experience considered “curious” is also useful in order to improve the current estimate of the value network. This may vary also according to the category of the intrinsic motivation method adopted. For example, the notion of curiosity used by prediction-based methods might be better suited in the context of experience prioritization than the one of other families of techniques. More research would deserve to be done on the topic.

6.3 Future Work

As we have seen, not all intrinsic motivation strategies have proven to be effective for use in priority generation. It was very interesting to note that the prediction-based method performed much better than the others, perhaps indicating that the entire category of methods might be well suited for this purpose. It would therefore be useful to test other intrinsic motivation techniques, especially those belonging to unexplored categories, to see how they perform in this context. Research could be directed towards the adaptation of other types of intrinsic reward techniques, or the development of new ones specifically designed for the purpose of prioritization.

Another component of our framework that could certainly be interesting to extend in possible future work concerns the preprocessing of the intrinsic scores generated by the modules. For example, it would be useful to try new normalization strategies, perhaps using statistics calculated from the transitions currently in the buffer. More generally, it would also be interesting to try new preprocessing methodologies that are not limited to simple normalization.

Finally, it remains to be tested the effect that a combination of TD and intrinsic motivation scores would have on prioritization, with the possibility of a dynamic combination of the two that changes during training.

6.4 Final Conclusions

In conclusion, we have seen how the prioritization of transitions based on temporal difference errors remains the preferred strategy at present. At the same time, however, we have seen much potential for improvement in the intrinsic motivation approach. The ability of the prioritizers to recognize the appearance of states close to the goal and to reward the related transitions accordingly is certainly worthy of interest. Furthermore, our analysis revealed that certain categories of intrinsic motivation techniques appear to be more suitable than others for use in prioritizing experiences. Much work remains to be done, but there appear to be premisses that would justify such an effort.

Bibliography

[1] A. N. M. Alison Gopnik and P. K. Kuhl. The scientist in the crib: minds, brains, and how children learn, 1999.

[2] A. P. Badia, P. Sprechmann, A. Vitvitskyi, D. Guo, B. Piot, S. Kapturowski, O. Tieleman, M. Arjovsky, A. Pritzel, A. Bolt, and C. Blundell. Never give up: learning directed exploration strategies, 2020. DOI: `10.48550/ARXIV.2002.06038`. URL: `https://arxiv.org/abs/2002.06038`.

[3] P. Becker-Ehmck, M. Karl, J. Peters, and P. van der Smagt. Exploration via empowerment gain: combining novelty, surprise and learning progress. In *ICML 2021 Workshop on Unsupervised Reinforcement Learning*, 2021.

[4] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: an evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, June 2013. DOI: `10.1613/jair.3912`. URL: `https://doi.org/10.1613%2Fjair.3912`.

[5] M. G. Bellemare, S. Srinivasan, G. Ostrovski, T. Schaul, D. Saxton, and R. Munos. Unifying count-based exploration and intrinsic motivation, 2016. DOI: `10.48550/ARXIV.1606.01868`. URL: `https://arxiv.org/abs/1606.01868`.

[6] C. Blundell, J. Cornebise, K. Kavukcuoglu, and D. Wierstra. Weight uncertainty in neural networks, 2015. DOI: 10.48550/ARXIV.1505.05424. URL: https://arxiv.org/abs/1505.05424.

[7] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym, 2016. eprint: arXiv:1606.01540.

[8] Y. Burda, H. Edwards, A. Storkey, and O. Klimov. Exploration by random network distillation, 2018. DOI: 10.48550/ARXIV.1810.12894. URL: https://arxiv.org/abs/1810.12894.

[9] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing*, STOC '02, pages 380–388, Montreal, Quebec, Canada. Association for Computing Machinery, 2002. ISBN: 1581134959. DOI: 10.1145/509907.509965. URL: https://doi.org/10.1145/509907.509965.

[10] C. Colas, O. Sigaud, and P.-Y. Oudeyer. How many random seeds? statistical power analysis in deep reinforcement learning experiments, 2018. DOI: 10.48550/ARXIV.1806.08295. URL: https://arxiv.org/abs/1806.08295.

[11] E. L. Deci and R. M. Ryan. *Intrinsic Motivation and Self-Determination in Human Behavior*. Springer US, 1985. DOI: 10.1007/978-1-4899-2271-7. URL: https://doi.org/10.1007%2F978-1-4899-2271-7.

[12] A. Ecoffet, J. Huizinga, J. Lehman, K. O. Stanley, and J. Clune. Go-explore: a new approach for hard-exploration problems, 2019. DOI: 10.48550/ARXIV.1901.10995. URL: https://arxiv.org/abs/1901.10995.

[13] S. Fujimoto, H. van Hoof, and D. Meger. Addressing function approximation error in actor-critic methods, 2018. DOI: 10.48550/ARXIV.1802.09477. URL: https://arxiv.org/abs/1802.09477.

[14] H. Hasselt. Double q-learning. In J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, volume 23. Curran Associates, Inc., 2010. URL: https://proceedings.neurips.cc/paper/2010/file/091d584fced301b442654dd8c23b3fc9-Paper.pdf.

[15] R. Houthooft, X. Chen, Y. Duan, J. Schulman, F. De Turck, and P. Abbeel. Vime: variational information maximizing exploration, 2016. DOI: 10.48550/ARXIV.1605.09674. URL: https://arxiv.org/abs/1605.09674.

[16] D. P. Kingma and J. Ba. Adam: a method for stochastic optimization, 2014. DOI: 10.48550/ARXIV.1412.6980. URL: https://arxiv.org/abs/1412.6980.

[17] A. S. Klyubin, D. Polani, and C. L. Nehaniv. All else being equal be empowered. In M. S. Capcarrère, A. A. Freitas, P. J. Bentley, C. G. Johnson, and J. Timmis, editors, *Advances in Artificial Life*, pages 744–753, Berlin, Heidelberg. Springer Berlin Heidelberg, 2005.

[18] T. Lai and H. Robbins. Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, 6(1):4–22, 1985. ISSN: 0196-8858. DOI: https://doi.org/10.1016/0196-8858(85)90002-8. URL: https://www.sciencedirect.com/science/article/pii/0196885885900028.

[19] L.-J. Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. 8(3–4):293–321, May 1992. ISSN: 0885-6125. DOI: 10.1007/BF00992699. URL: https://doi.org/10.1007/BF00992699.

[20] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning, 2013. DOI: 10.48550/ARXIV.1312.5602. URL: https://arxiv.org/abs/1312.5602.

[21] A. W. Moore. Efficient Memory-based Learning for Robot Control. Technical report, University of Cambridge, 1990.

[22] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell. Curiosity-driven exploration by self-supervised prediction, 2017. DOI: 10.48550/ARXIV.1705.05363. URL: https://arxiv.org/abs/1705.05363.

[23] D. Pathak, D. Gandhi, and A. Gupta. Self-supervised exploration via disagreement, 2019. DOI: 10.48550/ARXIV.1906.04161. URL: https://arxiv.org/abs/1906.04161.

[24] G. A. Rummery and M. Niranjan. On-line q-learning using connectionist systems. In 1994.

[25] D. E. Ryan RM. Intrinsic and extrinsic motivations: classic definitions and new directions. *Contemp Educ Psychol*, January 2000. DOI: 10.1006/ceps.1999.1020.

[26] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay, 2015. DOI: 10.48550/ARXIV.1511.05952. URL: https://arxiv.org/abs/1511.05952.

[27] A. L. Strehl and M. L. Littman. A theoretical analysis of model-based interval estimation. In *Proceedings of the 22nd International Conference on Machine Learning*, ICML '05, pages 856–863, Bonn, Germany. Association for Computing Machinery, 2005. ISBN: 1595931805. DOI: 10.1145/1102351.1102459. URL: https://doi.org/10.1145/1102351.1102459.

[28] R. S. Sutton. Learning to predict by the methods of temporal differences. 3(1):9–44, August 1988. ISSN: 0885-6125. DOI: 10.1023/A:1022633531479. URL: https://doi.org/10.1023/A:1022633531479.

[29] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018. ISBN: 0262039249.

[30] H. Tang, R. Houthooft, D. Foote, A. Stooke, X. Chen, Y. Duan, J. Schulman, F. De Turck, and P. Abbeel. Exploration: a study of count-based exploration for deep reinforcement learning, 2016. DOI: 10.48550/ARXIV.1611.04717. URL: https://arxiv.org/abs/1611.04717.

[31] H. van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning, 2015. DOI: 10.48550/ARXIV.1509.06461. URL: https://arxiv.org/abs/1509.06461.

[32] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas. Dueling network architectures for deep reinforcement learning, 2015. DOI: 10.48550/ARXIV.1511.06581. URL: https://arxiv.org/abs/1511.06581.

[33] C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, May 1989. URL: http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf.

[34] L. Weng. Exploration strategies in deep reinforcement learning. *lilianweng.github.io*, June 2020. URL: https://lilianweng.github.io/posts/2020-06-07-exploration-drl/.

[35] R. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning, 1992.

Acknowledgements

I would like to thank Prof. Mirco Musolesi and Dott. Giorgio Franceschelli for all the help they gave me in writing this thesis. Reinforcement learning has always fascinated me, and I am very happy to have been able to do a small research in this area.

I would like to thank my family who has always been there for me and supported me.

Finally, I would like to thank all my friends who have made these years wonderful.