Alma Mater Studiorum · University of Bologna

Department of Computer Science and Engineering

Master Degree in Computer Engineering

Reinforcement learning over encrypted data

Course of
Autonomous and Adaptive Systems

Supervisors:
Chiar.mo Prof.
Mirco Musolesi

Presented by:
Alberto Jesu

Assistant supervisors:
Chiar.ma Prof.ssa
Rebecca Montanari
Alessandro Staffolani
Victor-Alexandru Darvariu

Academic Year 2019/2020

Abstract

Machine learning, in its various forms and flavours, is a very widespread technique that has been applied to the most diverse fields. It enables its users to analyze and extract information from big amounts of data, allowing them to benefit from very useful insights on the data itself. *Reinforcement learning* is a particular paradigm of machine learning that, recently, has proved times and times again to be a very effective and powerful approach.

On the other hand, cryptography usually takes the opposite direction. While machine learning aims at analyzing data, cryptography aims at maintaining its privacy by hiding such data. However, the two techniques can be jointly used to create *privacy preserving models*, able to make inferences on the data without leaking sensitive information.

Despite the numerous amount of studies performed on machine learning and cryptography, reinforcement learning in particular has never been applied to such cases before. Being able to successfully make use of reinforcement learning in an encrypted scenario would allow us to create an agent that efficiently controls a system without providing it with full knowledge of the environment it is operating in, leading the way to many possible use cases. Therefore, we have decided to apply the *reinforcement learning* paradigm to encrypted data.

In this project we have applied one of the most well-known reinforcement learning algorithms, called *Deep Q-Learning*, to simple simulated environments and studied how the encryption affects the training performance of the agent, in order to see if it is still able to learn how to behave even when the input data is no longer readable by humans.

The results of this work highlight that the agent is still able to learn with no issues whatsoever in non-secure encryptions, like AES in ECB mode. For fixed environments, it is also able to reach a suboptimal solution even in the presence of secure modes, like AES in CBC mode, showing a significant improvement with respect to a random agent; however, its ability to generalize in stochastic environments suffers greatly.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Overview

As of late, *reinforcement learning* has proved many times of being able to effectively learn and solve the most diverse types of problems, such as games. This field boasts a very active research activity and, thanks to the very promising results shown up to now, it has the potential of being adopted as a solution for many and new types of problems. Indeed, recent results such as *AlphaGo* [1, 2, 3], DQN [4, 5], Rainbow [6], and PPO [7] have proved that state-of-the-art reinforcement learning is a very powerful technique, and this has motivated many researchers to apply this approach to different fields.

Furthermore, since the introduction of homomorphic encryption, machine learning in general, and even deep learning, has been used in conjunction with encrypted data order to obtain privacy preserving inference models, such as ML Confidential [8], CryptoNets [9, 10], and CryptoDL [11]. These results prove that it is possible to build inference models in which the model itself has no visibility of the input data, which might be sensitive, and any possible leak of data cannot be exploited by a malicious third party.

Moreover, some recent work [12] has shown that supervised learning models, specifically deep convolutional neural networks, are able to easily fit a random labeling of the training data, even if such data is only noise.

Similarly to machine learning with encrypted data, being able to apply reinforcement learning to encrypted states would provide immense value. Indeed, not only it would be possible to keep the confidentiality of possibly sensitive data intact even by delegating the execution of the model to someone else, but we would also be able to leverage the power of the recent reinforcement learning techniques. Hence, we have decided to investigate this possibility so as to test if it is possible for a reinforcement learning agent to operate a system without providing it full knowledge of its environment, therefore maintaining its privacy, while also studying the effects that the encryption step causes on the agent's performance.

1.2 Contributions

Specifically, the core of this work orbits around three key research questions, that will be central in the development of the algorithm. The first one is whether it is possible that a reinforcement learning agent learns from encrypted data, which should be intrinsically noise. The second one contemplates what is the impact of the encryption step in terms of training and evaluation performance. Finally, the last one is about the overhead introduced by the encryption. Following this lead, we are going to:

- Extend the standard Markov Decision Process framework and DQN algorithm so to include the encryption of the states;
- Identify the core issues arising from the employment of encrypted states in a reinforcement learning setting, such as the choice of a fitting state transformation pipeline and of a padding technique, the definition of the cipher mode of operation, and the decision of a key length;
- Present two case studies that explore the aforementioned issues and explore their implications on the ability to learn of the agent.

The case studies are essential in that they provide a common testbed for which existing reinforcement learning algorithms are known to perform well. The introduction of an encryption step arises numerous issues, such as state processing before the encryption, padding, the decision of the encryption cipher, and so on. All of these issues are likely to impact the behaviour of the agent, which can then be compared to the well-know algorithms. OpenAI Gym [13] is one of the most widely used collections of benchmarking environments in the reinforcement learning literature, and we will leverage two such environments, specifically MiniGrid and LunarLander, in order to cover both discrete and continuous state spaces.

1.3 Structure of the Thesis

Initially, we will start by defining what reinforcement learning means. We will formulate the reinforcement learning problem starting from a simplified version, and working up to its full formulation. Furthermore, we will go over some of the most important algorithms that have been applied to this task, while also discussing some of the most recent and breakthrough results obtained by this technique, touching also on some very widespread frameworks for the development and training of reinforcement learning agents. Subsequently, we will introduce the basic concepts of cryptography, by outlining the most diffused cryptographic schemes, starting with the symmetric-key scheme, the block ciphers and their modes of operation, while also reviewing the *AES* encryption standard. Moreover, we will talk about homomorphic encryption, employed in many recent work. Lastly, we will overview some recent and interesting works that aim to obtain a privacy-preserving machine learning model, or have studied state manipulations to increase the stability and better the performance of reinforcement learning algorithms.

The third chapter, instead, aims to create the building blocks for what will be central in the later chapters and to lay out the core concepts of this work. We will provide a high-level analysis of what it means to apply reinforcement learning to encrypted data by describing a hypothetical real-world use case; then, we will define what are the key research questions and what challenges the algorithm needs to overcome for it to be successful. Afterwards, we will provide the core concepts of the algorithm proposed, by detailing a modified MDP framework, identifying the components that our agent will make use of, and outlining the key aspects of our reinforcement learning agent.

In the fourth and fifth chapters we will go over the environments that our reinforcement learning agent will be tested upon, with each of the two chapters being specific to the related case study. In both chapters we will start by describing the key features of the environment chosen for the given tests; then, we will describe the experimental settings of each investigation, by listing the parameters chosen, the architecture of our agent, and the method we observed for carrying out such experiments. Afterwards, we will overview the implementation details of the classes developed, while also outlining their scope in the context of our application and their functional interfaces. Lastly, we will analyze the experimental results observed during the execution of the evaluation of our algorithm.

Chapter 2

Background

This chapter serves as an introduction to the core aspects of the work presented in this dissertation. Firstly, we will introduce the *reinforcement learning* problem, starting a with brief history of its early days to the current state of the art; then, we will illustrate the mathematical framework of the *Markov Decision Processes* of the full reinforcement learning problem; then, the main algorithms relevant for this work will be explained, from *Monte-Carlo methods* and *temporal difference methods*, to *function approximation* and *Deep Q-Learning*.

Secondly, we will introduce the topic of *encryption*: we will start by talking about the first, and most basic, ciphers, like *Caesar's cipher*; then, we will go over the symmetric scheme, differentiating between stream and block ciphers. For the latter, we will also briefly present some of the most common modes of operation. Afterwards, the Advanced Encryption Standard, one of the most well-known and widely used symmetric encryption schemes, will be introduced. The last section of this chapter will concern another scheme named asymmetric-key scheme.

2.1 Reinforcement Learning

Learning through interaction with the environment is one of the first things that comes to mind when thinking about learning. During our lives, we interact with our surroundings constantly, and we are aware of how the environment reacts to our actions and behave accordingly.

Reinforcement learning is a computational take on the concept of learning from interaction. The key idea behind this approach is that an agent finds itself in an environment of which it knows nothing — or next to nothing — about, and tries to understand which actions are beneficial by maximising a *reward* signal. The learner is not told which actions to take, but it must discover them by observing what behaviour yields the most reward.

Alongside *supervised learning* and *unsupervised learning*, reinforcement learning is one of the main paradigms of *machine learning.* It is different from the former because in supervised learning there is an external supervisor that provides labelled examples, and the objective of the process is to generalize and extrapolate a way of correctly classifying examples never seen before. Reinforcement learning, though, is a heavily interactive problem, and in such cases it is not practical to obtain examples of desired behaviour that are both correct and representative, for the generalization process, of all the situations the agent will come across: for this reason, the agent learns not from labels but from its own experience. Moreover, it is also different from the latter: unsupervised learning means finding hidden patterns and structure among unlabelled data, while in our case the agent seeks to maximize a numerical reward signal.

Since experience is a key factor, one of the biggest challenges in reinforcement learning is how to exploit this experience. In other words, when the agents finds itself in a new situation, should it take advantage of his prior knowledge and act the way it thinks is the best, or should it explore, hoping to find something better? This is the trade-off between *exploitation* and *exploration.* Usually, to achieve the highest reward, the agent should prefer the actions that it tried in the past, knowing that they are valuable. To discover them, though, it should also try actions that it never tried before: chances are that, by only exploiting, the agent gets stuck in a local maxima, rather than in a global one. For example, if the problem is stochastic, each action should be tried many times before gaining a reliable estimate of its value. Neither of the two approaches can be sought exclusively without failing at the task; the agent should try all actions while progressively favouring those that its experience proved to be the best.

Finally, reinforcement learning explicitly considers the whole problem of goal-directed learning from interaction with an uncertain environment, whereas the other approaches split the problem into sub-problems to tackle them individually. Instead, reinforcement learning problems start with a complete goal-seeking agent, able to sense the environment and to choose actions to influence its state despite the underlying uncertainty [14].

2.1.1 Brief History of Reinforcement Learning

The early history of reinforcement learning stems from two separated threads. One thread concerned trial and error as the essence of learning, coming from the psychological analysis of learning in animals [15]. The other one involved the usage of value functions and dynamic programming for solving the optimization control problem, that laid out a mathematical framework for the modern reinforcement learning. In particular, Richard Bellman was one of the most influential figures of the subject, introducing, in the mid-1950s, the "optimal return function", also known as Bellman equation, which is a necessary condition of optimality in dynamic programming methods [16], the discrete stochastic version of the optimal control problem called *Markov Decision Processes*, or MDPs [17]. These two threads came together around the 1980s to create what is now called reinforcement learning.

For the reinforcement learning paradigm to be more thoroughly known and applied, we must wait a couple of decades, coinciding with the increase of computational power and the surge of machine learning algorithms, with deep learning in particular contributing the most.

State of the Art

Since then, many problems of diverse nature were tested and solved by reinforcement learning algorithms, sometimes even outperforming human counterparts. It is usual for artificial intelligence algorithms to be tested on controlled environments, and games are the perfect subject: they are easy to reproduce, easy to emulate, easy to play for many episodes, and are easily generalizable to real-world scenarios, but at the same time they are hard to solve. In particular, board games like chess have always drawn the attention of the AI research, and reinforcement learning is no exception.

One of the most important early works that explored this direction is *TD-Gammon*, [18, 19, 20, 21], a reinforcement learning agent that was capable of playing backgammon at the level of the most skilled human players, while requiring minimal prior knowledge of the game. Whereas the classic AI applied to games relies on heuristic searched, the complexity of backgammon made these methods completely ineffective. The learning was a combination of TD(λ) and function approximation with artificial neural networks trained by backpropagation. The training step consisted in *self-play*, a technique in which the agent plays games against itself.

A group of researcher led by Vlodymyr Mnih published, in 2013 [4] and 2015 [5], a reinforcement learning algorithm that played the games of the arcade Atari console, reaching and even surpassing, on some of them, human-level performances, while having no prior knowledge of the domain whatsoever. The algorithm took the name of *Deep Q-Learning* and remains one of the most used and well-known reinforcement learning algorithms to date. The DQN agent makes use of Convolutional Neural Networks (CNNs) to extract features directly from pixel data. Each image, after a pre-processing step, is directly fed into the CNN whose outputs are the estimated action-value functions for each action in that state. A more in-depth analysis of these two papers is given in Subsection 2.1.6.

Soon after, another research group from Google Deepmind was able to achieve breakthrough results with *AlphaGo* [1, 2, 3], a reinforcement learning agent that was able to defeat the world champion of the chinese game *Go* Lee Sedol. This accomplishment is considered ground breaking because Go has a very big state space, making an exhaustive search infeasible, and it is very difficult to design an effective position evaluation function. AlphaGo took inspiration from the two preceding works by implementing self-play from TD-Gammon and the CNN approach from DQN, merging them with Monte Carlo Tree Search (MTCS) [22, 23]. The algorithm was improved twice: the first time, with the creation of *AlphaGo Zero* [2], which did not require any domain-specific knowledge and defeated the original AlphaGo 100 to 0; and the second time with *AlphaZero* [3], that took its predecessor approach and generalized it other board games such as chess and shogi.

The results here listed are only a small part of the ones obtained in recent years. The research activity on the matter is as active as ever, motivating many researchers in studying this subject and making reinforcement learning an ever-evolving paradigm.

2.1.2 Elements of Reinforcement Learning

In a reinforcement learning system, we can identify several sub-elements: beyond the previously mentioned agent and environment, which will be expanded upon in the next section, we can also recognize a *policy*, a *reward signal*, a *value function*, and, sometimes, a *model* of the environment. [14]

The *policy* encodes the agent's way of behaving at any given time, and it alone is sufficient to determine behaviour. It can be considered as a mapping between perceived states of the environment to actions to be taken in such situations. Generally, policies can be stochastic, specifying, rather than a single action, probabilities for each action. If the agent follows a policy π at time t, then $\pi(a|s)$ is the probability that the agent takes the action $A_t = a$ given that the environment is currently in the state $S_t = s$.

The *reward signal* defines the goal of every reinforcement learning problem. At every time step, the environment sends to the agent a scalar signal in response to its actions. The objective of the agent is to maximise the total amount it receives in the long run.

The only way the agent can influence the reward signal is through its actions, either directly or indirectly by changing the state of the environment. The reward is also the primary basis for altering the policy: if an action selected by a policy yields a low reward, that action can later be changed in hope for a better outcome. In general, reward signals can be stochastic functions of the state and the action taken in such state.

The *value function* specifies what is good for the agent in the long run, whereas the reward signal gives an immediate feedback. The value of a state is the total amount of reward the agent can expect to accumulate in the future, starting from that state: it determines the long-term desirability of the state after taking into account what it is likely to follow. Formally, the value function of a state s under a certain policy π is the *expected return* when starting in s and following π thereafter. There is also an action value function, that, similarly, is defined as the expected return starting from s and taking the action a, then following π.

Finally, the *model* of the environment is something that mimics the behaviour of the environment and allows us to make inferences on how it will react to the actions of the agent. The models are typically used for *planning*, that is, deciding a course of action by considering also possible future situations before they happen: in these cases, the reinforcement learning problem is solved with a *model-based*, method, as opposed to the simpler *model-free* approaches, that follow a intrinsically trial-and-error philosophy.

2.1.3 Finite Markov Decision Processes

Finite Markov Decision Processes, or MDPs, formalize the full reinforcement learning problem, in which each action influences both the reward and the future situations (and the rewards coming from them). As such, the sequential decision making outlined by MDPs can be used as a mathematical idealisation of reinforcement learning, for which we can make very precise theoretical statements.

Markov Decision Processes define two entities: an **agent**, which is the learner and the decision-maker, and the **environment**, that is everything else outside the agent. These two entities interact continually, with the former selecting actions and the latter reacting to them by presenting new situations to the agent, as well as giving rise to a reward signal that the agent seeks to maximize.

Formally, agent and environment interact at each of a sequence of discrete time steps $t = 0, 1, 2, \ldots$. At each time step t, the agent receives a representation of the current state of the environment $S_t \in \mathcal{S}$, where $\mathcal{S}$ is the set of all the possible states, and uses this information to select a valid action $A_t \in \mathcal{A}(S_t)$, where $\mathcal{A}(S_t)$ is the set of all the actions

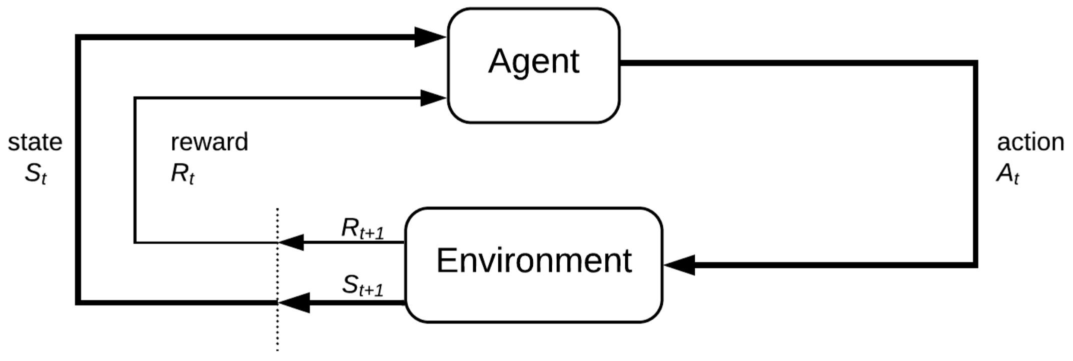


Figure 2.1: Agent-Environment interaction in a Markov Decision Process.

that can be taken in the state S_t. During the following time step, the agent receives a numerical reward $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$ and the representation of the new state $S_{t+1} \in \mathcal{S}$. The interaction between agent and environment over times gives rise to a *trajectory* of the form

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots \tag{2.1}$$

This framework can be considered an abstraction of the problem of goal-directed learning: the agent has the well-defined goal of maximising the total — that is, *cumulative* — amount of reward it receives over time; or, in other terms, the *maximisation of the expected value of the cumulative sum of a received scalar signal* called reward.

In this particular instantiation of the MDPs formulation, the sets of possible states $\mathcal{S}$, of possible actions $\mathcal{A}$, and of possible rewards $\mathcal{R}$ are all finite, from which the name *finite* MDP.

Returns

As previously mentioned, the agent's goal is to maximize the cumulative reward it receives. This amount is formalized by the *expected return.* If we can identify a clear, final time step T, then it is simply the sum of all the rewards received:

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \ldots + R_T \tag{2.2}$$

Once again, this approach, though, is only feasible when there is a clear notion of final time steps. In such cases, the interaction of the agent with the environment can be broken into sub sequences called *episodes*, each starting in a standard starting state and ending in a terminal state, followed by a reset to the initial configuration. Furthermore, these episodes are all independent from one another. This kind of task is called *episodic*.

In other cases, though, the nature of the interaction between agent and environment does not involve the concept of final time step, but it is continuous. In such *continuing* tasks, using the definition (2.2) would be problematic, because a continuing task means that the final time step $T = \infty$, and with it also G_t would tend to infinity. For this reason, the definition of G_t is slightly modified to include the concept known as *discounting*:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \quad 0 \leq \gamma \leq 1 \tag{2.3}$$

where γ is called *discounting factor*, and it idealizes the present value of future rewards: a reward received k time steps in the future is worth γ^{k-1} times of what it would be worth if received immediately. If $\gamma = 0$, the agent only focuses on immediate rewards, and does not care about future implications; as γ approaches 1, it takes future rewards more and more into consideration.

Value Functions and Policies

In many reinforcement learning algorithms, the concept of *value functions* is of central importance. Value functions are functions of states or state-action pairs that estimate how good is, for the agent, to be in a given state, or to perform a certain action from that state. The definition of these functions rely on the concept of policy, that encode a particular behaviour.

Formally, a policy π is a mapping from states to probabilities of selecting each action: if the agent is following the policy π at time step t, then $\pi(a|s)$ is the probability that $A_t = a$ given that $S_t = s$. Therefore, we can define the *state-value function* $v_\pi(s)$ of a state s under a policy π as the expected return when starting in s and following π thereafter.

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s\right], \quad \forall s \in \mathcal{S} \tag{2.4}$$

Similarly, the action-value function $q_\pi(s, a)$ of taking action a in the state s under a policy π as the expected return starting from s, choosing the action a and then following

π.

$$q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t|S_t=s, A_t=a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}\middle|S_t=s, A_t=a\right] \tag{2.5}$$

For any policy π and any state s, there is a consistency condition between the value of s and the value of its successor states, known as *Bellman equation* [16]:

$$V_\pi(s) = \sum_a \pi(a|s)\sum_{s',r} p(s',r|s,a)\big[r+\gamma v_\pi(s')\big], \quad \forall s \in \mathcal{S} \tag{2.6}$$

Solving a reinforcement learning task roughly means finding a policy that achieves a great reward over the long run. In other words, a policy π is better or equal to a policy π' if its expected return for all the states is greater or equal to that of π'. So, $\pi \geq \pi'$ if and only if $v_\pi(s) \geq v'_\pi$, $\forall s \in \mathcal{S}$. There is always at least one policy which is better than all the other ones, and we call it the *optimal* policy π_*. The corresponding *optimal* state-value function then is

$$v_*(s) \doteq \max_\pi v_\pi(s) \tag{2.7}$$

and, similarly, the optimal action-value function

$$q_*(s,a) \doteq \max_\pi q_\pi(s,a). \tag{2.8}$$

Under these circumstances, we can rewrite (2.6) as the *Bellman optimality equation*:

$$v_*(s) = \max_a \sum_{s',r} p(s',r|s,a)[r+\gamma v_*(s')] \tag{2.9}$$

for state-value function, and

$$q_*(s,a) = \sum_{s',r} p(s',r|s,a)[r+\gamma \max_{a'} q_*(s',a')] \tag{2.10}$$

for action-value functions. These versions of the Bellman equation roughly express the fact that the value of a state (or state-action pair) under a n optimal policy must equal the expected return for the best action in that state.

2.1.4 Monte Carlo Methods

Monte Carlo methods do not assume full knowledge of the environment, but instead learn from experience. The idea behind this kind of methods is to average the sample returns actually observed during the interaction with the environment. Since returns involve future rewards with respect to a certain time step t, they are only fully known if there is a final time step T. For this reason, usually Monte Carlo methods are employed for solving episodic tasks: once the episode is over, the return is known and the policy is updated.

In general, Monte Carlo methods can be broken down in three core problems:

- the *prediction* problem, that is how to compute v_π and q_π, assuming that the policy π is fixed;
- the *policy improvement* problem, which is the estimation of v_π and q_π while also improving π;
- the *control* problem to find the optimal policy π_*.

MC Control

The idea behind the control problem follows the process known as generalized policy iteration (GPI). We maintain an approximation of both the policy and the value function, and, iteratively, we alter the value function to more closely approximate the value function for the current policy, and we improve the policy with respect to the value function. Each acts as a moving target for the other, causing both to approach optimality.

For action-value functions, a greedy policy is one that deterministically chooses the action with the highest action-value:

$$\pi(s) \doteq \underset{a}{\operatorname{argmax}}\, q(s,a) \tag{2.11}$$

If, instead, we cannot use the assumption of exploring starts, we can identify two approaches: *on-policy* and *off-policy*. On-policy methods attempt to improve the policy used for deciding which action to take, whereas off-policy methods evaluate a completely different policy from the one used for making decisions.

For on-policy methods, usually we define a soft policy, meaning that $\pi(a|s) > 0\ \forall s \in \mathcal{S},\ \forall a \in \mathcal{A}(s)$. A very common on-policy method that does not rely on exploring start are ϵ-greedy policies, that grant all the non-greedy actions a minimal probability of selection equal to $\frac{\epsilon}{|\mathcal{A}(s)|}$.

2.1.5 Temporal Difference

Temporal Difference methods are a union of Monte Carlo methods and Dynamic Programming. Like MC methods, TD methods learn directly from experience without the requirement of a model of the environment; while, like DP, TD methods update their estimates based in part on other learned estimates. This process is called *bootstrapping*, and has the advantage over MC methods that they do not need to wait for the episode to finish before performing an update.

Q-Learning: Off-policy TD Control

One of the most well-known and widespread reinforcement learning algorithms is Q-Learning, designed by Christopher Watkins in 1989 [24]. Q-Learning learns an action-value function that directly approximates the optimal value function q_*. For convergence, it is only required that all pairs are continuously updated with the following rule:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big] \tag{2.12}$$

where the step size α, that controls the present weight of past rewards, is constant, and the update target is $R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$. It is called an *off-policy* method because, while the action value that will be updated is determined by the policy, the target for said update is taken as the maximum of the action values in the next state, thus not necessarily following the current policy. A pseudo-code algorithm for Q-Learning is illustrated in Algorithm 1.

Algorithm 1: Q-learning

```
Parameters:
    step size $\alpha \in (0, 1]$, small $\epsilon > 0$
Initialize:
    $Q(s, a)\ \forall s \in \mathcal{S}^+,\ \forall a \in \mathcal{A}(s)$ arbitrarily, except $Q(terminal, \cdot) = 0$
foreach episode do
    Initialize $\mathcal{S}$
    foreach step of episode until $S$ is terminal do
        Choose A from S using policy derived by Q (e.g. $\epsilon$-greedy)
        Take action A and observe R, S'
        $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$
        $S \leftarrow S'$
    end
end
```

2.1.6 Value Function Approximation

All the methods described so far have been demonstrated to converge under the finite MDP assumption. They rely on the storing of the values in a lookup table, consisting in one row for each possible state and one column for each possible action. For this reason, they are also known as *tabular* methods. This approach, though, quickly becomes intractable as the number of state increases. Consequently *value-function approximation* methods have been developed.

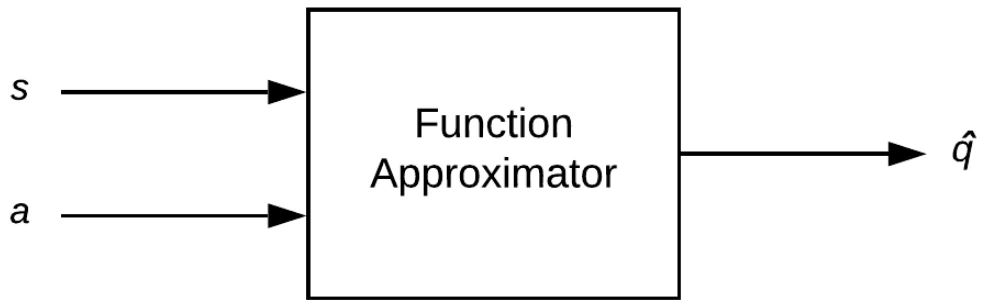


Figure 2.2: Value function approximators act like a black box to approximate the value functions. In this image, the action-value function $q(s, a)$ is approximated by $\hat{q}$.

Value-function approximation solves the problem of tabular methods by approximating the value function by using a parametrized functional form with weight vector $w \in \mathbb{R}^d$ instead of a matrix. For example, the approximated state-value function, given a state s and a weight vector w, would be defined as $\hat{v}(s, w) \approx v_\pi(s)$. The approximation $\hat{v}$ can be any function of the weights, both linear and non-linear; in general, though, a non-linear approximation is preferred. The function approximator takes samples from the desired function (in the example above, the state-value function) and, by updating the weight vector, attempts to *generalize* them to approximate the real function. A widely used technique is to use a multi-layer artificial neural network (ANN) to compute the value function.

Convolutional Neural Networks

Among the ANNs, one category in particular has obtained many good results when applied as function approximators. Convolutional Neural Networks (CNNs) are a subclass of artificial neural networks (ANNs) used for processing data arranged in a grid-like shape, with images as the most prominent usage; because of this, they are largely applied to computer vision and image recognition tasks. In general, CNNs are ANNs in which one or more layers consist of convolution operations [25]. Such operation, in a discrete domain, is defined as

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{2.13}$$

Usually, CNNs perform convolutions (or, sometimes, cross-correlations[1]) in which a bidimensional array, called *kernel* or *filter*, is slid across the image according to a certain *stride*, that determines the amount of positions each shift should skip. The size of the filter is smaller than that of the image, and the weights of the kernel are used to compute the convolution operation at each position. These weights represent the parameters of the layer, that are learned in order to detect features in the image. The output of this process is called *feature map* and it is usually passed as input to a successive convolutional layer or flattened into a one-dimensional tensor and passed to a fully connected layer.

In these situations, CNNs are better suited than fully connected ANNs because of two reasons:

- *sparse connectivity*: since the kernel is smaller than the image, we can detect meaningful features such as edges while storing fewer parameters, reducing memory requirements and improving its statistical efficiency.
- *parameter sharing*: instead of learning a different set of parameters for each position, we only learn one set and use it for the whole image. This means that while striding the kernel across the image, the kernel itself does not change, therefore reducing the storage requirements. Furthermore, parameter sharing allows the layer to be *equivariant* to translation, meaning that if the input changes the output changes in the same way.

Deep Q-Learning

One of the most important results in recent years in the field of reinforcement learning comes from the development of the *Deep Q-Learning* algorithm [4, 5] based on a multi-layer artificial neural network. This agent was able to master all 49 games of the Atari

[1]The cross-correlation function is defined in the same way as the convolution, but the kernel K is not flipped, hence the righthand side of the equation features sums instead of subtractions. This, however, as opposed to the convolution, makes the cross-correlation not commutative.

2600 console, simulated through the Arcade Learning Environment (ALE) [26], without any domain-specific knowledge.

The choice of using deep learning as the main engine behind the algorithm is justified by the impressive results obtained using deep neural networks in the field of computer vision [27, 28] and speech analysis [29, 30]. The key factor of the success of ANNs in these fields is that by making a neural network learn the representation of the data it is possible to obtain better results than handcrafted features. In particular, they used CNN to extract the features directly from the raw pixel data coming from the ALE and to output the estimated value function of each action.

A predecessor of the DQN algorithm is Neural Fitted Q-learning (NFQ) [31], that made use of RPROP [32] to update the neural network parameters. The batch update NFQ was based on, however, is very costly, proportional to the size of the dataset, while DQN applies stochastic gradient descent updates that are constant in time, thus better scaling. Moreover, while NFQ has been applied to visual data by scaling down the representation of the state with autoencoders, DQN applies reinforcement learning on the whole pipeline and directly on the images of the states.

Since the approximator used is a deep ANN, and the subject of the updates are no longer Q-values but the weights of the network, the Q-learning update is modified as follows:

$$w_{t+1} = w_t + \alpha\big[R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, a, w_t) - \hat{q}(S_t, A_t, w_t)\big]\nabla\hat{q}(S_t, A_t, w_t) \tag{2.14}$$

where w is the weight vector of the network, and $\hat{q}(S_t, A_t, w_t)$ is the action-value function approximated by the Q-network with weights w_t.

The *Deep Q-Network* (DQN) agent featured in these works takes as input the preprocessed images from the environment emulator. Then, three convolutional layers extracts the features of the images; lastly, two fully connected layers output a single estimated Q-value for each action. The agent also relies on a technique called *experience replay* [33], which consists in storing in a buffer known as *replay memory* the agent's experience $e = (S_t, A_t, R_{t+1}, S_{t+1})$ for each time step t. Then, when updating the network weights, a random batch of size D is drawn from the buffer and used to perform the updates. The advantages of experience replay over standard Q-Learning are two: firstly, randomly extracting experiences means that every tuple is potentially used in more than one update, increasing data efficiency; secondly, using consecutive samples would mean that the experiences are heavily correlated, thus, by randomizing the extraction, the correlation is broken and the variance of the updates decreases.

While in the first paper [4] the DQN agent was tested on seven games of the Atari suite, outperforming all the other approaches on six of them and even beating expert human players on three, in their second work [5] the authors modified the algorithm to improve its stability. In particular, this new version makes use of an additional DQN, labeled *target network* $\hat{Q}$, to compute the target $\max_a \hat{Q}(S_{t+1}, a)$ used in the backpropagation step of the other, on-line, network. More precisely, every C updates the on-line network is cloned into $\hat{Q}$, which is used to generate the targets for the successive other C updates. Therefore, the network update is now defined as:

$$w_{t+1} = w_t + \alpha\big[R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, a, w_t^-) - \hat{q}(S_t, A_t, w_t)\big]\nabla\hat{q}(S_t, A_t, w_t) \tag{2.15}$$

where $\max_a \hat{q}(S_{t+1}, a, w_t^-)$ is computed by the target network $\hat{Q}$ with the cloned weights w_t^-. Each C updates, the weights are cloned once again, so that $w_t^- = w_t$.

This greatly improves the stability of the algorithm compared to its predecessor, where an update to $Q(S_t, A_t)$ often also increases $Q(S_{t+1}, a)$ for all a, possibly causing oscillations or even divergence of the policy. Using this new version of the agent, the authors managed to master all 49 games of the Atari suite, outperforming human experts in the majority of them, while also keeping network architecture and hyperparameters consistent across all experiments.

2.1.7 OpenAI Gym

As we explained in Subsection 2.1.3, reinforcement learning can be formalized by MDPs. MDPs contemplate two entities: an agent, which selects one action at every timesteps, and an environment, which reacts to the agent's actions and provides it with rewards based on a reward function dependant on the quality of the action chosen.

When designing a RL agent, the environment should be one of the first elements that should be taken into account. A good environment should have precisely defined reward function, state space and observation space, and should provide the agent with enough information to successfully perform the training. Because of this, designing an environment from scratch not only is time consuming, but might make the results hard to read from the lack of a common benchmark. For these reasons, the environment for the experiments in this work will be chosen from the OpenAI Gym [13, 34, 35] framework, one of the most wide-spread toolkits for the development of reinforcement learning algorithms. Gym is an open-source framework that provides many pre-built environments, each with a well defined observation space, action space and reward function. All of the problems included in Gym expose a common interface to ease the development and the benchmarking of the RL agents and makes no assumptions about the structure and form of the agent, while being compatible with any of the numerical computation libraries, so as to allow the user to focus solely on the definition of the algorithm in the most general way possible and to quickly test it. In particular, Gym focuses on episodic tasks that can be formalized as partially observable MDPs (see Subsection 2.1.3).

Gym is an attempt to fix two of the core problems affecting RL research before its creation. Firstly, while in supervised learning large datasets such as ImageNet [36] were, and still are, central in driving progress, RL on the other hand lacked thereof. The open-source collections of environment at the time were poor in variety and seldom hard to use. Secondly, there were no standard environments to be used in publications, and, since even a subtle difference in the reward function or set of actions can drastically alter the outcome of a task, this makes it difficult to reliably reproduce the research and to compare the results of different papers.

In particular, benchmarking is a critical topic for what concernes reinforcement learning. The benchmarks environment are vital in enabling the research work to be accurately reproduced, thereby facilitating further advancements in the field: it is possible to reliably judge the efficacy of a method. with respect to another, only if the underlying assumptions regarding the environment are kept consistent.

Specifically, a Gym environment is created by calling `Gym.make(id)`, where `id` is the identifier string for the chosen environment. One of the most important functions exposed by the environment object is `step(action)`, which takes as input a number corresponding to a given action and performs it on the environment. This function returns four values: an `observation`, which is an environment-specific object encoding the new state of the environment (for example, pixel values from a camera, the configuration of the board in a board game, etc); a `reward`, a float number representing the reward following the action taken in the previous time step; `done`, a boolean flag that indicates the termination of an episode; and an additional `info` dictionary, containing diagnostic information useful for debugging.

Aside from `done` and `info`, we can clearly notice how the `step` function implements a classic agent-environment loop, with the quadruple $(S_t, A_t, S_{t+1}, R_{t+1})$: at each time step, the agent chooses the action for the `step` function based on the current `observation`, and the environment returns the `observation` for the following state and the associated `reward`. This process is started by calling `reset()`, a function that resets the environment to the initial state and returns the corresponding `observation`.

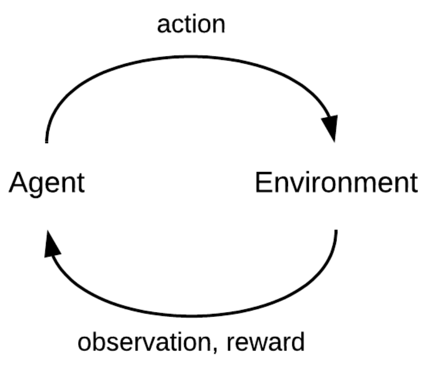


Figure 2.3: Classic agent-environment loop. The agent, based on the current state of the environment (represented by the observation object in Gym) chooses an action. The environment reacts to it by changing its state and by giving the agent a numerical reward.

MiniGrid

OpenAI Gym is easily extensible by design. One such extensions is MiniGrid [37], a collection of environments for Gym designed to be simple and lightweight, in which the agent is represented by a red triangle that moves in a discrete grid of $N \times M$ tiles with a green goal tile, and, optionally, pickable objects, doors, and obstacles.

2.2 Cryptography

Historically, cryptography was introduced as a means to maintain privacy of the information sent between two parties, even in the presence of a third party, called adversary, with access to the communication channel [38]. This type of cryptography is often referred to as classical cryptography, and its usage dates back to the ancient Egypt. Until the 20th century, cryptography did not have a solid theoretical basis, and we lacked a general definition of what a secure system would be.

This began to change with the 20th century and the proliferation of computers and communication systems, which brought a demand for a secure way to protect information in digital form and provide security services. Thus, modern cryptography was born: the emersion of a very rich theory providing scientific back up to the subject enabled cryptography to be studied as a science; furthermore, cryptography is no longer only about secrecy, but it now encompasses many other fields like authenticity, integrity, and digital signatures [39].

Classical cryptography, on the other hand, concerned exclusively the secrecy of the information, and it was mainly used by the military. One of the most well-known examples of classical cryptography usages is *Ceasar's cipher*. This scheme takes as input a sentence, and shifts each of its character by a fixed amount of positions in the alphabet. Of course, this type of encryption is not secure: we only need to perform a basic statistical analysis to understand what is the most common letter of the gibberish, then compare it with most common letter of the original language, and we would have figured out the number of positions of the shift. Furthermore, even without knowing the most common letter of the language, we could try all 26 possibilities until we find a configuration that turns the gibberish in a meaningful sentence. This type of attack is know as *brute force*.

2.2.1 Symmetric-key Setting

In the symmetric-key setting, also called private-key encryption, two parties that wish to communicate secretly share some secret information called *key*. The sender uses this key to *encrypt*, or scramble, the message before sending it, obtaining a *ciphertext*; then, the receiver, upon reception, uses the same key to *decrypt*, or unscramble, the ciphertext, acquiring the original readable message, known as *plaintext*.

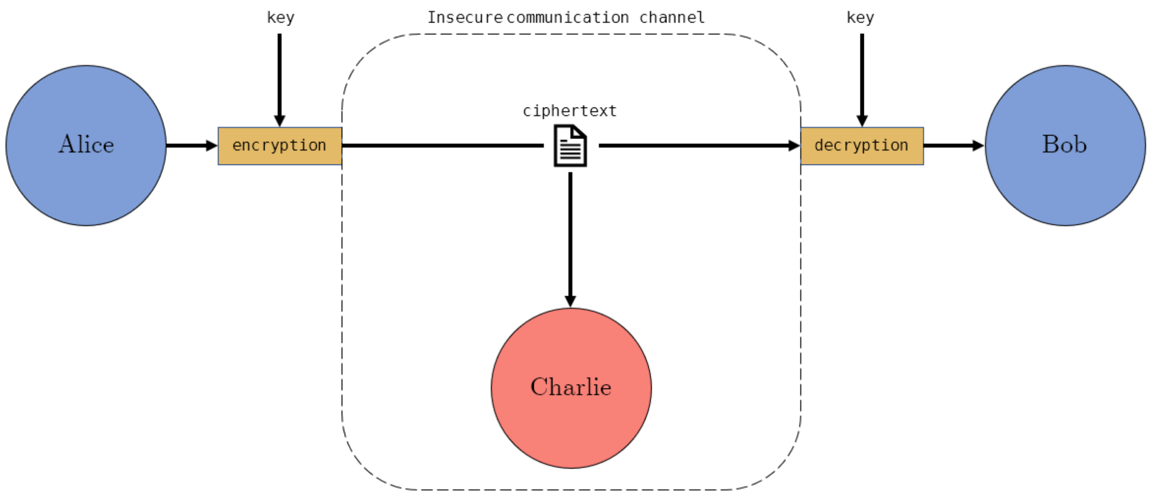


Figure 2.4: Symmetric encryption scheme. Alice encrypts a message with a key and sends it to Bob. Bob decrypts the message with the same key and obtains the original message. Even if Charlie eavesdrops the conversation and knows the algorithm used for the encryption, they cannot read the message without the key.

The symmetric encryption scheme relies on the assumption that the two parties have a secure way of deciding a common, secret key. This limits the applicability of systems that solely make use of symmetric key encryption; despite this, there are many settings in which these methods comply with the security requirements, making them broadly used. For example, in disk encryption the two parties are represented by the same user, who uses a key to read and write to disk. Also, symmetric methods are widely used in conjunction with asymmetric methods.

More formally, a symmetric-key encryption scheme, or *cipher*, is composed by three algorithms [39]:

- The *key-generation algorithm* `Gen` probabilistically outputs a key k;
- The *encryption algorithm* `Enc` takes the key k and the plaintext message m as input and outputs the ciphertext c, and we denote $\texttt{Enc}_k(m) = c$;
- the *decryption algorithm* `Dec` takes the key k and the ciphertext c as input and outputs the plaintext m, and we denote $\texttt{Dec}_k(c) = m$.

For the cipher to run correctly, we must have that

$$\texttt{Dec}_k(\texttt{Enc}_k(m)) = m, \ \forall k \in \mathcal{K}, \forall m \in \mathcal{M} \tag{2.16}$$

where $\mathcal{K}$ is the key space and $\mathcal{M}$ the message space.

Ciphers are divided into two categories, based on their modality of operation:

- stream ciphers encrypt each bit individually, by carrying out a logic operation (for example, a XOR) between the plaintext and a key stream;
- block ciphers divide the plaintext into blocks of a predetermined length and encrypt them with a key, so that the encryption of each bit of plaintext in a block depends on every other bit in the same block.

2.2.2 Modes of Operation

A mode of operation specifies how block ciphers should encrypt messages of arbitrary length [39]. Since block ciphers divide the message into fixed-length blocks, the message length should be equal to or a multiple of the block size. If it is not, the plaintext is usually *padded* with arbitrary bits (PKCS#7 [40] being the current standard) to a multiple of the block size before encryption. Therefore, from now on we will assume that the length of the plaintext is always a multiple of the block size. In the following paragraphs, some of the most well-known modes of operation will be listed.

Electronic Code Book - ECB

The ECB mode is the most naive mode of operation. Given a message $m = m_1, m_2, \dots,$ m_l, the ciphertext is obtained by encrypting each block m_i of plaintex independently, that is, $c = \langle F_k(m_1), F_k(m_2), \dots, F_k(m_l)\rangle$, where $F_k(\cdot)$ is the encryption function performed by each block with key k. The ECB encryption is deterministic, and it is not indistinguishable in presence of an adversary, even if it is only used once. This is due to the fact that if the same block is repeated several times in the plaintext, the resulting ciphertext blocks encrypted with the same key will also be the same.

Cipher Block Chaining - CBC

The CBC mode involves a random initialization vector, denoted IV, of length n. The first ciphertext block is generated by computing the encryption of the XOR between the IV and the first plaintext block m_1: $c_1 = F_k(IV \oplus m_1)$. From here, each subsequent ciphertext block is obtained by encrypting the XOR between the current plaintext and the previous ciphertext. More formally, the i^{th} ciphertext block is obtained as $c_i = F_k(c_{i-1} \oplus m_i)$, where $c_0 = IV$. At the end of the process, the encrypted message is $\langle IV, c_1, c_2, \dots, c_l\rangle$. The IV does not have to be kept secret, and it is usually sent in plaintext along with the ciphertext, so that the receiver can successfully decrypt the message.

CBC is probabilistic, but has the drawback that each ciphertext c_i is required to encrypt c_{i+1}, hence the encryption must be carried out sequentially and cannot be parallelized (however, decryption can be parallelized provided that the recipient of the message already has all of the ciphertext blocks). Moreover, two identical plaintexts encrypted with the same key and IV will result in identical ciphertexts. For this reason, it is mandatory that the same IV is not used more than once, but a new, randomized one should be generated each time.

Output Feedback - OFB

In this mode, the block cipher is used to generate a pseudorandom stream that is then XORed with the plaintext, turning a block cipher into a stream cipher. The first step is to choose a random $IV \leftarrow \{0,1\}^n$. From the IV, we generate a stream by doing $r_i = F_k(r_{i-1})$, where r_i is the i^{th} block of the stream and $r_0 = IV$. From here, the

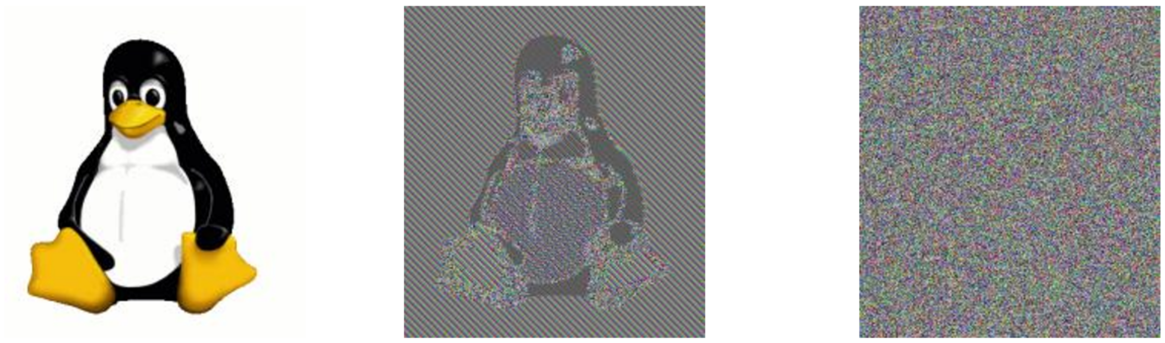


Figure 2.5: Tux penguin image in plaintext (left), encrypted in ECB (middle) and in a secure block cipher mode (right). Note how in the ECB version, while the pixels are encrypted, it is still very possible to discern the original image since the plaintext is made by many recurring patterns, which ECB preserves. Courtesy of [41].

XOR between each plaintext block and the corresponding stream block is performed: $c_i = m_i \oplus r_i$. As in the CBC mode, the IV is usually included in plaintext along with the encrypted message to allow decryption.

OFB is probabilistic. Both encryption and decryption must be carried out sequentially and cannot be parallelized. In this case, however, the pseudorandom stream can be computed beforehand, after which encrypting or decrypting the data is much faster.

Counter - CTR

Much like OFC, CTR also turns a block cipher into a stream cipher. As usual, a random $IV \leftarrow \{0,1\}^n$ is chosen and will take the name of `ctr`. The encryption step consists in $r_i = F_k(\texttt{ctr} + i)$, where the addition is performed modulo 2^n. Lastly, the encryption of each plaintext block is computed as $c_i = r_i \oplus m_i$.

Both encrytpion and decryption can be fully parallelized, and its pseudorandom stream can be computed ahead of time like in the OFB case. It is also possible to perform *random access* decryption, that is, decrypting the single i^{th} block without decrypting anything else.

2.2.3 AES - The Advanced Encryption Standard

In 1997, the National Institute of Standards and Technology (NIST) of the United States kicked off a competition to define a new cryptographic algorithm that would be called *Advanced Encryption Standard* [39]. Many teams of expert cryptographers and cryptanalysts submitted candidate algorithms, and all had interest in finding some potential flaws in the other proposals. Thus, each of the submissions was thoroughly analyzed for their security and performance. In 2000, the NIST announced *Rijndael*, named after the authors Vincent Rijmen and Joan Daemen, as the winner of the competition [42].

Intuitively, AES is essentially a substitution-permutation network. It holds a 4 by 4 vector named *state*, initialized to the input of the cipher (16 bytes), and over which the substitutions and permutations are performed. Each *round* of AES-Rijndael is divided into four stages [39]:

1. `AddRoundKey`: at each round, we obtain from the master key a 16-byte round key, reshaped as a 4 by 4 matrix of bytes. Then we compute the XOR of the round key with the state array. As such, this step consists in computing $a_{i,j} = a_{i,j} \oplus k_{i,j}$, with $i, j \in [1, 4]$.

2. `SubBytes`: using a fixed lookup table S, each byte of the state array is substituted by another byte. S is a bijection over $\{0, 1\}^8$, so that $a_{i,j} = S(a_{i,j})$, with $i, j \in [1, 4]$.

3. `ShiftRows`: each row of the state array is shifted to the left by a number of position equal to $s = i-1$, therefore the first row is left untouched, the second row is shifted by one position, the third by two, and the fourth by three. The shifts are cyclic, meaning that each byte that would end up out of the 4 by 4 boundaries is reinserted at the end of the array. For example, byte $a_{2,1}$ becomes byte $a_{2,4}$ after the shift.

4. `MixColumns`: an invertible linear transformation is applied to each column, that are interpreted as a polynomial over $GF[2^8]$, then multiplied modulo $x^4 + 1$ with the fixed polynomial $c(x) = 3x^3 + x^2 + x + 2$.

The number of rounds performed in the AES-Rijndael algorithm is dependant on the size of the key: 10 for 128-bit keys, 12 for 192-bit keys, and 14 for 256-bit keys.

Regarding its security, the only successful attacks found to date are for reduced-rounds variants of AES, and even those still have a high complexity. For the full AES construction, no attack better than a brute force search over the key space is known.

2.2.4 Homomorphic Encryption

Homomorphic encryption (HE) was first introduced by Craig Gentry in 2009 [43], as a solution to a much older open problem labelled *pivacy homomorphism* by Ronald Rivest et al. in 1978 [44], and later named *fully homomorphic* encryption scheme by Gentry himself. The core concept behind this scheme is to allow one to compute an arbitrary function over the encrypted data without needing to decrypt it. In other words, given the encryptions $E(m_1), E(m_2) \dots E(m_n)$ of the plaintext messages $m_1, m_2, \dots m_n$, one should be able to compute $f(m_1, \dots m_n)$ for any function f.

Like any other encryption scheme, HE relies on the main functions `Gen`, `Enc`, and `Dec` for the generation of the key, the encryption step and the decryption step respectively. However, in addition there is an *evaluation* function `Eval`. In general, HE can be summarized as

$$\texttt{Dec}(s_k, \texttt{Eval}(p_k, C, c_1, c_2, \dots, c_n)) = C(m_1, m_2, \dots, m_n) \tag{2.17}$$

where s_k is a secret key, p_k is the corresponding public key, $m_1, m_2, \dots, m_n$ is a set of plaintext messages, $c_1, c_2, \dots, c_n$ is a set of ciphertexts and C is a circuit.

The key idea that allows us to compute functions over the ciphertext is to make the encryption function `Enc` add a small value, called *noise*, to the plaintext m_i during the encryption step. Hence, each ciphertext c_i will have a small amount of noise associated with it and, when we compute the sum of two ciphertexts, their noise grow with them. The decryption step `Dec`, then, is only able to correctly decrypt the ciphertext as long as the noise is below a certain threshold, that creates a bound on the amount of operations that can be performed. Furthermore, the only supported operations are additions and multiplications.

Gentry [43] designed the first fully homomorphic encryption scheme, that is, a HE scheme that supports an arbitrary depth circuit, by introducing a technique used to decrease the noise associated with the ciphertexts without needing to access s_k called *bootstrapping*. However, because of the big computational cost and slow processing time, FHE is impractical for actual use.

Successive versions [45, 46, 47, 48, 49] of the scheme improved on the initial one designed by Gentry and gradually made it more and more practical. All of the FHE schemes rely on an underlying Somewhat Homomorphic Encryption (SHE) scheme, that performs the additions and the multiplications on the encrypted data, responsible for the growth of the noise factor. The bootstrapping technique is then responsible of the reduction of the noise level, extending the SHE to FHE.

Since the noise factor grows exponentially in SHE, this scheme is very limited from the point of view of how many operations are computable before bootstrapping. Indeed, SHE can only evaluate polynomial functions up to a bounded degree. However the scheme was later improved by making the noise factor growth polynomial in the number of multiplications [49, 50], instead of exponential. Such schemes are called Leveled Homomorphic

Encryption (LHE) schemes and allow the evaluation of the polynomial functions to a higher, although still bounded, degree without using the costly bootstrapping technique.

One of the most well-known implementations of homomorphic encryption is Microsoft SEAL [51], a C++ cryptographic library that includes two recent HE schemes: BFV [52] and CKKS [53].

2.3 Related Work

Machine learning models can greatly benefit their user in upgrading the service they offer, but not every owner has the economical or structural resources to build a computational hub for running them. Cloud providers are obviously a great choice, but when the data that the models should be trained or run upon is sensitive, many privacy-concerning issues are raised.

For this reason, many works that try to combine machine learning with cryptography have been published in recent years. However, to date, none of them applies reinforcement learning to such scenarios.

2.3.1 Machine Learning with Encrypted Datasets

Unfortunately, not every service that might benefit from the application of machine learning has the economical or structural resources to privately build and train the models. For this reason, many cloud providers offer solutions, general or tailored, to use a model remotely, so that clients can make use of machine learning without needing to build a computational center for it. This, alongside the clear advantages, also carries some concerns, especially about the privacy of the data that is being handed to the providers.

This example has the purpose of introducing the main issue that some recent works that aimed to combine machine learning with privacy-preserving procedures try to solve. One of the first works that explored this direction is ML Confidential [8], published in 2012, a confidential machine learning algorithms based on low-degree polynomial versions of classification algorithms. The encryption scheme was a *leveled* homomorphic encryption scheme, in which the noise growth is limited to a polynomial function, allowing to perform a bounded amount of operations without resorting to the bootstrapping technique (recall Section 2.2.4). The authors implemented a Linear Means classifier and a Fisher's Linear Discriminant (FLD) [54] on the publicly available Wisconsin Breast Cancer Dataset [55], obtaining better results in terms of efficiency compared to the other HE encryption benchmarks of the time.

There have also been applications [56] of cloud-hosted prediction services that took as input private encrypted medical data and returned, always in encrypted form, the probability of suffering of cardiovascular disease. The working proof of concept leveraged a fully homomorphic encryption scheme to allow computation on the data.

Furthermore, in 2015, R. Bost et al. [57] constructed three major classification models: hyperplane decision, Naïve Bayes, and decision trees, combined with AdaBoost. The models were applied to a FHE scheme based on a polynomial representation requiring only a small number of multiplications. Not only they obtained efficient models, but also created a library of building blocks for constructing many others classifiers in a privacy-preserving way, such as a multiplexer or a face-detection classifier.

Not long after, the first neural networks were applied to encrypted data as well. However, since homomorphic encryption only supports additions and multiplications, neural networks cannot be simply applied to encrypted data as they are, but require some modifications. One of the most challenging tasks is to approximate the activation functions, such as the Rectified Linear Unit (ReLU) or Sigmoid, with functions that only make use of additions and multiplications, like polynomials.

For example, in 2016, another Microsoft research group worked on the implementation of a project named CryptoNets [9]. Also in this case, the considered encryption scheme is HE and the assumption is that a cloud service hosts the neural network and allows data owners to send their data in encrypted form. The neural network is then applied, and the prediction is returned always in encrypted form. Homomorphic encryption, as stated in Section 2.2.4, allows a function to be computed on the encrypted data without knowledge of the secret key. Therefore, since the cloud service is not in possession of such key, it is not able to gain any information on the data nor on the predictions. The activation function was approximated with the lowest degree non-linear polynomial, the square function. CryptoNets consisted in a CNN and was applied to the MNIST [58] dataset, a common benchmark for convolutional neural networks consisting in handwritten digits to be correctly classified, after encryption. It achieved 99% accuracy and was able to hold a throughput of 51,000 predictions per hour.

However, CryptoNets quickly became ineffective for deeper ANNs. An improvement to the model [10] addressed this problem by applying batch normalization [59] to the data. This time, the ReLU units were approximated with a Taylor series. The new, deeper CryptoNet was applied to the same encrypted MNIST dataset registering accuracy similar to the best non-secure versions at the time.

Soon after, CryptoDL [11] was published, a new model for applying neural networks to encrypted data within the limitations of homomorphic encryption schemes. Firstly, the authors designed methods to use low-degree polynomials (which are essential for effinciency) for approximating the most common activation functions used in CNNs, like ReLU, Sigmoid or Tanh. Then, the training is performed with the approximated activation functions. Lastly, they implemented the neural network model and tested it on the encrypted MNIST, achieving a 99.52% accuracy (very close to the non-private state of the art version, with 99.77%), and to the CIFAR-10 dataset [60], considered to be more complex than MNIST, achieving an accuracy of 91.5%.

In the same year was published another project involving deep learning system [61], this time in conjuction with asynchronous stochastic gradient descent and additively homomorphic encryption to protect the gradients from an *honest-but-curious* cloud server, that is, a cloud server that does not actively try to break the protocol, but will try to learn as much information as possible from the legitimately received messages. The gradients are stored on the server in an encrypted form, and the additive homomorphic encryption scheme allows to carry out computations on such gradients without the decryption key. By doing this, they achieved an intact accuracy, while also leaking no information to the server.

2.3.2 State Manipulation in Reinforcement Learning

While machine learning over encrypted data has its share of studies, on the other hand reinforcement learning, at the time of this work, has never been applied to encrypted states. As stated in Section 2.1, while sharing some similarities, supervised learning and reinforcement learning are two different paradigms. Specifically, instead of classifying the input data into labels, reinforcement learning yearns to find a *policy*, a mapping from situations to actions, that maximizes a reward signal. This means that RL should probably follow another direction with respect to the works that were previously discussed.

A paper [62] was published in which count-based exploration methods[2], usually employed in conjunction with tabular methods such as Q-Learning, was to high-dimensional or continuous deep RL benchmarks. Interestingly, the authors extended the counting-based methods by discretizing the state space with a hash function, reaching near state-of-the-art performances on several environments from Atari 2600 and rllab [63], a suite for the benchmarking of continuous control tasks.

[2]With count-based exploration methods we mean methods for action selection based on the counting of the state-action visitations. One such method is the well know Upper Confidence Bound, usually applied to the bandits problem: $A_t = \text{argmax}_a \left(\hat{r}(a_t) + \sqrt{\frac{2 \log t}{n(a_t)}}\right)$

This paper is interesting to the matter at hand because it shows that a deep RL agent is able to learn and to converge to optimality even if the representation of the state greatly changes. In some way, this is similar to the task we are trying to solve: encrypted data is a representation of the data manipulated so that an adversary is not able to understand the underlying plaintext.

However, the concept of manipulating the input to the agent in some way is not novel. Data augmentation is a procedure that aims at incrementing the amount of data by manipulating the existing data, or by synthetizing new data [64]. This technique is employed especially in computer vision tasks, with operations such as geometric transformations, random erasing, feature space augmentation, random cropping or random flipping [65], that are able to improve the model's ability to generalize and, as such, lead to better classification results.

The data augmentation techniques to improve the generalization capabilities of the models were only recently investigated in the context of reinforcement learning algorithms. There have been instances in which data agumentation such as cutout [66], where multiple rectangular regions are masked and assigned a random color, or randomized convolutions [67], where random convolutional newtorks are used to output randomized feature maps over which the RL agent is trained, have proved to be effective. Two very recent works leveraged data augmentation for reinforcement learning: Data-regularized Q [68] regularizes the Q-function in conjunction with off-policy methods and randomized shifting of the input image, while RAD [69] aims to create a framework for reinforcement learning data augmentation with many data augmentation techniques, such as randomized cropping, translation, colour jitter and others.

While not directly associable to cryptographic operations, these approaches show that the modification of the input data for reinforcement learning agents not only is a viable approach, but can also improve its performance under certain circumstances. Therefore, it might be natural to ask ourselves how the cryptographic step fits in this picture, and if the noise produced by the encryption is in some way exploitable for the learning of a reinforcement learning algorithm.

2.4 Summary

This first chapter serves as an introduction to the concepts that play a key role in the understanding of our work, and are, as such, required background knowledge. We have thoroughly explained the problem of reinforcement learning, starting by its history and some state of the art works, then defining its simplified form and working our way up through MDPs, Monte Carlo and Temporal Difference methods, to value function approximation, a very powerful technique that underlies most of the recent works in the field. In the same context, we also have reviewed a very widespread framework for the development and benchmarking of reinforcement learning algorithms known as OpenAI Gym, and one of its most well-known extensions MiniGrid.

The second section, instead, deals with cryptography. We started with its basic definitions and a brief history of the disclipline; afterwards, We overviews its two main usages in the symmetric- and asymmetric-key schemes, while also defining the block cipher's modes of operations and the AES algorithm. Homomorphic encryption, that plays an important part in some recent work, has also been introduced.

Lastly, the third section details some important recent works in the field of machine learning and encryption, that constitute the research that has been done in the field most closely realated to our project.

The idea behind our work is that, despite a very active research activity on both the subject of reinforcement learning, and privacy-preserving machine learning, nothing has been published yet that tries to merge the two approaches. Indeed, a successful analysis of the concept might lead the way to some very interesting and potentially useful applications.

Therefore, this will be the purpose of this work. In the following chapter we will give an overview of the application of reinforcement learning to encrypted data and we will, first, formally formulate the problem by identifying the key research questions and the central aspects at play, and then, propose a reinforcement learning algorithm as a potential solution.

Chapter 3

Approach

As we discussed in the previous chapter, many recent works have focused their attention on the study of state manipulations in the context of reinforcement learning or on the application of cryptographic primitives to inferential machine learning applications. However, the combination of the two approaches has not yet been explored, therefore it will be the central topic of this work.

This chapter will serve as an introductory overview of our approach to the general problem of the application of reinforcement learning on encrypted data, providing the basic building blocks for the understanding of the algorithm and of the state transformations used to perform the experiments. In the first section, we will start by giving a high-level overview of the problem by defining a hypothetical real-world use case; then we will lay out the main research questions that our work aims to answer.

The second section, instead, focuses on the building of the algorithm followed during our experiments. First, we will provide a structural overview of the main components involved; then, we will build a modified MDP framework in which our agent will work; lastly, we will go more in-depth in the algorithm proposed.

3.1 Reinforcement Learning over Encrypted Data

Being able to apply reinforcement learning to encrypted states would mean creating an agent that learns a policy without supplying it with complete information about its surroundings. This would allow us to decouple the environment in which the agent moves and acts from the algorithms that control such agent, leading the way to numerous use cases similar to the ones explained in the previous papers.

3.1.1 Purpose

To better understand the aspects at play in this scenario, let us think of a hypothetical use case. For example, we might find ourselves in an industrial setting, in which there is a plant with a robot that performs some kind of operation. Such robot can be very efficiently powered by a reinforcement learning algorithm in which the states are the images of a camera that frames both the robot and the plant, but, since such algorithm requires a big amount of computational power to perform efficiently, we are not able to build it and train it ourselves. For this reason, we could turn to cloud providers, but privacy concerns prevent us from sharing the pictures of the plant, so that competitors cannot steal our designs.

This basic and hypothetical use case confirms that being able of taking advantage of reinforcement learning, that lately has proved to be a very powerful technique in heterogeneous scenarios, with encrypted data might be worthy of investigation. In general, this use case can be extended to all those scenarios in which we wish to decouple the learning algorithm from the actual environment in which the agent performs its task. As discussed in the previous section, this would allow us to create many practical implementation where, in general, the beneficiary of the agent is incapable, for any reason, of building a reinforcement learning agent autonomously. Since the states are the only descriptive information the agents needs, if those do not leak clues about the environment, we can effectively delegate the creation of the agent to someone else without providing knowledge of possibly sensitive data. Furthermore, the most interesting part is that it would also mean that it is possible to adequately train an agent that has no proper knowledge of the environment it is operating in.

3.1.2 Research Questions

Nonetheless, it might also raise some questions. Firstly, as discussed in the previous chapter, reinforcement learning is different from supervised learning, because it tries to find a *policy*, that is, a mapping between situations and actions, in order to maximize a numerical reward, instead of assigning labels to data. Having encrypted states might very well break this mapping because the encryption has the purpose of turning the plaintext into noise, so that an adversary is not able to identify the data any longer.

Secondly, reinforcement learning is an intensely interactive problem, in which the agent and the environment continuously interact with each other. Encrypting the states might introduce a significant overhead based on the type of encryption performed, as well as on how the states are represented.

Therefore, we can identify three key questions that will be central in the following pages:

- Is a reinforcement learning agent still able to learn with encrypted states?
- What is the impact of the encryption on the training performance?
- Does the encryption step introduce a significant overhead?

Providing an answer to these questions is the key concept behind the work carried out in this project, and, as such, we will try to create a reinforcement learning agent and train it over encrypted data, assessing its capabilities to learn while also measuring the impact, both in term of complexity and efficiency, of the cryptographic primitives employed for such task.

3.2 The Algorithm

With the problem just outlined, we can start to identify a possible direction in which to channel our efforts to try to create a RL agent that learns on encrypted data. Specifically, we will follow the questions mentioned in the previous section to create a logic process that will frame the problem, allowing us to build an infrastructure for the training and the testing of the agent.

As we discussed in 2.3.2, while obviously not being close to encryption, manipulations of the state in reinforcement learning have proved to be a viable approach. The encryption, from this point of view, is not very different: indeed, it consists in a transformation of the state. The difference, though, is that in the works mentioned, the transformations had the purpose of improving the agent's ability to generalize, while, in our case, we try to hide the data so that a malevolent third party cannot infer anything on it. Of course, this might prevent the agent from learning a policy from the states, because we would break the mapping between state and action. Therefore, we will develop an algorithm to test if learning still happens.

We have then developed a pipeline that will be applied in this work in order to test the agent's ability to learn. We can identify three major components, each with a very specific task:

- *state processing pipeline*: the first step each state goes through is a preprocessing routine. The states might have some redundancy that increases the state space without adding relevant information about the current situation. This is true in

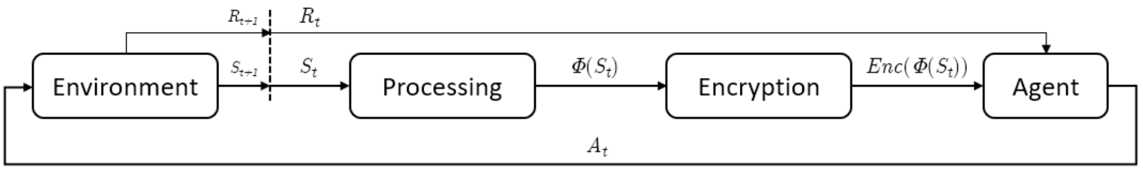


Figure 3.1: General workflow proposed for the project. The environment, at each time step t, outputs a state S_t. This state is processed with a scenario-specific preprocessing function $\phi(S_t)$; its output is then encrypted with an encryption function $Enc(\phi(S_t))$, which is what the agent receives alongside the reward R_t. The agent, with this information, chooses an action A_t.

general, and many algorithms implement a processing step to speed up and stabilize the agent's learning, but even more so when cryptography is involved, because less data to encrypt also means less noise as input to the agent.

- *encryption*: after the preprocessing, each state undergoes encryption with a given cryptographic primitive.
- *rl agent*: the last entity in our algorithm is the reinforcement learning agent, that receives the encrypted states and performs training on them.

Ideally, in the use case we defined in the previous section, the three components are situated in different physical places. Going back to the plant example, the state processing and the encryption step would be performed by the owner of the plant. The encrypted data would then be sent to the cloud provider that feeds it to the reinforcement learning algorithm which will output what is the best action to perform. Afterwards, the representation of the action will be sent back to the robot of the plant, which will decode its meaning and physically act on the environment.

More in depth, we can extend the general MDP framework we outlined in 2.1.3. We will define t as the current time step of the environment, and $S_t \in \mathcal{S}$ as the state at the given time step, where $\mathcal{S}$ is the set of all possible states; $\phi(S_t)$ is the preprocessing function, carried out by the first component listed above, that transforms the state S_t, and $Enc(\cdot)$ is the chosen encryption method, performed by our second component, that accepts as input the processed state $\psi(S_t)$. The environment, at each time step t, outputs a state S_t which encodes its current configuration, and a numerical reward R_t. Now, however, the state is not directly sent to the agent, the third component, but undergoes a sequence of transformations beforehand. Firstly, the state is processed with the pre-processing function ϕ, that outputs $\phi(S_t)$. Then, the encryption method encrypts the processed state, resulting in the ciphertext $c = Enc(\phi(S_t))$. To simplify the notation, we will call $\psi(S_t) = Enc(\phi(S_t))$. The reinforcement learning agent receives this encrypted representation of the state alongside the unchanged numerical reward R_t and uses this knowledge to choose an action $A_t \in \mathcal{A}(\psi(S_t))$. Therefore, the *trajectory* of the agent-environment interaction (2.1) will now take the form

$$Enc(\phi(S_0)), A_0, R_1, Enc(\phi(S_1)), A_1, R_2, Enc(\phi(S_2)), A_2, R_3, \dots \quad (3.1)$$

$$= \psi(S_0), A_0, R_1, \psi(S_1), A_1, R_2, \psi(S_2), A_2, R_3, \dots. \quad (3.2)$$

With the extended MDP framework, we can also redefine the policy π as

$$\pi(A_t = a | S_t = \psi(s)) \quad (3.3)$$

and the state-value functions and action-value functions as

$$v_\pi(\psi(s)) \doteq \mathbb{E}_\pi[G_t | S_t = \psi(s)] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = \psi(s)\right] \quad (3.4)$$

$$q_\pi(\psi(s), a) \doteq \mathbb{E}_\pi[G_t | S_t = \psi(s), A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = \psi(s), A_t = a\right]. \quad (3.5)$$

The key takeout is that the agent has no access to the internal, raw state of the environment, but only sees the encrypted and processed version $\psi(s)$ of the states, and tries to learn and to choose actions on them.

Under these assumptions, we consider the behaviour of the agent that interacts with such environment, with the goal of choosing the actions that maximize the future rewards, discounted with the discount factor γ, so that at time step t, the future discounted return is $G_t = \sum_{k=t}^{T} \gamma^{k-t} R_k$, where T is the time step at which the episode terminates. The agent selects and executes action with an ϵ-greedy policy based on the learned action-value function q, that works on the processed states $\psi(s) = Enc(\phi(s))$.

Specifically, the reinforcement learning model employed for the task is the DQN with target network, as seen in [5], since it is a well-known and widespread deep reinforcement learning algorithm that is also more sample-efficient with respect to other modern techniques. In the context of the extended MDP framework of above, the update rule 2.15 becomes

$$w_{t+1} = w_t + \alpha\big[R_{t+1} + \gamma \max_a \hat{q}(\psi(S_{t+1}), a, w_t^-) - \hat{q}(\psi(S_t), A_t, w_t)\big] \nabla \hat{q}(\psi(S_t), A_t, w_t) \quad (3.6)$$

where the target values for the update, represented by

$$y_t = R_t + \gamma \max_a \hat{q}(\psi(S_t), a, w_t^-) \tag{3.7}$$

are computed by the *target* network, whose weights w^- are cloned from the *online* network after every C updates.

Similarly, the agent's *replay memory* will store experience tuples of the form

$$e = (\psi(S_t), A_t, R_{t+1}, \psi(S_{t+1}))$$

and the updates of the neural network are performed by randomly sampling a minibatch of transitions from the replay memory and using them to compute a target for Q-Learning. This sampling can start as soon as the replay memory buffer contains enough transitions to fully form a minibatch.

The only difference with the original algorithm is that, instead of either making the agent work directly with the states output by the environment or preprocessing them before, we include an additional layer of state manipulation between the preprocessing and the agent in the form of the encryption function $Enc(\cdot)$. This encryption has the purpose of hiding the features of the states to the agent and to a hypothetical third party that tries to read the communications that go through the channel, which is considered insecure by default. The pseudocode for the DQN algorithm modified under this assumption is shown in Algorithm 2.

3.3 Summary

In this chapter we started from the background knowledge we have built up previously and have explained the central concept behind our work. Indeed, we started off by illustrating the general problem of reinforcement learning over encrypted data by detailing a possible use case and what innovation it would bring to the discipline, then we have defined the research questions that will be central in the experiments.

Afterwards, in the second part, we have formally defined the problem. Initially, we have detailed the components that are part of our application, each with their scope and purpose. Then, we have defined the pipeline that we will follow by extending the base MDP framework that we presented in Section 2.1.3; lastly, we have formalised the algorithm that will be employed in the experimental part of our project.

The next two chapters will start from here and will build upon the theoretical frame-

Algorithm 2: DQN with encrypted states

```
Parameters:
    replay memory capacity $N$
    small $\epsilon > 0$
    amount of cloning steps $C$
Initialize:
    replay memory $D$ to capacity $N$
    action-value function $\hat{q}$ with random weights $w$
    target action-value function $\hat{q}$ with weights $w^- = w$
foreach episode do
    Initialize $S_0$
    Obtain $\psi(S_0) = Enc(\phi(S_0))$
    foreach step of episode $t$ until terminal state $S_T$ do
        With probability $\epsilon$ select a random action $A_t$, otherwise
          $A_t = \text{argmax}_a \hat{q}(\psi(S_t), a; w)$
        Execute action $A_t$ and observe $R_t, S_{t+1}$
        Obtain $\psi(S_{t+1})$
        Store transition $(\psi(S_t), A_t, R_t, \psi(S_{t+1}))$ in $D$
        if $D$ contains enough transitions then
            Sample random minibatch of transitions $(\psi(S_j), A_j, R_j, \psi(S_{j+1}))$ from
              $D$
            Set $y_j = \begin{cases} r_j & \text{if } S_{j+1} \text{ is terminal} \\ r_j + \gamma \max_{a'} \hat{q}(\psi(S_{j+1}), a'; w^-) & \text{otherwise} \end{cases}$
            Perform a gradient descent step on $(y_j - \hat{q}(\psi(S_j, a_j; w))^2$ w.r.t.
              network parameters $w$
        end
        Every $C$ steps clone $w^- = w$
    end
end
```

work that has been built in this chapter. Indeed, we will show two case studies, on two different environments, that represent the concretization of the problem and of the components that were defined here. Each case study will instatiate all parts of the pipeline $\psi(\cdot)$ and we will provide a mapping between these components and their specific implementation for the task at hand.

Chapter 4

First Case Study: MiniGrid

After outlining the general problem of reinforcement learning over encrypted data, the next step would be to find out if a RL agent can actually learn anything from an encrypted state. For this reason, we will now try to answer the research questions we outlined in Chapter 3. To do that, we will follow the general algorithm defined there and apply our implementation of such algorithm to various environments whose states will be encrypted, while thoroughly describing how each component will be realized.

More in depth, this and the next chapters will be case studies of two instantiations of the problem under different environments. In particular, this chapter will focus on the application of our reinforcement learning agent to MiniGrid, a gridworld-based (and as such, discrete) navigation environment. This environment represents an interesting case study because it features a simple task, in which the agent needs to reach the goal tile, while also being straightforward to analyze due to its visual nature.

In the first section, we will start by describing the experimental settings of the first experiments. Here, we will go over the key aspects of the subset of MiniGrid environments we considered; then, we will define the concretizations of the preprocessing function $\psi(\cdot)$ and the encryption function $Enc(\cdot)$, so that the complete state processing pipeline $\psi(\cdot)$ will be determined. Afterwards, the neural network architecture will be explicated; finally, we will list the hyperparameters chosen for the task.

The second section will focus on the implementation details of the classes developed to carry out such task. Each component will have their implementation structure and functional interface defined, starting by the environments itself and then listing all the classes responsible for the actualization of our processing pipeline, ending with the class of the central component of the algorithm, the agent.

Finally, the last section of this chapter will go over all the experimental results that

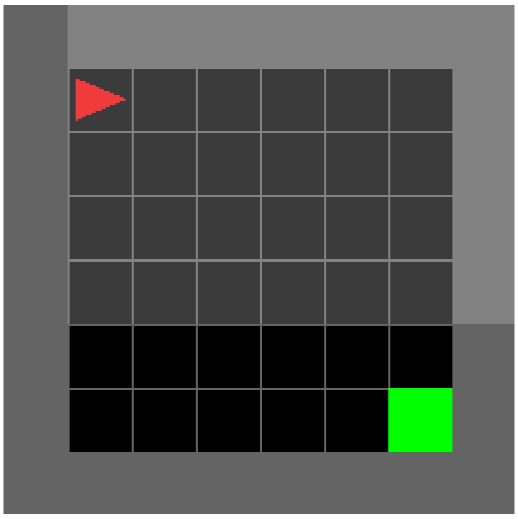


Figure 4.1: Empty 8x8 environment in MiniGrid. The agent is a red triangle that starts in the top-left corner of an empty room and has to reach the goal at the opposite corner. Courtesy of [37].

the algorithm was able to achieve under the settings previously detailed.

4.1 Experimental Settings

The first part of our experiments were focused on the application of our reinforcement learning agent to a gridworld-based environment. A widespread suite of gridworld environments is MiniGrid [37], an OpenAI Gym extension that offers a great number of bidimensional and discrete navigation games.

4.1.1 MiniGrid

In particular, the environments chosen for carrying out the experiments of this work are the empty room environments, where the agent finds itself in an empty square room and has the goal of reaching the destination, expressed by a green tile, situated in the bottom-right corner. The starting position is at the opposite corner to the destination, or a randomly chosen tile and direction in the random variants. At each time step, the agent is described by its position (i, j) in the grid and the direction it is currently facing in, one between right, upwards, left, or downwards. The empty rooms are of variable size, namely 5x5, 6x6, 8x8, and 16x16.

Observations

While MiniGrid, by default, outputs observations as Python dictionaries describing the current situation in the environment, it also provides wrappers that enable using images of the grid as states. Images are more information-rich of simple dictionaries and allow us to easily interpret the current situation in case of plaintext, and to spot out recurring patterns immediately for the encrypted case; hence, the observations for the experiments are $\langle H \times W \times 3 \rangle$ RGB images of the rooms.

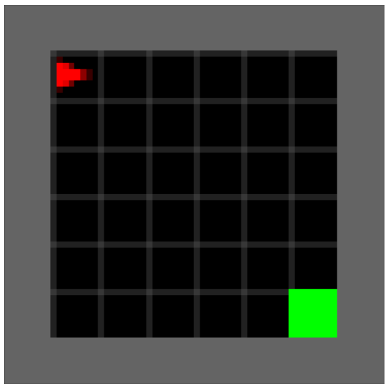


Figure 4.2: Observation of the starting state of the 8x8 empty room environment in MiniGrid. Each tile is 8x8 pixels, and there is an outer gray wall surrounding the walkable area.

Action Space

As noted above, MiniGrid offers many environmental configurations, with some even involving openable doors, lava, moving obstacles, and pickable objects like keys. For this reason, the action space includes some actions that are not needed in all the environments: particularly, the empty rooms involve no objects nor doors, therefore the action space has been reduced to the minimal set. Both the actions spaces are listed in Table 4.1.

Rewards

Lastly, MiniGrid's default reward function is characterized by *sparse* rewards. Indeed, only a portion of the states gives the agent a feedback: in our case, the agent gets no reward until it reaches the destination. The reward assigned, then, is calculated as follows:

$$R_T = 1.0 - 0.9\frac{T}{L} \tag{4.1}$$

Number	Meaning
0	**Turn left**
1	**Turn right**
2	**Move forward**
3	Pick up object
4	Drop object
5	Toggle
6	Done

Table 4.1: MiniGrid discrete action space comprises of many actions, some of which are not useful in all environments. The subset of actions considered for the empty room environments is highlighted in bold text.

where T is the time step in which the agent reached the goal and L is the maximum amount of time steps in an episode. If, instead, the agent did not reach the goal after L time steps, the episode is concluded and the reward is 0. We decided to keep the default reward function since it is a good indicator of how many steps are required to reach the goal.

4.1.2 State Preprocessing

In Section 3.2 we have formalized an extended version of the classic MDP by adding two additional state transformations: a preprocessing function $\phi(\cdot)$ and an encryption function $Enc(\cdot)$, which, piped together, are denoted as $\psi(\cdot)$. In this section, we will focus on the preprocessing function $\phi(\cdot)$.

MiniGrid allows us to obtain colour images as states of the environment. These images are three-dimensional RGB integer arrays of shape $\langle H \times W \times 3 \rangle$, as shown in Figure 4.2. Each tile occupies a surface of 8x8 pixels, and every room is wrapped by a layer of non-walkable gray wall. This means that the size of the image is 8 times bigger than the size of the corresponding environment: for example, the 8x8 environment outputs 64x64 images.

Some of the information carried by the images is, in some way, redundant. In our implementation there is no need for 8-pixels wide tiles, nor for the outer walls. Having redundant information not only slows down learning in the basic plaintext case, but also increases the number of possible states for the encryption case, making the overall task much harder.

For this reason, the first step each image undergoes is a downsampling of factor 8, so that each tile is represented by a single pixel. Then, the image is converted from RGB to grayscale. A downside of resizing the images to the minimal possible dimension is that

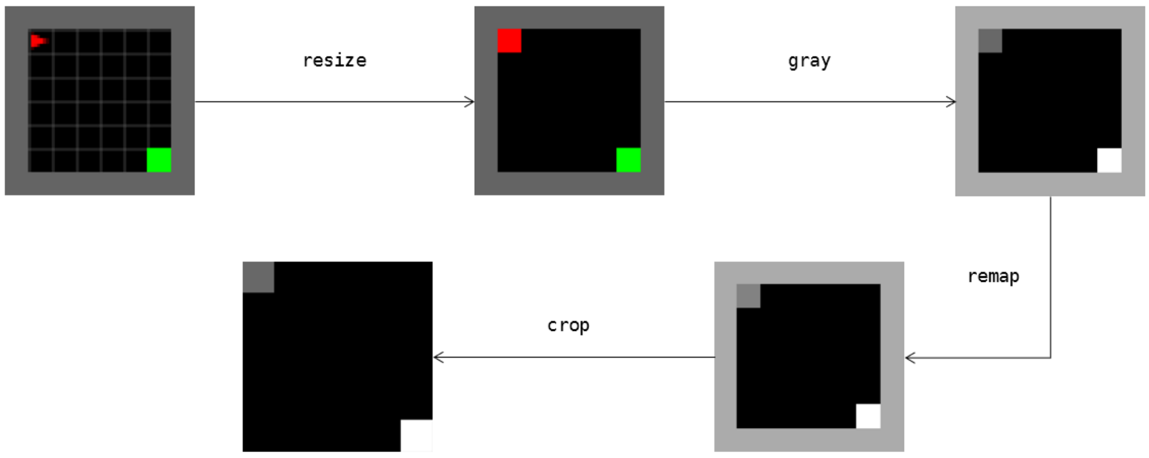


Figure 4.3: Image processing pipeline for the 8x8 environment. First, the image is resized by a factor of 8, so that each tile is represented by a single pixel. Then it is converted to grayscale, and the direction of the agent is mapped with a fixed pixel intensity. Finally, the image is cropped.

the direction of the agent is no longer discernible, because it is not possible to see where the tip of the red triangle representing the agent is currently facing. This would make it seem like the actions of turning left or right have no impact on the environment; so, the direction of the agent is remapped into four different pixel intensities, by changing the value of the pixel corresponding to the agent's current position to a fixed value for each direction. Moreover, during this step, the intensities of the pixels are also remapped to cover the whole $[0, 255]$ range, so to favour colour separation.

Afterwards, the outer walls are cropped to further reduce the size of the image. In the empty room environments, the walls are always a square outer layer that wraps the walkable grid. Therefore, this step only consists in deleting the first and last rows and the first and last columns from the image. Figure 4.3 shows a schematic of the complete pipeline.

4.1.3 Encryption

In the MDP extension, after the preprocessing function $\phi(\cdot)$, the next step consists in encrypting the result of the process with a given encryption function $Enc(\cdot)$. Therefore, we will now focus on such function.

The cryptographic scheme chosen for our experiments is *AES* [42], a widespread symmetric block cipher. The reason for choosing a symmetric encryption scheme, and

AES in particular, are two. Firstly, also recalling the use case outlined in the previous chapter, the easiest and fastest way to obtain privacy-preserving images of states is to use a symmetric encryption scheme with a key that only the owner of the data possesses. Our models do not need to decrypt the images, since they will be trained on the encrypted versions. Secondly, between the symmetric key schemes AES with 256 bit key is a good candidate, because it is considered cryptographically secure and it is widely used.

However, the first encryption-like test is performed with a deterministic scramble of the pixels of the image. Before the run, the indexes of the pixels of the image are randomly scrambled and the result is stored in a lookup table that encodes a translation for each pixel position to a new position. Form then on, each image during training is scrambled in the same way. This step, while not being a proper encryption, acts as a middle ground between plaintext and ciphertext and it is useful to asses the performance of the agent.

Then, the agent is trained on the images encrypted with AES in ECB mode. This mode, as noted in Section 2.2.2, is not cryptographically secure because it is deterministic and we can still spot recurring patterns in the ciphertext. The key is randomly generated before the start of the training.

Lastly, the agent is trained on the CBC version of AES. The key is randomly generated before the start of the training, then the IV is randomly reinitialized before every encryption step. This makes it so, even with the same plaintext, the ciphertext is different.

Padding

Since AES is a block cipher, the plaintext size needs to be a multiple of the block size, in this case 16 bytes. For this reason, the images of the states need to be padded to comply to this requirement, and the padding type is another parameter of our pipeline. We identified to possible candidates: a custom padding, used to preserve the spatial information of the pixels within the image, or PKCS#7 [40], the current cryptographic standard.

Since the experiments will be carried out using block cipher primitives, the plaintext needs to be padded to a multiple of the block size before being encrypted. In our case, we identified two possible padding procedures: a custom "wall" padding, that pads the image trying to maintain the spatial information of the pixels, and PKCS#7 [40] padding, the current cryptographic standard, that pads the trailing end of the plaintext with $k - (l \mod k)$ bytes, each having value $k - (l \mod k)$, where l is the length of the input, and k the block size. This primitive, however, works on one-dimensional array of bytes, therefore the resulting padded message, once reassembled in a $H \times W$ shape, will feature an additional layer of pixel scrambling.

We also added another encryption-like test to ease in the complexity. This procedure consists in a deterministic scramble of the pixels of the image. Before the run, the indexes of the pixels of the image are randomly scrambled and the result is stored in a lookup table that encodes a translation for each pixel position to a new position. Form then on, each image during training is scrambled in the same way. This step, while not being a proper encryption, acts as a middle ground between plaintext and ciphertext and it is useful to asses the performance of the agent.

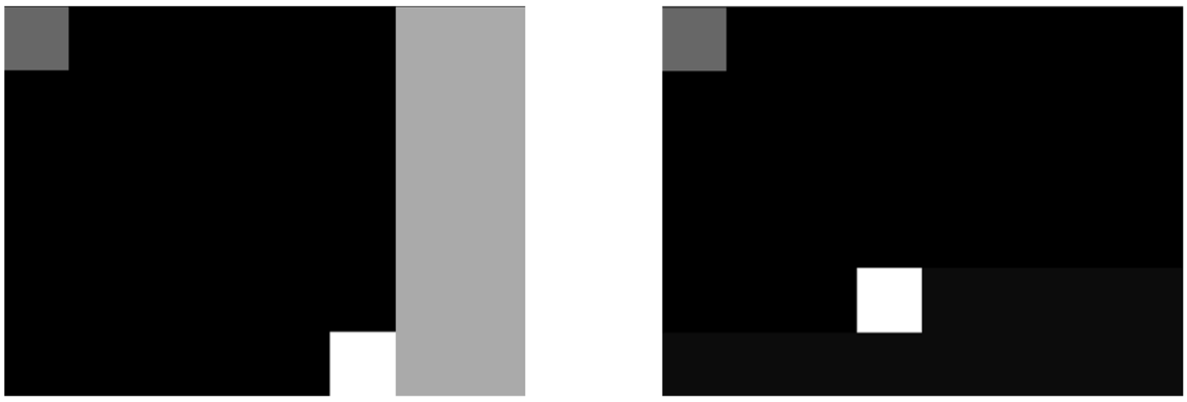


Figure 4.4: Comparison of the two padding techniques in the 8x8 environment. On the left, the custom padding that maintains pixel distribution. On the right, the standard PKCS#7 padding. In this case, it is possible to see the twelve padding pixels of intensity value 12 after the goal.

More in depth, using PKCS#7 would change the shape of the image fed to the neural network, therefore losing spatial information of the pixels. For example, the 5x5 environment outputs 40x40 images. After the preprocessing, the images are scaled down to simple 3x3 and, as noted above, AES requires 16-byte blocks, so we need 7 additional bytes to reach the threshold. If we simply apply PKCS#7, we would append the padding bytes at the end of the byte array; afterwards, when reshaping this byte array to a $H \times W$ shape to perform convolutions, we would end up with a 4x4 image, in which the pixels are no longer in their original position: pixel in position $(2, 1)$ would now be in position $(1, 4)$ and so on, effectively adding a layer of pixel scrambling. Hence, we also devised an additional padding procedure which is performed by adding rows or columns, enough to reach a size multiple to the block size, to the resized image, filled with values corresponding to the pixel intensity of the walls. In this way each pixel retains its original position when the encrypted image is given to the CNN, but it has the disadvantage of possibly increasing the number of padding bytes. Both of the padding techniques will be tested during the experiments.

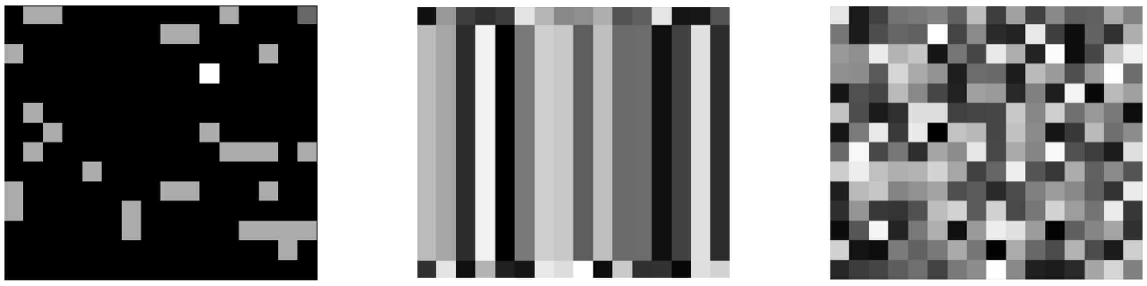


Figure 4.5: Example of encrypted states in the 16x16 environment. On the left, the deterministic shuffle. In the middle, AES in ECB mode. On the right, AES in CBC mode. The state that is being encrypted is the same for all these examples, and it corresponds to the initial state, with the agent in the top-left corner facing to the right.

4.1.4 Neural Network Architecture

The structure of the ANN depends on the current environment, but all the agents implemented feature at least two convolutional layers and two fully connected layers, and a number of outputs equal to the cardinality of the subset of valid actions. Convolutional neural networks introduce spatial inductive biases that are useful for capturing local features withing the grids (for example, close positions in the grid lead to similar rewards) and reduce the total amount of training parameters, enabling faster learning, hence are appropriate for our objective.

The convolutional layers take as input the image of the state of the environment, with particular regards to its number of channels. Each layer outputs a *feature map*, built with the usage of a *filter* that passes through the image and computes the convolution of the pixels inside its *kernel*, in a sliding window approach. The resulting feature map is characterized by a certain number of channels, and it is passed as input to the following convolutional layers.

Usually, in a CNN, after the convolutional layers, there also are some fully connected layers. Hence, the feature map output of the last convolutional layer needs to be *flattened*, that is, transformed to a one-dimensional collection of features, in order to be passed to the fully connected layers. Each of these layers takes as input for each unit all the outputs of the previous layer's units. The final layer consists in a fully connected layer with a number of outputs equal to the number of actions $|\mathcal{A}(S_t)|$.

Specifically, for the 5x5, 6x6, and 8x8 environments the neural network structure consists in a first convolutional layer that takes as input the images with 1 channel (since they are grayscale), and outputs 4 channels, with a kernel of size (2, 2) and a stride of 1. Then, a second convolutional layer takes as input 4 channels, outputs 8 channels, with the same kernel and stride of the previous layer. Given the size of the input images, the very small kernel size and stride turned out to be appropriate for the task. The resulting feature map is then flattened and given as input to 32 fully connected units. A final, 3-units layer outputs the Q-values for each action. The activation function for all the hidden layers is the Rectified Linear Unit (ReLU).

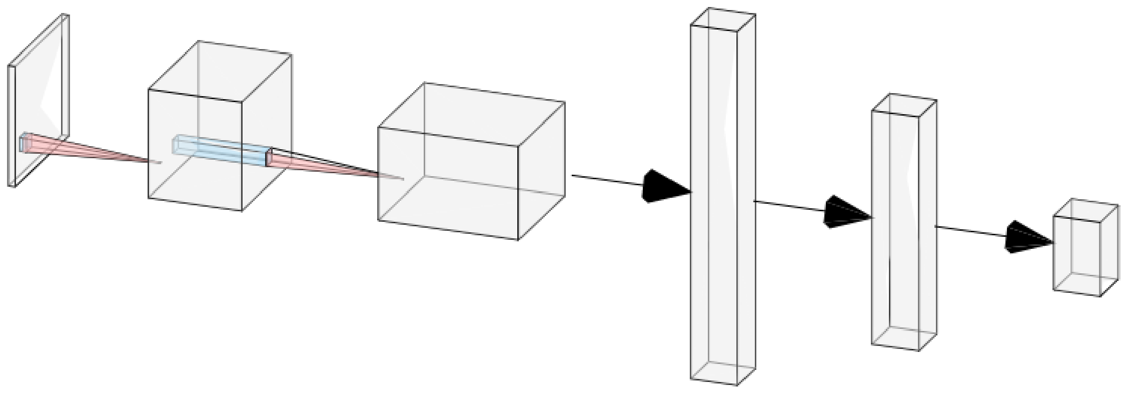


Figure 4.6: Illustration of the CNN architecture for the 8x8 state. The input image, after preprocessing, is of shape 6x8x1. The first convolutional filter outputs a 5x7x4 feature map. The second convolutional filter outputs a 4x6x8 feature map, which is flattened into 192 units. Then, there are a fully connected layer of 32 neurons and the output layer of 3 nodes, one for each action. Illustration created with [70].

This small architecture proved not to be sufficient for the bigger 16x16 environment. The architecture is then modified as follows: the first convolutional layer takes the 1-channeled image and outputs 2 channels via a (2, 2) kernel with stride 1; then, the second convolutional layer takes the 2-channeled feature map and outputs 4 channels with the same filter; a final convolutional layer transforms the 4 channels into a 8-channel feature map with the same filter. This feature map is then flattened and processed with two 32-units fully connected layers and a final fully connected layers with 3 unis that outputs the Q-values for each action. Also in this case, the activation functions for the hidden layers are ReLUs.

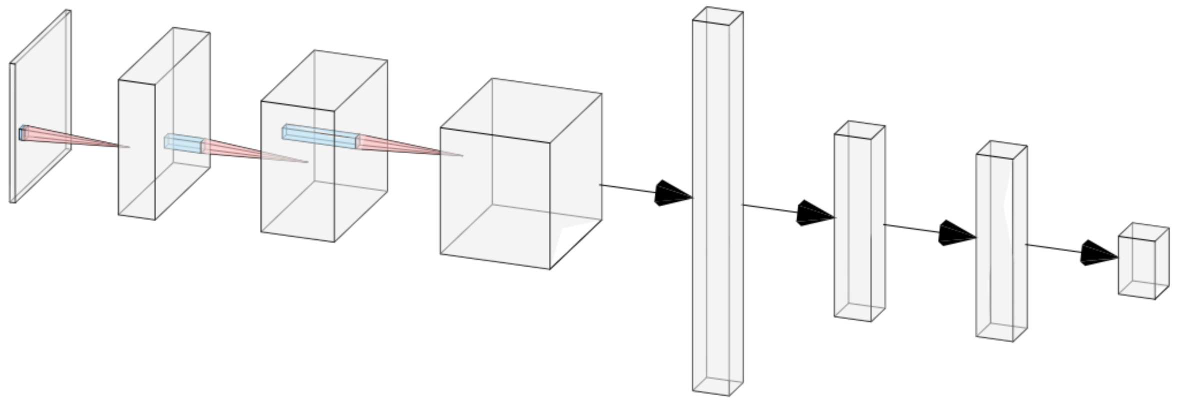


Figure 4.7: CNN architecture for the 16x16 state. The input image, after preprocessing, has a shape of 14x16x1. Then, three convolutional layers are applied: the first outputs a 13x15x2 map, the second 12x14x4, and the third 11x13x8. The last feature map is flattened into 1144 units. Then, two 32-units fully connected layers and a final output layer of 3 nodes are applied. Illustration created with [70].

4.1.5 Hyperparameters

The choice of hyperparameters is dependant on the environment's characteristics. As we mentioned earlier, MiniGrid's empty room have varying sizes, and the relative hyperparameters have been found empirically. These values are then kept consistent for all the experiments, both in the fixed starting state case and in the random starting states, in order to favor a correct comparison. Table 4.2 lists all the hyperparameters used and their values.

4.2 Implementation

We will now go over the implementation details for the application of reinforcement learning over encrypted data. The programming language of choice is *Python*, because of its greatly active communities of statistics and machine learning, and for the presence of very useful scientific computation packages such as *NumPy* and *PyTorch*.

Parameter	**5x5**	**6x6**	**8x8**	**16x16**
N frames	$2 \cdot 10^5$	$2 \cdot 10^5$	$2.5 \cdot 10^5$	$4 \cdot 10^5$
Learning rate	$1 \cdot 10^{-3}$	$1 \cdot 10^{-3}$	$5 \cdot 10^{-4}$	$1 \cdot 10^{-4}$
Memory size	$1.5 \cdot 10^5$	$1.5 \cdot 10^5$	$1.5 \cdot 10^5$	$1.5 \cdot 10^5$
Batch size	64	64	64	64
ϵ-decay	0.995	0.995	0.995	0.995
ϵ-max	1	1	1	1
ϵ-min	0.1	0.1	0.1	0.1
γ	0.9	0.9	0.9	0.9
C	100	100	100	100

Table 4.2: Hyperparameters for each MiniGrid environment.

4.2.1 Environment Wrapper

The environment, created with Gym's `Gym.make()`, is wrapped with MiniGrid's wrappers `RGBImgObservationWrapper` and `ImgObservationWrapper`, that allows to obtain the observation of the current state of the environment as RGB images, stripping the state of the natural lanuage string that describes the goal of the agent. An additional wrapper, named `EnvWrapper`, is then created. This class has the task of carrying out all the conversions between *NumPy* arrays, returned by the environment, and *PyTorch* tensors, required for the neural network computations.

This class also incorporates the classes for the image processing and the encryption that will be overviewed in the later sections, so that all the transformations of the state are independent of the agent class. It exposes a functional interface that is identical to the one of Gym's environment object, so that the agent does not need specific functions to handle the wrapper, and it can be used indistinguishably with both a wrapped or unwrapped environment. Notably, the wrapper instantiates the image processing class, the padding class, and the encryption class based on the requirements of each experiment, then stores them into class proprieties. The `EnvWrapper` class, among some other minor utilities, offers the following core methods:

- `transform(state, position, direction)`: takes as input the non processed current state of the environment and the agent's current position and direction. First, it calls the processing method from the image processing class, then, if needed, the encryption method from the encryption class. It returns a *PyTorch* tensor with the processed state, may it be encrypted or in plaintext.
- `step(action)`: takes as input the chosen action and performs it in the underlying environment. It returns the reward, the `done` flag and the processed new state.

- `reset()`: resets the environment to the initial configuration. It returns the processed state.
- `clone()`: creates an independent deep copy of the environment wrapper and of the underlying processing, encryption, and environment objects. This clone is identical to the calling object, and, if used in an encrypted case, it also shares the same encryption key or index table. The main usage of this function is to create an independent object with a different environment seed to be used in evaluation mode, in order to not interfere with the training environment.
- `seed(seed)`: sets the given random seed in the underlying Gym environment. This is especially useful when using environments with random components, such as random starting states or random obstacles.

4.2.2 Image Processing

The image processing pipeline $\phi(\cdot)$ is implemented in the class `ImgProcessor`, that handles all the preprocessing-related tasks. During instantiation, it can disable either the cropping or the risizing portion of the pipeline. It exposes the method `transform`, that takes as input the image of the current state and the position and direction of the agent, and it performs the preprocessing previously explained to return the processed image.

Both the downsampling and the greyscale conversion are implemented via OpenCV [71] utilities. The conversion is performed using luminance values as follows:

$$Y \leftarrow 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B \tag{4.2}$$

where R, G, B are the luminance values of the Red, Green, and Blue channels of each pixel, as provided by [72].

4.2.3 Encryption

The encryption-related tasks underlying the function $Enc(\cdot)$ are handled by classes specific for each type. All the classes implement an abstract `BaseEncryptUtils`, that exposes the abstract method `encrypt(plain)`. Then, each encryption procedure is implemented in a separate class:

- `HomomorphicUtils` implements the homomorphic encryption case. Its `encrypt` method encrypts the given plaintext in an array of bytes by calling *PyFhel*'s `encryptInt` primitive.

- `DeterministicShuffler` handles the deterministic shuffle operation. During initialisation, it creates a lookup table by randomly shuffling the original pixel indexes, then it uses it in the `encrypt` method to return the corresponding ciphertext.
- `AESCrypto` is the class for the AES-based encryption methods. It extends the `BaseEncryptUtils` class by overriding its `init` method, specifically to istantiate the AES algoithm from the library *pyca/cryptography*, while leaving the `encrypt` method abstract. This method is then specialized by the two subclasses `ECBCrypto` and `CBCCrypto`, that implement it according to the right modality.

The factory `Crypto` has the task of instantiating the correct cryptographic subclass based on its input parameters, so that the addition of new cryptographic methods is straightforward.

4.2.4 Padding

The two padding techniques we have mentioned in Section 4.1.3 are implemented each in a separate class that extends an abstract `BasePadder`. The former is handled by the class `CustomPadder`, that pads the image adding walls to the two-dimensional image array. During initialisation, it computes a suitable new shape for the image, then proceeds to pad each state with the method `pad(img)`, that takes the current state of the environment, pads it with the aforementioned procedure, then returns it as a mono-dimensional array of bytes. The latter, instead, is realized in the class `PKCSPadder`, that converts the state image first to an array of bytes, then pads it accordingly with *pyca/cryptography* utilities. The creation of the proper class is delegated to the *factory class* `Padder`, that instantiates the correct padding class based on its input parameters, so that the addition of new padding techniques is straightforward.

4.2.5 Agent

The neural networks are created inside an `Agent` class that models the behaviour of the reinforcement learning agent. The agent class is the core element of the application, since it is the one responsible for the creation of the neural network and for the definition of the behaviour in the environment, while also being in charge of carrying out the gradient descent step. It handles the exploration-exploitation tradeoff and interacts with the environment at each time step, while also storing its experience in the *replay memory* buffer. In particular, the `Agent` first creates the online network and the target network and stores them into the variables `online_net` and `target_net`, which are saved as class proprieties. At each time step t,the agent takes the image of the current state S_t and, after the preprocessing step, feeds it to the neural network, which approximates the action-value function. It obtains estimates $\hat{q}(S_t, a, w_t)$ for each $a \in \mathcal{A}(S_t)$.

The next step consists in choosing an *optimizer*, which is the component in charge of performing the gradient descent optimization of the neural networks' weights. One key aspect of the optimizer is the *learning rate* parameter, that controls the magnitude of the changes in the weights with respect to the estimated error, the *loss*. If the learning rate is too small, the training will slow down significantly and there is a risk of getting stuck in a local minimum, while leading to overfitting and unstable learning otherwise. The optimizer chosen is *Adam* [73], since it is a standard optimizer widely used because of its adaptive learning rate scheduler, that can speed up learning and help avoid manual exploration with constant learning rates [74]. The *PyTorch* package offers the implementation of this optimizer in its `torch.optim` module. Instead, for the *learning rate*, the values in Table 4.2 were chosen empirically by conducting a search over the learning rates $0.005, 0.001, 0.0005, 0.0001$ and found 0.001 to be the most effective for the smaller environments, and 0.0005 and 0.0001 for 8x8 and 16x16 respectively.

Another very important factor in every agent implementation is how to balance exploration and exploitation in a satisfying manner (recall Section 2.1). At the beginning of training, the agent has no knowledge of what its task will be, therefore it has to explore very often in order to discover which are the best actions in each situation. As the agent builds some knowledge of the task and of the environment, it will gradually drift towards a more greedy approach, choosing actions that led to good rewards in the past. For this reason we chose to follow an ϵ-greedy approach, with ϵ decaying exponentially for each episode, starting from ϵ_{max} to a minimum value ϵ_{min}. Therefore the agent, at the beginning of training, will behave randomly very often, with a probability ϵ_{max}. The probability will gradually decrease to ϵ_{min} as more episodes are completed, meaning that the random actions will be selected fewer and fewer times. The agent computes the current ϵ value with the method `get_eps()`, that returns the maximum value between ϵ_{min} and the currently decaying ϵ.

The DQN algorithm relies on the technique known as *experience replay* (recall Deep Q-Learning, 2.1.6). The replay memory is realized in the agent as a separate class called `ReplayMemory`, that implements the replay memory buffer as a circular buffer of a fixed size N. Each entry of this list consists in a tuple (S_t, A_t, R_{t+1}, S_{t+1}, `done`), where S_t is the image of the state at time step t, A_t is the action taken in that time step, R_{t+1} is the reward received by the environment in the successive time step, S_{t+1} is the image of the state resulting from the action, and `done` is a flag that determines if the episode ended.

The `ReplayMemory` class provides two methods:

- `push(item)`: stores the given item, in the form of the aforementioned tuple, into the buffer. If the current size of the buffer would exceed N, instead the oldest tuple is replaced, following a FIFO criterion.
- `sample(batch_size)`: uniformly samples a number `batch_size` of tuples from the buffer.

Each experience is obtained by the agent while interacting with the environment. When it gathers all the components required, it calls its method `remember(s,a,r,s_,d)` which stores the components into the tuple and calls the method `push` from the replay memory.

The agent also offers other relevant methods:

- `target_network_update()`: every C updates to the neural network, it clones the weights of the `online_net` into the `target_net`, as happens in [5].
- `choose_action(state)`: takes the current state of the environment and decides what action to perform. More specifically, it generates a random number p: if $p < \epsilon$, the next action will be random. If instead $p \geq \epsilon$, the state is given to the `policy_net` and the next action will be the greedy one with respect to the q-values estimated by the neural network.
- `update()`: samples a number `batch_size` of experiences from the replay memory and uses them to create the target for the neural network update. Then it computes the loss and calls the optimizer to perform the gradient descent step.

4.3 Experimental Results

This section has the purpose of evaluating the application outlined in the previous pages, by highlighting the results obtained by training the reinforcement learning agent on the encrypted states from MiniGrid, with the aim of studying the impact of the encryption step on the training performance, so that we can provide an answer the the questions defined in Chapter 3.

For the first part of the experiments, the agent was run on the whole set of environments for N steps, starting from the 5x5 grid and scaling up to the 16x16 grid. For each environment, the training was performed every time on a different version of encryption.

The first run, that acts like a baseline, was performed on the plaintext case, in which the images of the states only undergo the preprocessing explained in Section 4.3, but not the encryption. Then, the same training is performed once again with the deterministic shuffle case; afterwards, the training is repeated on the ECB version and, lastly, on the CBC version. The details of these techniques can be found in Section 4.1.3.

The first set of environments tested was the empty room environments with the fixed starting states. Afterwards, we studied the impact of the key length and of the padding type on the same set of environments. Lastly, we investigated the impact of a random starting state and position on the performance of the agent.

Moreover, each training batch is performed 10 times, each run with a different random seed. Indeed, the agent's performance is heavily influenced by randomness. A certain set of random actions performed during the exploration-heavy parts of the training might help the agent to converge faster, while a different set might prevent it from converging at all. Using a fixed seed allows us to correctly repeat the experiments, and repeating the runs over 10 different seeds grants us some statistical stability, while also assessing the agent's resilience to noise.

4.3.1 Encryption Overhead

All the training runs and the time measurements reported were performed on an ASUS Laptop with 16 GB of RAM, an Intel Core i7-6700HQ CPU on 4 cores at 2.6 GHz, and a Nvidia GeForce GTX 960M GPU with 2GB memory.

One of the first step performed to evaluate the agent over encrypted data was to measure the overhead caused by the encryption step during the state processing pipeline. Since reinforcement learning is a very interactive problem, a big encryption overhead would reduce drastically the feasibility of this approach. These results are compared to a sample of a homomorphic encryption approach, typical of the papers discussed in Section 2.3, such as ML Confidential [8], CryptoNets [9], and CryptoDL [11]. The homomorphic encryption primitives are implemented in Pyfhel [75], a Python wrapper around Microsoft SEAL.

As Figure 4.8 suggests, the encryption step for the considered cryptographic functions does not induce a significant overhead. On the other hand, we can see that homomorphic encryption's time complexity, with respect to AES, is hundreds, or thousands for the biggest state, times slower. Thus, homomorphic encryption, as of now, does not represent a feasible encryption method for reinforcement learning problems.

Indeed, as Table 4.3 suggests, the overhead induced by the encryption step is minimal

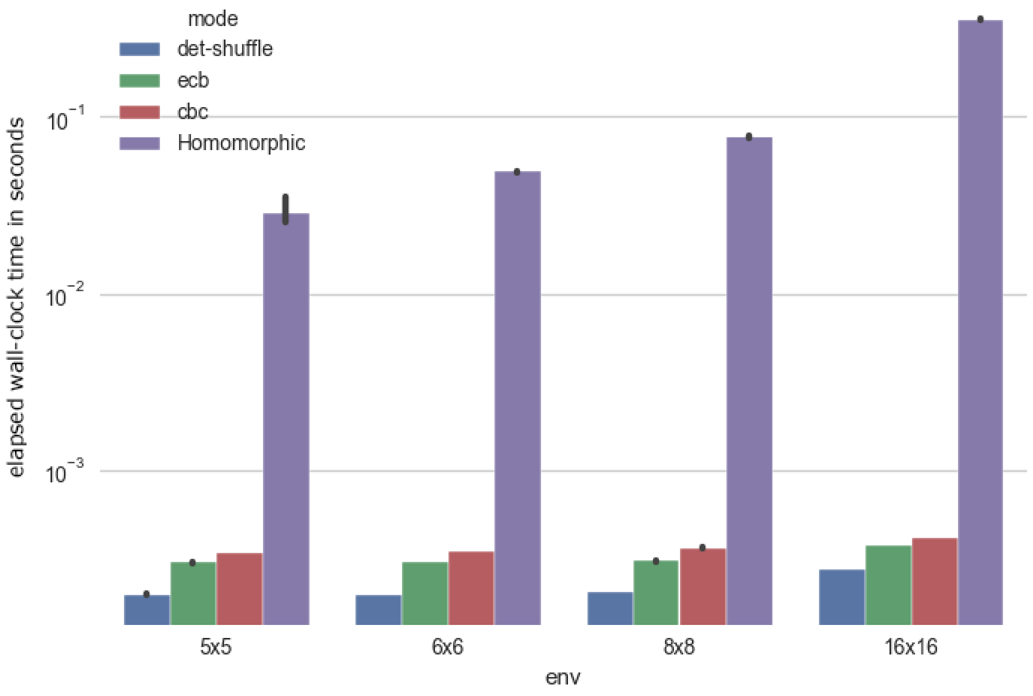


Figure 4.8: Encryption timings in seconds.

with respect to the average full step in each environment.

4.3.2 Fixed Starting State

The first batch of experiments was run on the fixed-case variants of MiniGrid empty rooms. The agent was trained on each map size separately four times, starting from the plaintext case, in which the state is only padded. Then, we trained the agent again after the deterministic shuffling of the pixels of the images; afterwards, the training was repeated on the states encrypted with the ECB mode of AES. Lastly, the same has been done for the CBC version of AES.

As the plots in Figure 4.9 suggest, the performance of the agent varies depending on the size of the grid. Across all the experiments, the plaintext agent, the deterministic shuffle agent and the AES.ECB agent all perform fairly well. While the plaintext serves as a baseline for the other encryption methods, the behaviour of these two agents is explained by the determinism of their state transformation. Indeed, both in the deterministic shuffle and in ECB mode for each plaintext state there is one and only one

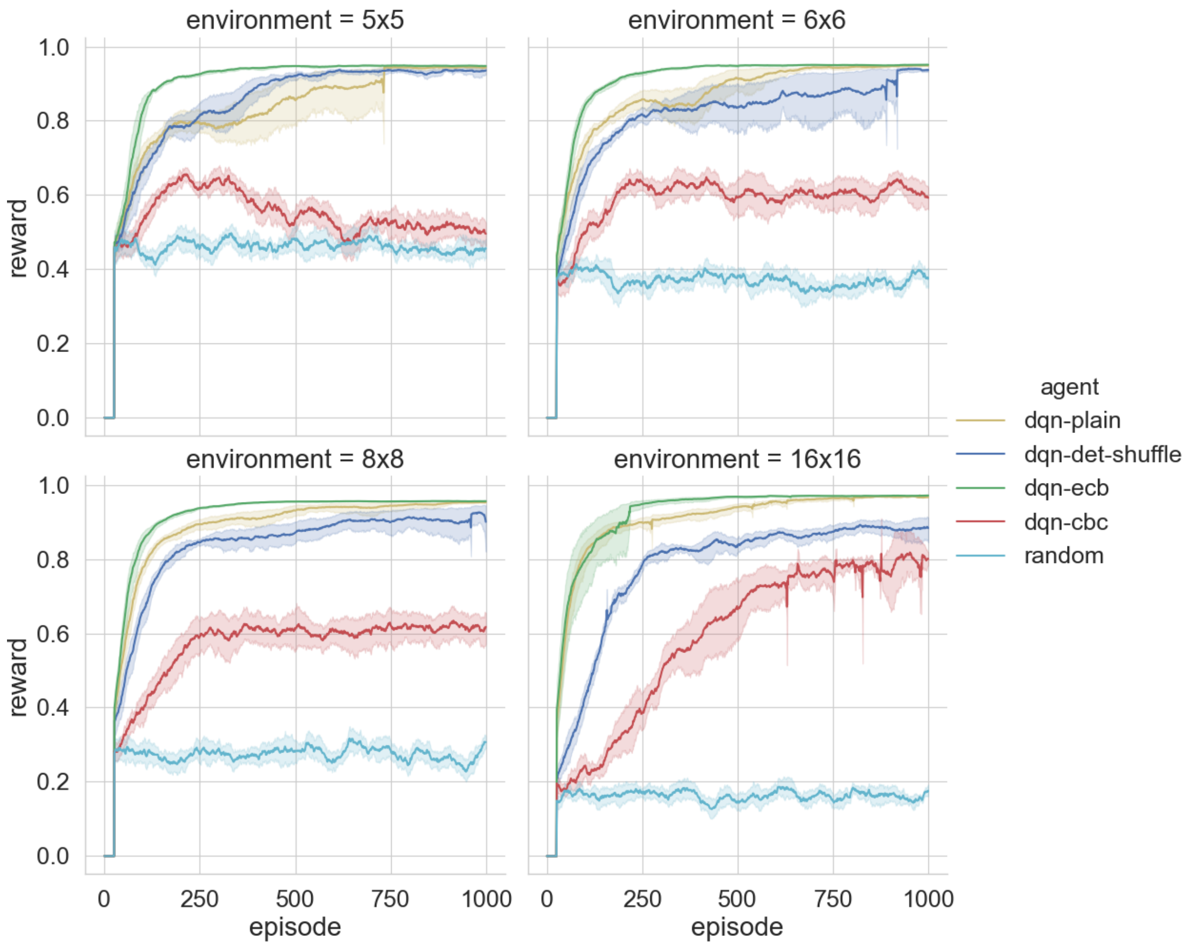


Figure 4.9: Training results on the fixed starting state environments.

Environment	None	Det. Shuffle	ECB	CBC
5x5	1.00186	1.00287	1.00285	1.00286
6x6	1.00187	1.00287	1.00287	1.00286
8x8	1.00188	1.00288	1.00288	1.00288
16x16	1.00197	1.00297	1.00299	1.00298

Table 4.3: Comparison in seconds of the duration of an average full step on the environment.

ciphertext counterpart.

However, while CBC is non-deterministic, it still shows a trend of improvement over the random baseline that only grows bigger as the size of the state increases, heavily outperforming it in the 16x16 environment. A possible explanation of this phenomenon lies in the nature of the environment: being the starting state fixed, it is likely that the agent, instead of learning a state-action mapping, learns a *sequence* of actions that leads it to the goal. Instead, the random baseline only acts randomly, and, since the reward is tied to the amount of steps done before reaching the goal, it receives worse rewards as the state gets larger. Therefore, in the next section we will repeat our experiments with the randomized variants of the environment.

Furthermore, we also have studied the impact of both key length for the AES algorithm as well as the padding type on the agent's performance. For the inspection of the padding procedure, we repeated the whole batch of experiments with both PKCS#7 and our custom padding, and compared the results. The same, except for the plaintext and deterministic shuffle cases that do not depend on a key, has been done for three different key sizes: 32, 24, and 16 bytes.

Figure 4.10 compares the behaviour of the agent trained on the two padding techniques detailed in Section 4.1.3. As we can see, the plots highlight that the specific padding technique does not induce a significant difference.

Instead, Figure 4.11 shows the comparison of the agent's training performance with different key sizes, starting from the default 256 bit (solid line), then 192 (dashed line), and 128 (dotted line). As the plots make clear, the length of the key size does not induce any difference on the ability of the agent to learn. Indeed, while the key size does increase the encrypted state space, the possible amount of states in our case is heavily bounded by the limited dimensions of the images. This makes it so that the performance of the agent is very similar regardless of the key size.

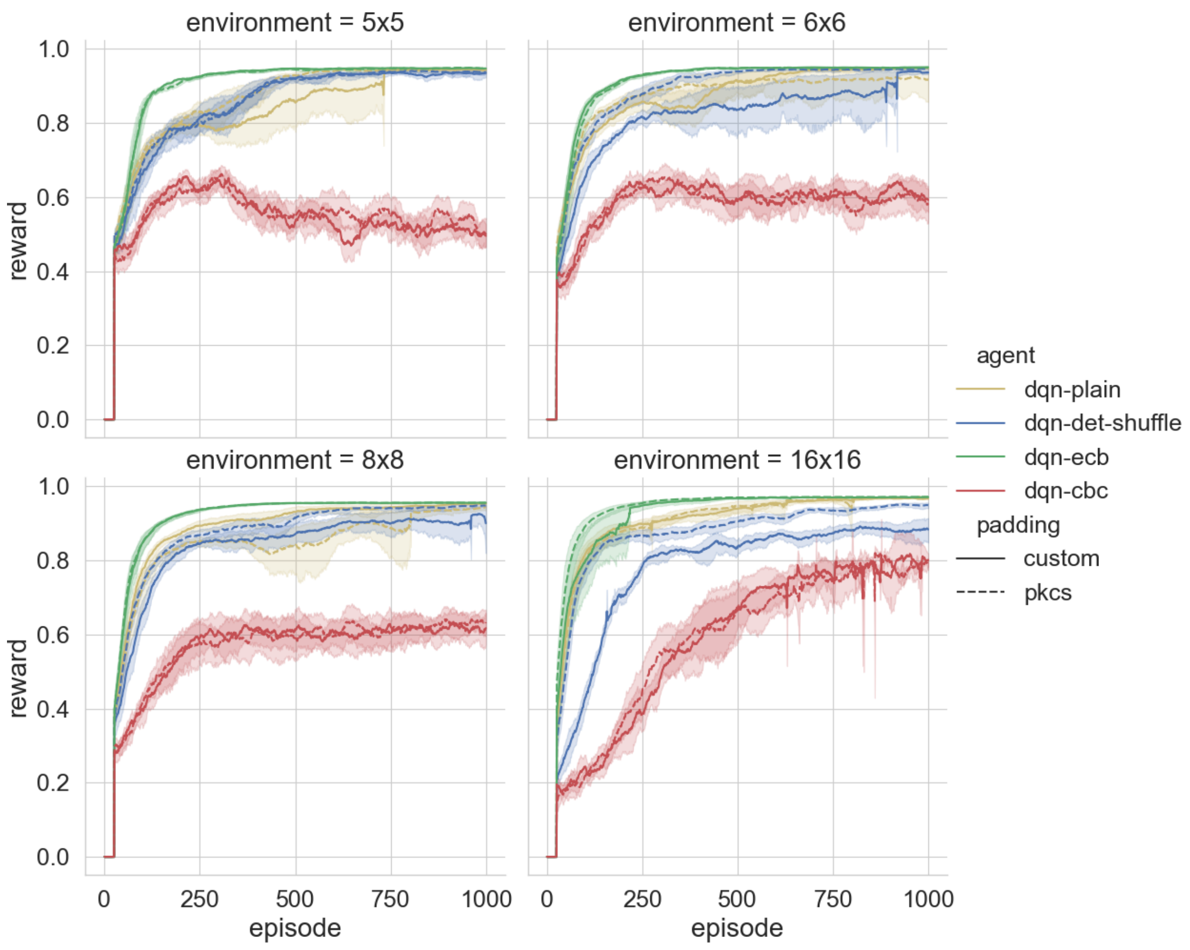


Figure 4.10: Impact of the type of padding.

4.3.3 Random Starting State

Once the analysis of the fixed starting state environments was over, we moved onto a variant of such environments. Instead of being fixed in the top left corner every time, the initial position and direction of the agent are randomized. This adds a layer of randomness to the environment and it is therefore useful to assess what impact this randomness has on the agent's learning. Because of this, alongside the agent's training loop, we also added evaluations runs, executed each $\frac{N}{10}$ frames, in which the weights of the neural network are frozen and the agent enters in `eval` mode in which it solely behaves greedily.

Therefore, the training run has been repeated with the same structure as the fixed starting state case. The agent has been trained and evaluated on all four encryption modalities (plaintext, shuffle, AES in ECB mode, and AES in CBC mode).

Figure 4.12 shows the training performance of the agent under the random starting

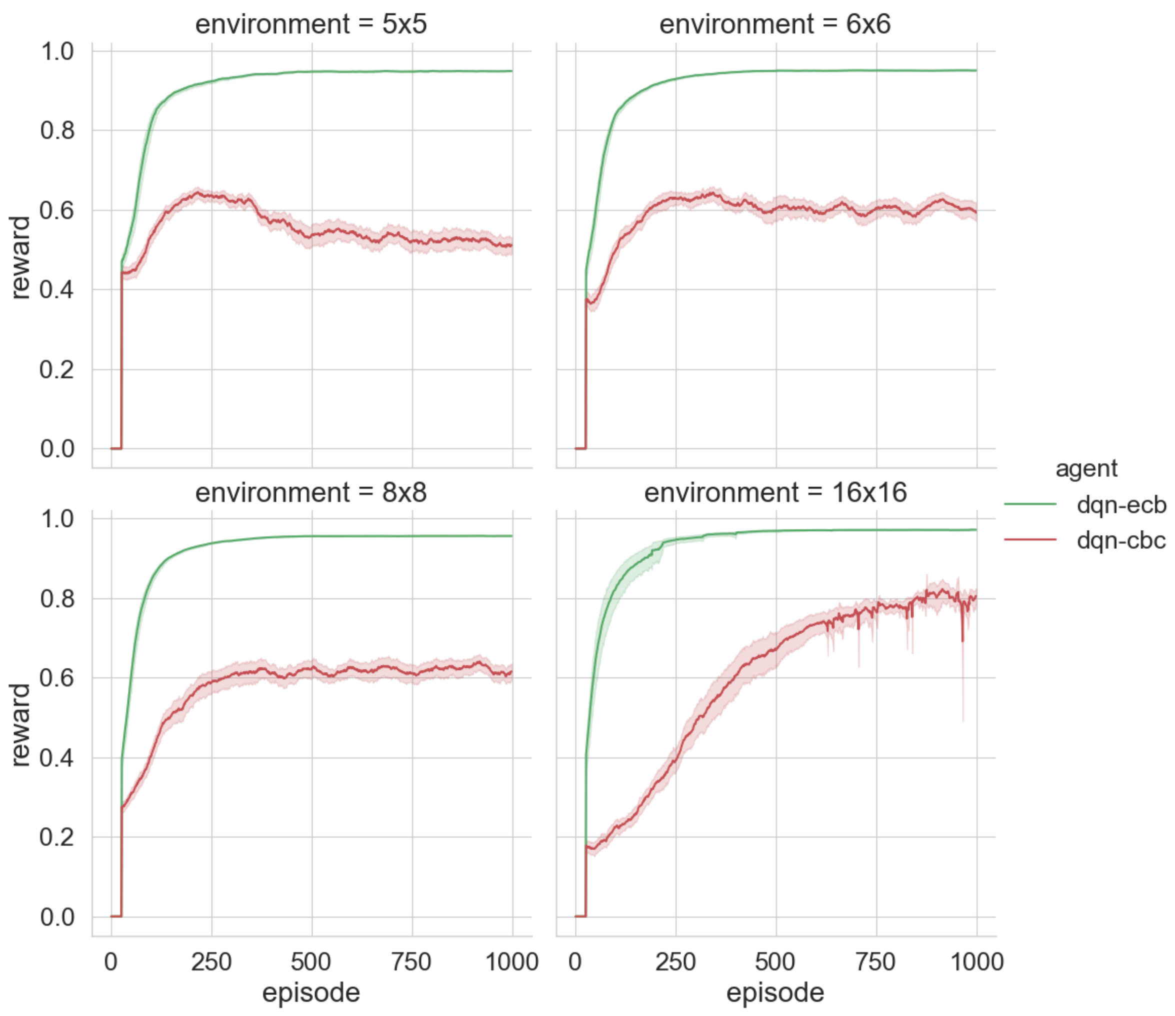


Figure 4.11: Impact of the key length in the encrypted cases.

state environments. The trend shown in Figure 4.9 persists even if the position of the agent is randomized, with the deterministic cases (plaintext, deterministic shuffle, and ECB) performing well across all the environments, and CBC gradually distancing itself from the random baseline.

The evaluations of the agent (Figure 4.13) in this randomized variant of MiniGrid highlight a similar trend, with the cases with deterministic transformations all gradually improving their performance. The only exception, however, is CBC, that despite being able to significantly surpass the random baseline during training, is not able to generalize in most of the environments. A possible explanation for this occurrence is that, since CBC is non-deterministic, that is, the same plaintext does not produce the same ciphertext, during training the agent might be biased towards some patterns in the ciphertext that do not appear during the evaluations. This bias is very harmful where there are few

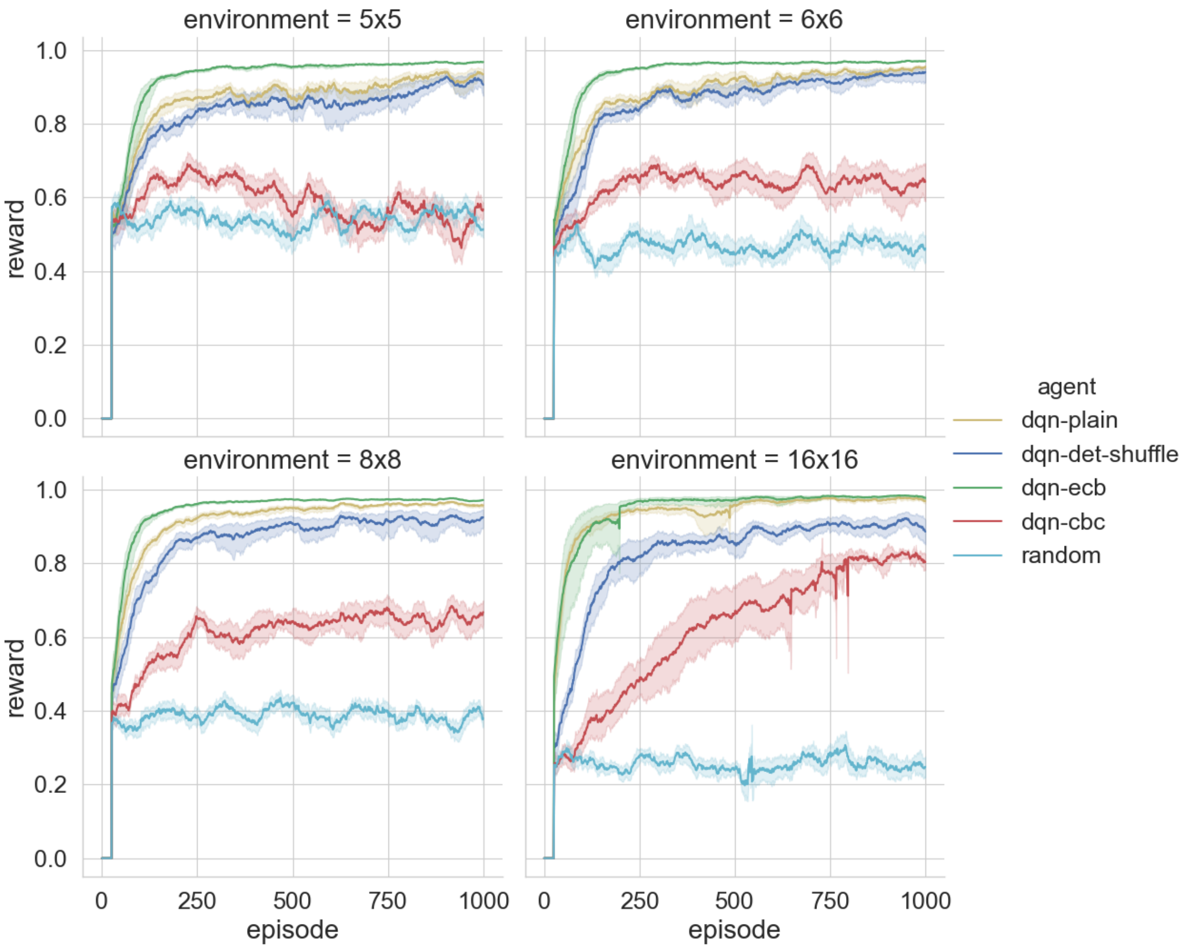


Figure 4.12: Training results in the random starting state environments

states, such as the smaller environments. As the number of states increases, the patterns learnt gradually become noise, and the agent tends to align with the random agent. This hypothesis is not entirely consistent with all the cases as, in the 8x8 environment, the CBC agents shows to perform considerably better than the random baseline (Figure 4.13, bottom left), but it is plausible that this behaviour will even out with a bigger set of random seeds.

Furthermore, ECB shows a decrease in performance for the 16x16 environment (Figure 4.13, bottom left) with respect to its behaviour in the other cases. We have yet to find an explanation, but, for this case as well, more seeds might show a more stable performance.

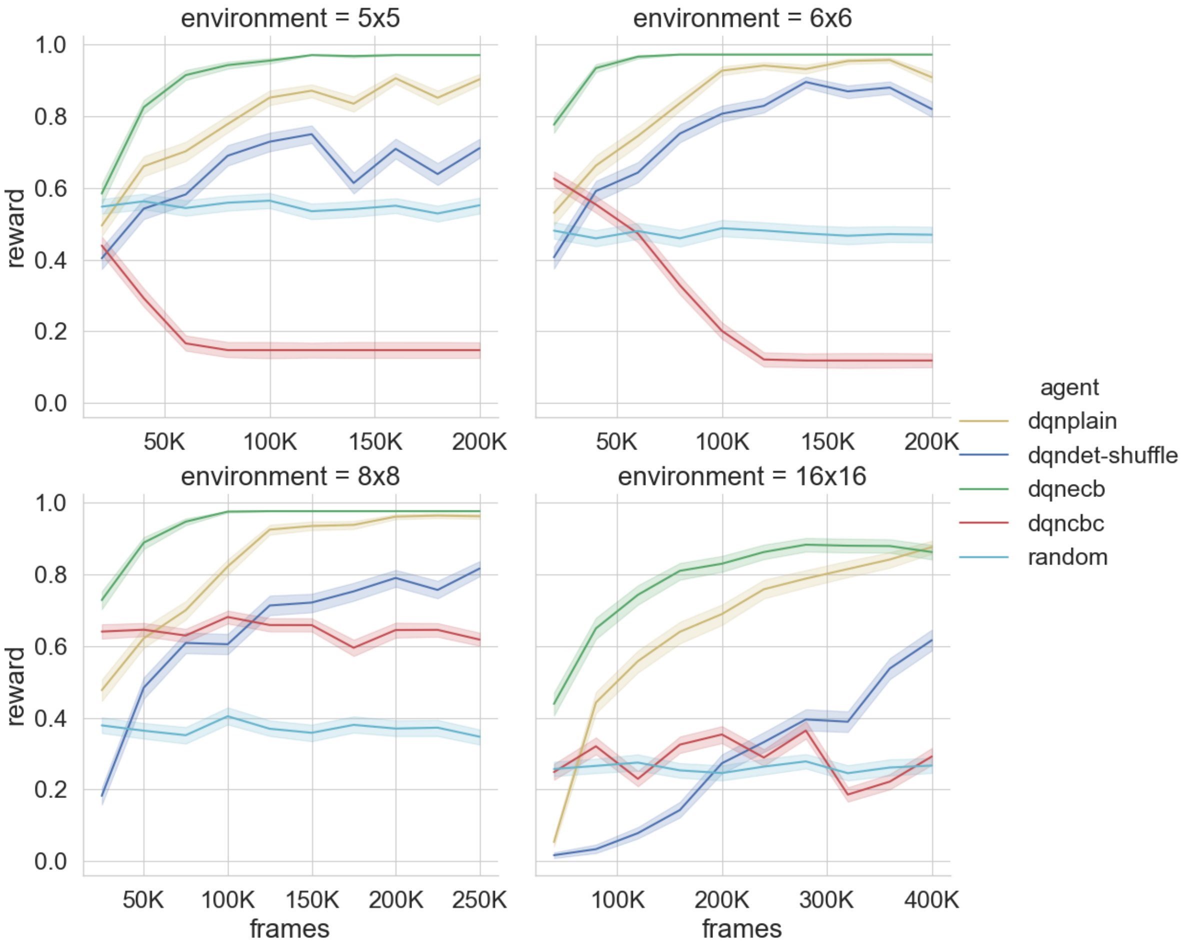


Figure 4.13: Evaluation results in the random starting state environments.

4.4 Summary

In this chapter we have conducted a case study on the application of reinforcement learning to encrypted data on a simple, gridworld environment. In the first section, we have described the experimental settings, by introducing the key aspects of the environment, the neural network architectures and the instantiation of the state-processing function $\psi(\cdot)$ that we defined in Chapter 3. Then, in the second section, we have provided the implementation details of the classes developed to carry out the algorithm. Lastly, in the third section, we went over all the experimental results obtained.

The results that we have observed during this first case study are interesting. The agent is able to learn as effectively with deterministic state transformations, and it shows some traces of learning even with non-deterministic encryption methods.

Table 4.4 shows the total amount of unique states that can be encountered in each

Environment	Plaintext	Det. shuffle	ECB	CBC
5x5	34	34	34	$3.40 \cdot 10^{38}$
6x6	62	62	62	$1.16 \cdot 10^{77}$
8x8	142	142	142	$3.94 \cdot 10^{115}$
16x16	782	782	782	$2.79 \cdot 10^{539}$

Table 4.4: Number of unique possible states in MiniGrid. In the deterministic state transformations the possible amount of unique states is bounded by the limited number of different configurations of the environment. However, being non-deterministic, CBC is not limited to those configurations and the possible unique states are all the pixel combinations: 256^{n_p}, where n_p is the total number of pixels of the corresponding image.

environment size and encryption type. In the deterministic transformations, being one-to-one mappings from unprocessed states to processed states, the state space is bounded by the possible configurations of the plaintext version, which can be computed as $n_{dir} \cdot (n_{tiles} - 1) + 2$, where n_{dir} is the amount of possible directions of the agent (in our case 4), n_{tiles} is the number of walkable tiles in the respective environment. The subtraction of 1 to the tiles and addition of 2 to the result of the multiplication is done to take into account the goal, that is on the bottom-right corner. The agent cannot start an episode being already inside the goal, so it can only get there. Hence, when in the goal, the agent can only be facing to the right or downwards.

Instead, non-deterministic transformations, such as CBC encryption, are not linked to the plaintext, therefore the state space is composed by all the possible values. In our case, since we are working with pixel intensities, the state space of CBC is given by 256^{n_p}, where n_p is the total number of pixel of the plaintext image. Indeed, a random agent exploration proves that it is possible to encounter the same encrypted state numerous times with a deterministic technique, but each state bill be unique with a non-deterministic one. However, despite the combinatorial explosion of the state space, the very limited amount of different configurations can lead to the emergence of some patterns in the ciphertext, hinting the agent towards the solution, hence obtaining relatively positive results during training, but negative results during evaluation.

In order to prove this statement, the next chapter will investigate these encryption techniques in a different environment: LunarLander.

Chapter 5

Second Case Study: Lunar Lander

Once the analysis of MiniGrid was over, we moved our focus to a completely different environment. By keeping the research questions of Chapter 3 in mind, we will now try to apply the same methodology we observed for MiniGrid in Chapter 4 to a new context and see if the agent's achievements still hold. The environment, named Lunar Lander, is one of the classic OpenAI Gym benchmarks. As opposed to the simplicity of MiniGrid, the task behind LunarLander is more difficult, as it involves the control of a lunar module in a continuous environment, hence it represents a much more complex control problem on which to test our agent, comparing the differences in performance between continuous and discrete state spaces.

This chapter will closely follow the structure of the previous one. The first section will deal with the experimental settings of this portion of work, starting from the environment, then describing the new processing and encryption pipeline $\psi(\cdot)$, the neural network architecture and the hyperparameters.

The second section will instead undertake the task of explaining the specialized class and methods developed, starting from the initial structure, for this new task, with particular focus on the new state processing techniques.

The third and last section, finally, will list and detail the experimental results obtained by our agent in the new environment.

5.1 Experimental Settings

The second part of our work focused on the application of the agent to a different environment. While MiniGrid was a discrete navigation environment, Lunar Lander is a continuous control task based on the Box2D physics engine and comes from the collection of standard benchmarks offered by OpenAI Gym.

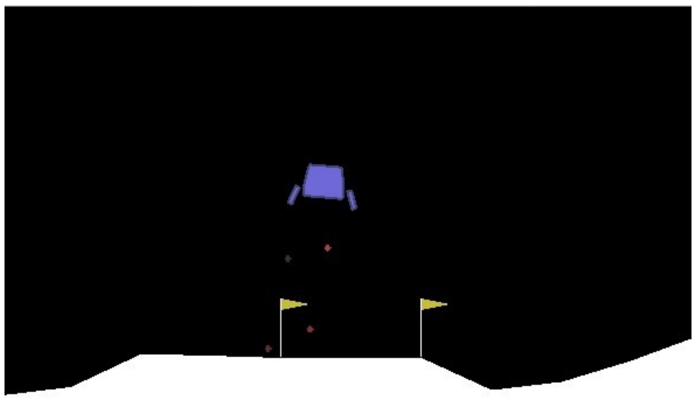


Figure 5.1: Screenshot of the LunarLander environment. The agent controls the purple lunar module and it has the task of landing it onto the landing pad on the ground, delimited by the yellow flags.

5.1.1 Lunar Lander

In this environment the agent controls a lunar module and has the task of landing it safely onto the landing pad on the ground. The landing pad is always situated at coordinates $(0, 0)$ and, as such, is the origin of the coordinate system of the environment; furthermore, it is always a flat region delimited by two yellow flags.

Observations

The default state space of LunarLander is continuous and consists in a 8-dimensional vector of unbounded floating point values,

$$(x, y, v_x, v_y, \theta, \omega, l_l, l_r) \tag{5.1}$$

where:

- x, y are the coordinates of the lander with respect to the landing pad;
- v_x, v_y are the horizontal and vertical velocities;
- θ is the angle of the lander with respect to the ground;
- ω is the angular velocity of the lander;

- l_l, l_r are boolean flags (encoded as floats) that are set to 1 if the corresponding leg of the lander is making ground contact.

Action Space

The action space of the environment is discrete and is composed by 4 actions, all related to the lander engine. The complete action space is listed in Table 5.1.

Number	**Action**
0	Do nothing
1	Fire left engine
2	Fire main engine
3	Fire right engine

Table 5.1: Action space of Lunar Lander.

Reward

The reward from moving from the top of the screen, which is the initial state, towards the landing pad is in the interval $[100, 140]$, depending on the distance of the lander with respect to the pad. If the lander crashes or lands, a bonus -100 or $+100$ is assigned respectively and the episode ends. The episode also ends if the lander moves too far away from the pad. A further $+10$ bonus is added for each of the legs that make ground contact. Fuel is infinite, but for each frame passed firing the main engine, a -0.3 penalty is assigned to the overall score.

5.1.2 State Processing

While with MiniGrid we focused on images, for Lunar Lander we will, instead, leverage the default `Box` object returned by OpenAI Gym. The state is composed by 8 features, each expressed by a 32-bit floating point number. However, since after the encryption the states become essentially noise, the byte-to-float conversion prior to the neural network led to very high or very low numbers, hence causing an overflow in the forward pass of the neural network resulting in `NaN` losses and q-values.

For this reason, the state preprocessing function $\phi(\cdot)$ consists in the discretization and binning of the continuous components of the state. By studying the environment we have found some empirical ranges in which the variables are likely to be observed during the episodes, and used them to perform a 256-bin discretization of the state. The ranges used for the discretization are listed in Table 5.2.

Variable	Interval
x	$[-1.5, 1.5]$
y	$[0, 1.5]$
v_x	$[-5.0, 5.0]$
v_y	$[-5.0, 5.0]$
θ	$[-6.0, 6.0]$
ω	$[-5.0, 5.0]$

Table 5.2: Discretization intervals of the continuous variables of the state.

5.1.3 Encryption

After the preprocessing function $\phi(\cdot)$, the next component is the encryption primitive $Enc(\cdot)$. As in the previous case study on MiniGrid, the primitives involved in the transformations are the same, with the exception of the deterministic shuffling. Indeed, since we are no longer working with images, and as such pixels, but now the state is composed by continuous, descriptive variables, there is no spatial correlation between the values of the state, and deterministic shuffle would only change their order within the array itself with no impact on the agent; hence, we decided to not include it in the following experiments.

However, the method followed for the training of the agent remains unchanged. It is first trained on the plaintext version, then on the ECB mode of AES, and, lastly, on the CBC mode of AES, with the keys randomly generated before each run, and, for the CBC case, the IV is randomly reinitialized before each encryption step.

Padding

Given that the encryption primitives are based on AES, which is a block cipher, there still is the need of making the state reach a multiple of the nominal 16 bytes of the block size. Since the preprocessing function $\phi(\cdot)$ discretizes the state in 256 bins, the values of the processed states can be represented by 8-bit integers, therefore occupying a single byte each.

Hence, the full state size after preprocessing is stored in a 8-byte long array. The padding step before the encryption has the task of adding bytes to this representation to reach the minimal length of 16. However, as mentioned above, the variables of the state do not have any correlation between each other in terms of their spatial distribution within the array itself; because of this, the quickest and more reliable way to pad this data is to employ the standard PKCS#7 padding, that will add $k - (l \mod k)$ bytes, with value $k - (l \mod k)$, where k is the block size and l is the length of the message, at the trailing end of the state. In our case, the result of this operation will be a padding tail of 8 bytes, each with the value 8.

5.1.4 Neural Network Architecture

For this case, since there is no spatial information in the state of the environment, the ANN features a fully connected architecture. This means that each unit in each neural network layer is connected to all the units of the subsequent layer. The first layer, also known as input layer, consists of 16 units, which corresponds to the size of the array of features that are being given as state. Afterwards, the data is passed forward through three 128-unit hidden layers and, lastly, a 4-unit output layer, where the number of nodes is equal to the cardinality of the set of actions $|\mathcal{A}(S_t)|$. The activation functions for all the hidden layers are ReLUs.

The architecture has been found empirically, and its big size is contrasting to the relatively small scale ANNs that are typically employed for the solution of similar tasks. However, the reason for it lies in the discretization step, that makes the control of the agent much more coarse-grained with respect to its non-processed, continuous counterpart, hence making the task harder. Indeed, we have found that, with fewer hidden layers, the agent was not able to learn at all. Its performance slowly improved with more complex network, up to the three hidden layers with 128 units each used in these experiments.

5.1.5 Hyperparameters

The set of hyperparameters used for the execution of the experiments is only one and it is kept consistent across encryption types to provide a better comparison. The full set of values is listed in Table 5.3

5.2 Implementation

Considering the experimental setup we have just reviewed, this part of the project that focuses on Lunar Lander reuses all of the components we have introduced for the first case study in Section 4.2, as thye were designed to be general purpose. The only exception is the environment wrapper `LunarWrapper`, which, for this environment, features some minor changes.

Parameter	**Value**
N steps	$5 \cdot 10^5$
Learning rate	$5 \cdot 10^{-4}$
Memory size D	$2 \cdot 10^5$
Batch size	128
ϵ-decay	0.99
ϵ-max	1
ϵ-min	0.01
γ	0.99
C	200

Table 5.3: Hyperparameters for LunarLander.

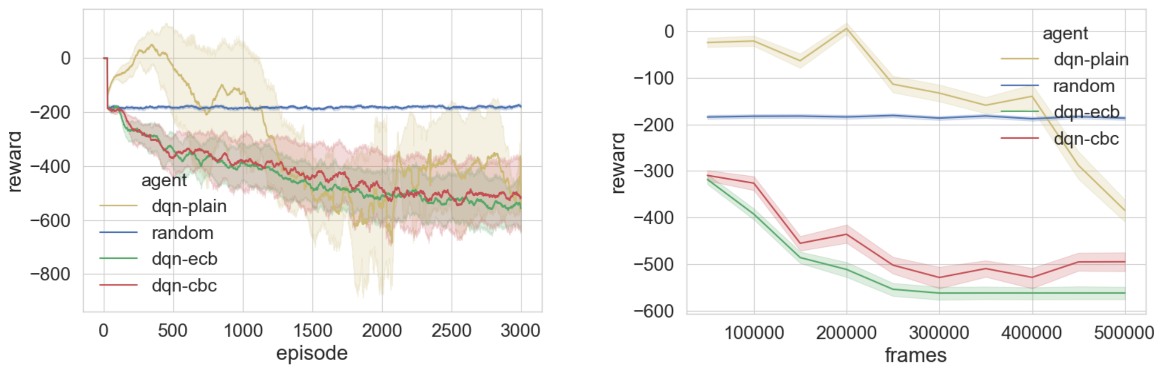
In particular, since we are no longer working with images, but with monodimensional arrays of values, the state transformations are not operated by a standalone class, but are carried out by the specialized wrapper itself on the go with a new method called `discretize`. This function, called within the `transform` method, takes as input a `variable`, an `interval` and the total amount of desired bins `n_bins`, and returns the integer value of the binned variable, that is, the index of the bin under which the measure falls into. The discretization operated by `discretize` is the realization of the preprocessing function $\phi(\cdot)$ outlined in Section 5.1.2, and it plays a key part in preventing the overflowing of the neural network during forward passes.

However, the other components of the project remain unchanged, hence their implementation details can be found in Section 4.2.

5.3 Experimental Results

In this section we will evaluate our agent on the LunarLander environment. The purpose of this evaluation is to compare the results obtained by the agent in the previous case study and to see if the agent is able to obtain similar results.

The experiments will train the agent on the LunarLander environment, under the state transformation function $\psi(\cdot)$ illustrated in the sections above. The training is repeated, separately, with the plaintext case, then with AES in ECB mode, and lastly with AES in CBC mode. Furthermore, after every $\frac{N}{10}$ frames, the weights of the agent's neural network will be frozen and an evaluation will start, in which the agent will behave purely greedily for 100 episodes, to assess the current state of the learning. Each train-evaluation loop is then repeated 10 times, with different seeds, to ensure some statistical stability and to favor repeatability.



(a) Training (b) Evaluation

Figure 5.2: Performance of the agent in the LunarLander environment.

While not ever converging, Figure 5.2a shows that the plaintext version is able to start improving its behaviour during the first part of the training, considerably outperforming the random agent before drastically diverging. On the other hand, the encrypted versions are not able to learn at all and perform worse than the random baseline.

Regarding the plaintext case, it is likely that with a deeper search of the hyperarameter space and of the discretization space the agent will improve its performance and stabilize its behaviour. For example, the total amount of bins of the discretization step can be increased in order to give the agent better fine-grained control. However, the same cannot be stated for the encrypted versions, given their very large state space.

5.4 Summary

In this chapter we have conducted a second case study on the application of reinforcement learning to encrypted states on a more complex environment. In the first section, we have defined the experimental setup by detailing the core aspects of the environment, the state processing function $\psi(\cdot)$ and the network architecture. In the second part we have focused on the implementation of the new state processing pipeline, with all the other components of the project being untouched given their general-purpose design. Lastly, in the third section, we have listed the experimental results on our study.

The results obtained in this second environment are consistent with the statement made in Section 4.4.

Indeed, as opposed to MiniGrid, the state space in this case is exponentially larger for the plaintext and ECB versions. The original state of Lunar Lander (see Section 5.1.1)

Plaintext	**ECB**	**CBC**
$1.12 \cdot 10^{15}$	$1.12 \cdot 10^{15}$	$3.40 \cdot 10^{38}$

Table 5.4: Number of unique possible states in LunarLander. In this case, the state space of the plaintext and ECB versions is much larger: it is given by all the possible bin combinations of the state. The state space of CBC, instead, is smaller, since the size of the state is smaller with respect to MiniGrid images.

is composed by six continuous variables and two boolean flags. After the processing function $\phi(\cdot)$, the continuous parts of the state are discretized in 256 bins, meaning that each variable can now be one of 256 distinct values. Instead, the boolean flags are already discrete, and can only be one of two values, either 0 or 1. Therefore, the amount of possible unique states is now given by $256^6 \cdot 2 \cdot 2$ in the deterministic cases (6 binned variables and 2 boolean flags). For CBC, instead, since its non-determinism makes it unrelated to the configurations of the state, the state space is given by all the possible combinations of values. After $\phi(\cdot)$, each variable is represented by an 8-bit integer, which represents a number within $[0, 256[$, and, after padding, there are 16 bytes in the state, hence 256^{16}.

The poor performance of the agent in the encrypted versions might be explained by the correlation of the variables of the state. Despite being deterministic, ECB can break these correlations, or create new ones that are harder to identify. Furthermore, even with a smaller state space in CBC with respect to MiniGrid, the agent does not show any sign of improvement during training. This is likely due to the environment: while MiniGrid had very few possible configurations, the very large state spaces of the plaintext show that it is not possible for the agent to reliably find a pattern in the noisy data.

Chapter 6

Conclusions

6.1 Contributions

The purpose of this work was to assess the feasibility of the approach of applying reinforcement learning to encrypted states. We have therefore identified three key research questions in Chapter 3 from which the project started. In order to provide an answer to these questions, we have:

- Extended the Markov Decision Process framework so as to include encrypted states, by revisiting the core concepts of reinforcement learning under this assumption;
- Applied this revisitation to the DQN algorithm;
- Defined a set of functions for the preprocessing in order to prepare the states for the encryption step;
- Defined multiple padding techniques and encryption primitives to carry out the experiments;
- Presented two case studies in which we have applied our algorithm to different environments and studied the impact of the transformations on the ability to learn of the agent.

Overall, from the studies directed on our application of the agent to encrypted states, it emerged that:

- It is feasible to apply reinforcement learning on encrypted data in certain scenarios while still maintaining a relatively high performance of the agent. These scenarios, specifically, are those whose state spaces are small, as demonstrated in Section 4.3.

In particular, deterministic encryption mechanisms in these scenarios do not hinder the agent's learning, and also non-deterministic procedures show signs of learning (Section 4.3.2). However, adding randomness to these environments hurts the generalization ability of the agent in case on non-deterministic encryption methods (Section 4.3.3).

- In more complex scenarios where the state spaces are large, instead, the application of the encryption step to the state leads to a complete collapse of the performance of the agent and to the divergence of the algortihms (Section 5.3).
- The encryption overhead for standard block ciphers and block modes is negligible with respect to the rest of the standard reinforcement learning loop, whereas homomorphic encryption turned out to be too expensive in this context, as demonstrated in Section 4.3.1.
- The length of the key for the encryption primitive has no impact as long as the space of possible ciphertexts is larger than the space of unique states, and the specific type of padding employed also makes little difference, as shown in Section 4.3.2.

6.2 Limitations

The work conducted in this dissertation, however, presents some limitations, especially regarding time limits.

- The amount of random seeds used for the experiments does not guarantee stability, hence, repeating them with more seeds could improve the statistical confidence of the results.
- Particularly for the second case study, time constraints have prevented a deeper exploration of state processing schemes and hyperparameters.

6.3 Future Work

Regardless of the results achieved, however, the research on the application of reinforcement learning to encrypted states is far from over, and we can identify several direction in which to steer future further investigations, all interesting in perspective:

- Some other state-of-the-art reinforcement learning techniques might give more insights on the matter. For example, policy gradient methods, which do no work by the estimation on q-values, might behave differently in presence of encrypted states.

- In complex environments with large state spaces there is the need of exploring different encryption mechanisms in order to preserve the reward structure so that learning can still take place.

6.4 Implications

The results shown above are indeed interesting, but the mechanics of reinforcement learning over encrypted data are not well understood yet. While a successful application of these techniques would lead the way to numerous use cases and scenarios and allow us to decouple the agent from the environment, and despite the experiments showing that our approach tends to work consistently well with limited state spaces and small problems, large state spaces tend to prevent the agent from learning. Indeed, more investigation is still needed to better understand the underlying causes that led to the emersion of such behavior before this method can be reliably used in the real world.

Bibliography

[1] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with Deep Neural Networks and Tree Search. *Nature*, 529(7587):484–489, January 2016.

[2] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550:354–, October 2017.

[3] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144, 2018.

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari With Deep Reinforcement Learning. In *NIPS Deep Learning Workshop*. 2013.

[5] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 02 2015.

[6] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Daniel Horgan, Bilal Piot, Mohammad Gheshlaghi Azar, and David Silver. Rainbow: Combining Improvements in Deep Reinforcement Learning. *CoRR*, abs/1710.02298, 2017.

[7] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *CoRR*, abs/1707.06347, 2017.

[8] Thore Graepel, Kristin Lauter, and Michael Naehrig. ML Confidential: Machine Learning on Encrypted Data. In *Proceedings of the 15th International Conference on Information Security and Cryptology*, ICISC'12, page 1–21, Berlin, Heidelberg, 2012. Springer-Verlag.

[9] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy. Technical Report MSR-TR-2016-3, February 2016.

[10] Hervé Chabanne, Amaury de Wargny, Jonathan Milgram, Constance Morel, and Emmanuel Prouff. Privacy-Preserving Classification on Deep Neural Network. Cryptology ePrint Archive, Report 2017/035, 2017. `https://eprint.iacr.org/2017/035`.

[11] Ehsan Hesamifard, Hassan Takabi, and Mehdi Ghasemi. CryptoDL: Deep Neural Networks over Encrypted Data. *CoRR*, abs/1711.05189, 2017.

[12] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. *CoRR*, abs/1611.03530, 2016.

[13] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym, 2016.

[14] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* The MIT Press, Cambridge, MA, USA, 2018.

[15] Edward L. Thorndike. Animal Intelligence. 1911.

[16] Richard E. Bellman. *Dynamic Programming.* Princeton University Press, Princeton, NJ, 1957.

[17] Richard E. Bellman. A Markovian Decision Process. *Journal of Mathematics and Mechanics*, 6(5):679–684, 1957.

[18] Gerald Tesauro. Practical issues in temporal difference learning. *Machine Learning*, 8(3-4):257–277, 1992.

[19] Gerald Tesauro. TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6(2):215–219, 1994.

[20] Gerald Tesauro. Temporal difference lerning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[21] Gerald Tesauro. Programming backgammon using self-teaching neural nets. *Artificial Intelligence*, 134(1-2):181–199, 2002.

[22] Rémi Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In *Proceedings of the 5th International Conference on Computers and Games*, CG'06, page 72–83, Berlin, Heidelberg, 2006. Springer-Verlag.

[23] Levente Kocsis and Csaba Szepesvári. Bandit Based Monte-Carlo Planning. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *Machine Learning: ECML 2006*, pages 282–293, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[24] Christopher John Cornish Hellaby Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, May 1989.

[25] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016.

[26] 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, jun 2013.

[27] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. Imagenet classification with deep convolutional neural networks. *Neural Information Processing Systems*, 25, 01 2012.

[28] Pierre Sermanet, Koray Kavukcuoglu, Soumith Chintala, and Yann LeCun. Pedestrian Detection with Unsupervised Multi-Stage Feature Learning. *CoRR*, abs/1212.0142, 2012.

[29] Li Deng, Alex Acero, George Dahl, and Dong Yu. Context-Dependent Pre-trained Deep Neural Networks for Large Vocabulary Speech Recognition. In *IEEE Transactions on Audio, Speech, and Language Processing*, volume 20, pages 30–42, January 2012. IEEE SPS 2013 best paper award.

[30] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6645–6649, 2013.

[31] Martin Riedmiller. Neural Fitted Q Iteration – First Experiences with a Data Efficient Neural Reinforcement Learning Method. In João Gama, Rui Camacho, Pavel B. Brazdil, Alípio Mário Jorge, and Luís Torgo, editors, *Machine Learning: ECML 2005*, pages 317–328, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[32] M. Riedmiller and H. Braun. A direct adaptive method for faster backpropagation learning: the RPROP algorithm. In *IEEE International Conference on Neural Networks*, pages 586–591 vol.1, 1993.

[33] Long-Ji Lin. *Reinforcement Learning for Robots Using Neural Networks.* PhD thesis, USA, 1992. UMI Order No. GAX93-22750.

[34] OpenAI Gym. `https://gym.openai.com/`.

[35] OpenAI Gym documentation. `https://gym.openai.com/docs`.

[36] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[37] Maxime Chevalier-Boisvert, Lucas Willems, and Suman Pal. Minimalistic Gridworld Environment for OpenAI Gym. `https://github.com/maximecb/gym-minigrid`, 2018.

[38] Mihir Bellare and Phillip Rogway. *Introduction to Modern Cryptography.* University of California, 2005.

[39] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography.* Chapman and Hall/CRC Press, 2007.

[40] Burt Kaliski. PKCS #7: Cryptographic Message Syntax Version 1.5. RFC 2315, March 1998.

[41] Larry Ewing. `https://isc.tamu.edu/~lewing/linux/`. Created with The GIMP.

[42] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002.

[43] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. `crypto.stanford.edu/craig`.

[44] R L Rivest, L Adleman, and M L Dertouzos. On Data Banks and Privacy Homomorphisms. *Foundations of Secure Computation, Academia Press*, pages 169–179, 1978.

[45] Nigel P. Smart and Frederik Vercauteren. Fully Homomorphic Encryption with Relatively Small Key and Ciphertext Sizes. In *Public Key Cryptography - PKC 2010, 13th International Conference on Practice and Theory in Public Key Cryptography, Paris, France, May 26-28, 2010. Proceedings*, volume 6056 of *Lecture Notes in Computer Science*, pages 420–443. Springer, 2010.

[46] Zvika Brakerski and Vinod Vaikuntanathan. Fully Homomorphic Encryption from Ring-LWE and Security for Key Dependent Messages. In Phillip Rogaway, editor, *CRYPTO*, volume 6841 of *Lecture Notes in Computer Science*, pages 505–524. Springer, 2011.

[47] Craig Gentry, Shai Halevi, and Nigel P. Smart. Better Bootstrapping in Fully Homomorphic Encryption. In Marc Fischlin, Johannes Buchmann, and Mark Manulis, editors, *Public Key Cryptography – PKC 2012*, pages 1–16, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[48] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic Evaluation of the AES Circuit. In *Advances in Cryptology - Crypto 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 850–867. Springer, 2012.

[49] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) Fully Homomorphic Encryption without Bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, page 309–325, New York, NY, USA, 2012. Association for Computing Machinery.

[50] Zvika Brakerski. Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances*

in Cryptology – CRYPTO 2012, pages 868–886, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[51] Microsoft SEAL (release 3.6). `https://github.com/Microsoft/SEAL`, November 2020. Microsoft Research, Redmond, WA.

[52] Junfeng Fan and F. Vercauteren. Somewhat Practical Fully Homomorphic Encryption. *IACR Cryptol. ePrint Arch.*, 2012:144, 2012.

[53] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic Encryption for Arithmetic of Approximate Numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017*, pages 409–437, Cham, 2017. Springer International Publishing.

[54] R. A. Fisher. The Use of Multiple Measurements in Taxonomic Problems. *Annals of Eugenics*, 7(7):179–188, 1936.

[55] A. Frank and A. Asuncion. UCI machine learning repository. `http://archive.ics.uci.edu/ml`, 2010.

[56] Joppe Bos, Kristin Lauter, and Michael Naehrig. Private Predictive Analysis on Encrypted Medical Data. Technical Report MSR-TR-2013-81, September 2013.

[57] Raphael Bost, Raluca Popa, Stephen Tu, and Shafi Goldwasser. Machine Learning Classification over Encrypted Data. 01 2015.

[58] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[59] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, Lille, France, 07–09 Jul 2015. PMLR.

[60] Alex Krizhevsky. Learning multiple layers of features from tiny images. Master's thesis, University of Toronto, Department of Computer Science, 2009.

[61] L. T. Phong, Y. Aono, T. Hayashi, L. Wang, and S. Moriai. Privacy-Preserving Deep Learning via Additively Homomorphic Encryption. *IEEE Transactions on Information Forensics and Security*, 13(5):1333–1345, 2018.

[62] Haoran Tang, Rein Houthooft, Davis Foote, Adam Stooke, Xi Chen, Yan Duan, John Schulman, Filip De Turck, and Pieter Abbeel. #Exploration: A Study of Count-Based Exploration for Deep Reinforcement Learning. *CoRR*, abs/1611.04717, 2016.

[63] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking Deep Reinforcement Learning for Continuous Control. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1329–1338, New York, New York, USA, 20–22 Jun 2016. PMLR.

[64] Connor Shorten and Taghi Khoshgoftaar. A survey on Image Data Augmentation for Deep Learning. *Journal of Big Data*, 6, 07 2019.

[65] Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324, 1998.

[66] Karl Cobbe, Oleg Klimov, Chris Hesse, Taehoon Kim, and John Schulman. Quantifying Generalization in Reinforcement Learning. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 1282–1289. PMLR, 09–15 Jun 2019.

[67] Kimin Lee, Kibok Lee, Jinwoo Shin, and Honglak Lee. Network Randomization: A Simple Technique for Generalization in Deep Reinforcement Learning, 2020.

[68] Denis Yarats, Ilya Kostrikov, and Rob Fergus. Image Augmentation Is All You Need: Regularizing Deep Reinforcement Learning from Pixels. In *International Conference on Learning Representations*, 2021.

[69] Michael Laskin, Kimin Lee, Adam Stooke, Lerrel Pinto, Pieter Abbeel, and Aravind Srinivas. Reinforcement Learning with Augmented Data, 2020.

[70] Alexander Lenail. NN-SVG. `http://alexlenail.me/NN-SVG/index.html`.

[71] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[72] OpenCV colour conversion. `https://docs.opencv.org/master/de/d25/imgproc_color_conversions.html#color_convert_rgb_gray`.

[73] Diederik Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *International Conference on Learning Representations*, 12 2014.

[74] Aurelien Geron. *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems.* O'Reilly Media, Sebastopol, CA, 2017.

[75] Alberto Ibarrondo, Melek Onen, and Laurent Gomez. Pyfhel: Python for homomorphic encryption libraries. `https://github.com/ibarrond/Pyfhel`, 2017.

Acknowledgements

With this work ends a long, yet incredibly rewarding, journey that I am so glad I started. This dissertation, in a way, represents the fitting conclusion of this chapter of my life. So, I would like to spend the last few words of this document to mention all the people that played a role in this achievement, both during my dissertation and during the period of my studies.

First of all, I would like to thank prof. Mirco Musolesi and prof. Rebecca Montanari for making this project possible in the first place. You allowed me to work on what I consider two of my main interests simultaneously, and introduced me to the world of research, enabling a period of incredible personal and professional growth that I probably would not have taken myself.

Secondly, I would like to thank Victor and Alessandro for being able to deal with my (multiple) mental breakdowns with a smile, and for always assisting and supporting me in the greatest way possible. You are good friends to me before supervisors, and I am glad you were with me during this journey.

I would also like to spend some words on my closest friends that have been with me throughout these grim times. Thank you, Riccardo, for the Castelline nights; Alessandro, for the still-to-be-done photography plans; Nicola, for the endless stream of memes; Federico, Giada, Giovanni, Foglia, Luca, Sula, for turning many of these dull lockdown nights into good times. All of you managed to make this period of distancing feel much warmer.

Lastly, but most importantly, I would like to thank my girlfriend Paola. These times have been incredibly hard, I suffered the loss of a loved one, I faced familiar problems, but you were always there by my side to support me, always pushing out the best version of me in every situation. Without you I never would have made it this far. I love you!

Thanks to everyone, from the bottom of my heart.