EURECOM
Sophia Antipolis

A first introduction to reinforcement learning and openAI Gym environments

Advisor: Prof. Pietro Michiardi

Project by Daniele Reda

Spring 2017

Contents

Chapter 1

Introduction and Motivations

This document has been created as final report for the semester project of Eurecom Spring 2017 semester. The project has been done with supervision of Professor Pietro Michiardi whom I infinitely thank for his patience and help in following a correct path in the discovery of the world of reinforcement learning.

1.1 Why reinforcement learning?

The question that everyone might ask is: *Why reinforcement learning?* Well, there are many answers that one could give. First of all, it's getting popular and it's going to become the new hype around machine learning soon, so it's better to be prepared for this new wave around this field. Second, reinforcement learning represents the future and the world itself. Try to imagine a baby that is learning his first steps. He tries to stand up but falls and so he tries again. He tries again, again and again until he finally manages to stand up. And then from there he tries to lift one foot and does a step but he falls again. And so he will stand up again and try again until he will finally manage to make some steps. The same happens in nature.

We humans, initially, were undeveloped and at a primal stage. With time, evolution comes into act and during the years we developed new abilities and evolved according to our needs. And reinforcement learning is exactly this. It's the model of learning and evolving only from previous experiences in order to improve. Reinforcement learning tries to reproduce nature and aims to apply human learning abilities to a machine. It's the *future.*

1.2 The journey to this report

The initial goal of this project was to play with openAI's Gym environments, reproduce results of the agent in openAI's gitHub page and use that as a starter point to make tests and solve other environments. Studying the model and playing with it, I realized that the methodology and the theory behind reinforcement learning were absolutely unknown arguments to me. This is why, together with the project supervisor, it has been decided to study what reinforcement learning was, how it worked and what were its fundamentals. This report wants to be a written handout of everything it has been learned in these last months of research.

1.3 Report outline

The structure of this report is the following.

In chapter 2, an overview of Markov Decision Processes is given, underlying basis of reinforcement learning.

In chapter 3, it is presented the concept of reinforcement learning, focusing on the different functions and main components. The last part of this chapter is dedicated to present the main algorithms studied during the semester project period.

In chapter 4, openAI Gym and openAI Universe are introduced and the work done with these environments is reported.

In chapter 5 some conclusions and future works are reported while chapter 6, last one, wants to be a basket full of links, papers, books and other type of interesting materials to study or give a look to get to know more about reinforcement learning.

Chapter 2

Markov Decision Processes

Reinforcement learning refers to both a learning problem and a subfield of machine learning. A typical reinforcement learning setting is depicted in figure 2.1 and, as you can see it, shows a controller and a system continuously connected with each other where the controller receives the state of the system and the reward of the previous iteration and outputs a new action to send to the system. In response to this action, the system makes a transition and this cycle is repeated indefinitely. The main goal of reinforcement learning is to actually learn the problem and manage to control the system in order to maximize the reward. Problems with these characteristics are defined as Markov Decision Processes (MDPs) and the purpose of this chapter is to introduce MDPs as they describe the environment for reinforcement learning.

2.1 Problem Definition

A Markov Decision Process is defined as a tuple, a triple, a 4-elements tuple or sometimes as a 5-elements tuple, depending on the specifications. In any case, here are the most important elements that define a MDP:

- A **state space** $\mathcal{S}$ which is a countable non-empty set of states $\int$;

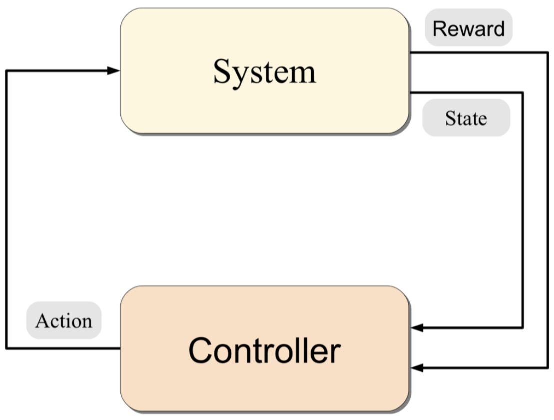

Figure 2.1: A typical reinforcement learning setting

- An **action space** $\mathcal{A}$ which defines a countable non-empty set of actions a, or $a(s)$ if the set of actions is function of the state where we are in;
- A **transition probability distribution** or **transition model** $\mathcal{P}$ that defines the probability of moving from state s to s' following action a. This probability is defined as $\mathbb{P}(s_{t+1} = s' | s_t = s, a_t = a)$;
- An **immediate reward function** $\mathcal{R}$ which gives the reward received when a is chosen in state s and can be written as $\mathcal{R}(s)$ or $\mathcal{R}(s, a)$ or $\mathcal{R}(s, a, s')$;
- and a **discount factor** γ that will be described later.

First of all, why are MDPs called Markov Decision Processes? Now that we have the definition, we can explain it. MDPs are a tool for modeling sequential decision-making problems where a decision maker interacts with a system in a sequential way [14]. With this definition, we already explained the *decision* and the *process* part of the name; but why *markovian*?

Well, the **Markovian property** is a property stating that the conditional probability of the future depends only on the present state and not on past states or, in other words, that the current state where whatever person, computer, system you are trying to model is, is sufficient to decide on future actions and spaces without having to look backwards in time. We can see this in the definition of the transition probability distribution where the conditional probability of s_{t+1} is conditioned only by s_t and not by s_{t-1}.

2.2 An example

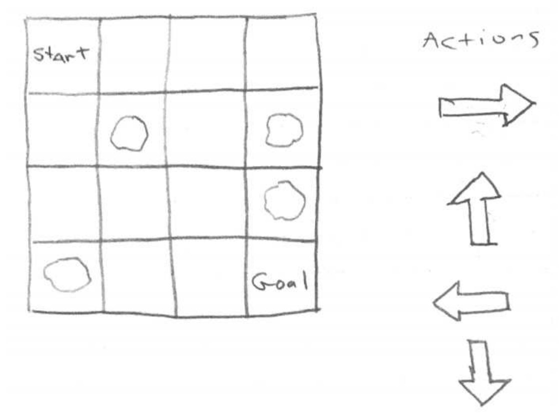

Figure 2.2: Frozen Lake. An example of a MDP game [13]

MDPs can be better explained with the game of Frozen Lake depicted in figure 2.2. The game's story is that a guy was playing frisbee but with a wrong shot his frisbee got stuck in the middle of a frozen lake and it has to be retrieved. The ice is very slippery and so every time a step is made, you have some probability of ending up in a different place. The grid system shows the starting cell, the final cell and some holes where you shouldn't end up.

In this game, the state is the complete grid system with the place where you are marked. There are 4×4 possible states (all possible cells in the grid) although some of them will be bad states, meaning that if you reach one of those states you "lose".
The possible actions are obviously 4, as the possible directions you are able to follow, i.e. LEFT, RIGHT, UP, DOWN.
The model instead describes the rules of the game or better, describes what will happen if you do something in a particular place. In this case, since

the ice is slippery, when you choose to follow an action you will have 0.5 probability of going in the correct direction and 0.5 probability of ending up in a wrong one.
For example, given that you are in the top left starting corner, your possible actions are going DOWN or RIGHT. Let us say that you choose DOWN, mathematically you will get formally:

$$\begin{aligned}
\mathbb{P}(s_{t+1} = DOWN \mid s_t = START,\ a_t = DOWN) = 0.5\\
\mathbb{P}(s_{t+1} = RIGHT \mid s_t = START,\ a_t = DOWN) = 0.5\\
\mathbb{P}(s_{t+1} = UP \mid s_t = START,\ a_t = DOWN) = 0\\
\mathbb{P}(s_{t+1} = LEFT \mid s_t = START,\ a_t = DOWN) = 0
\end{aligned}$$

using the notation of s_{t+1} equal to DOWN (RIGHT, UP, LEFT respectively) meaning "the space you would end up into, having moved down (right, up, left respectively) of one cell".
We still have to define the reward in the game. Since the goal is a very good cell, the reward of reaching that state would be +1 and since all the other cells are just a path to the good cell, they would give reward 0.

Now that we have seen all the definitions in a game example, the understanding should have become clearer.

2.3 The reward function

We still have to give a closer look to the reward.
The reward is a function that maps the space and the action to a real number that can be translated into the *goodness* of your action and can be mathematically written as:

$$r(s, a, s') = \mathbb{E}[\mathcal{R}_{t+1} \mid S_t = s, A_t = a, S_{t+1} = s'] \tag{2.1}$$

since it gives the expected immediate reward.

The **return** is the total discounted sum of the rewards from time step t:

$$\mathcal{R}_{t+1} + \gamma \mathcal{R}_{t+2} + \gamma^2 \mathcal{R}_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k \mathcal{R}_{t+k+1} \tag{2.2}$$

Where the discount $\gamma \in [0, 1]$ is the present value of future rewards. Thus, if $\gamma < 1$ then rewards far in the future worth exponentially less than rewards received in the first stages.
If $\gamma < 1$ then the MDP is defined as *discounted* MDP instead if $\gamma = 1$ the MDP is *undiscounted.*

We have to explain better the reward function. If $\gamma = 1$ and for every step the reward is positive, then actually the sum from 0 to ∞ is equal to ∞. If this is the case, then there is no real gain in changing state since the return at an infinite time step in the future is infinite so it doesn't matter to move now or later. This is the *existential dilemma of immortality* that says: if I have infinite time in the future and I know that moving now or later (making an action) will bring me to infinite reward, why should I move now?
This is why the discounted reward is important. It treats rewards nearest to the current time step more with respect to the ones further in the future. The discounted summation has a boundary, in opposition to the undiscounted one: in fact, if we define as $\mathcal{R}_{max}$ the maximum reward for one single step, and hypothetically we say that every step is rewarded with this amount of reward, still the discounted return is bounded by:

$$\sum_{k=0}^{\infty} \gamma^k \mathcal{R}_{max} = \frac{\mathcal{R}_{max}}{1 - \gamma} \tag{2.3}$$

that is actually a geometrical series and so it can be easily computed. As we can see, if γ is close to 0, the higher boundary is near $\mathcal{R}_{max}$ and when γ is near to 1, the higher boundary becomes bigger and bigger until, for $\gamma = 1$, the result degenerates to ∞.

The goal of the system, or decision-maker, is to choose a behavior that maximizes the expected return.

2.4 Policy

The Markov Decision Process as depicted above describes a problem. Given a problem, we always need to find a solution; and a solution for a Markov Decision Process is called a **policy** π. But, *what is a policy?* A policy is a sort of "brain" for the decision maker that tells how to choose actions. There are two kinds of policies:

- **deterministic policies** where the action is just a function of the state:
$$a = \pi(s) \tag{2.4}$$
- **stochastic policies** where the action is a conditional distribution given a state:
$$a \sim \pi(a \mid s) \tag{2.5}$$

An **optimal policy** is the one the maximizes the long term return.
We'll go more in depth in policies in the next chapter.

2.5 Summary

A Markov Decision process has two main functions, the reward function $\mathcal{R}$ and the transition model function $\mathcal{P}$. This last one suggests the actions to do at each state in order to reach another state while the first one gives rewards after every step to suggest if you are going well or not.

When we have those two functions that is, we can predict which reward will be received and which will be the next state for any state-action pair, the MDP can be solved through Dynamic Programming techniques and the optimal policy can be found.
If those two functions can't be computed or are not available, other approaches must be taken and reinforcement learning is the best approach.

Chapter 3

From MDPs to Reinforcement Learning

As said before, when we have both the reward function $\mathcal{R}$ and the transition model $\mathcal{P}$ we can solve the MDP by means of dynamic programming.

Truth to be told, for most cases we cannot precisely predict neither $\mathcal{R}$ nor $\mathcal{P}$ and we have to approximate, explore and estimate functions and states in order to converge to the optimal solution (the optimal policy).

Reinforcement Learning in fact, is a technique to handle Markov Decision Processes in an environment we don't fully know.

3.1 Value Function and the Bellman equation

We spoke before about *optimal policy* but we never defined it. The optimal policy, is the one that maximizes our long term expected reward, that is:

$$\pi^{\star} = arg\,max_{\pi}\ \mathbb{E}\Big[\sum_{t} \gamma^t \mathcal{R}(s_t) \mid \pi\Big] \tag{3.1}$$

We can see in this function, that the initial state is not defined (it's random), and that the optimal policy must be optimal starting from any state.

We can actually define also the **value function** that gives the expected long term reward following a policy starting from a state s:

$$\mathbb{V}^{\pi}(s) = \mathbb{E}\Big[\sum_{t} \gamma^t \mathcal{R}(s_t) \mid \pi,\ s_0 = s\Big] \tag{3.2}$$

This function helps us to define in a better way the optimal policy:

$$\pi^{\star} = arg\,max_a \sum_{s'} \mathbb{P}(s_{t+1} = s' | s_t = s, a_t = a)\ \mathbb{V}(s') \tag{3.3}$$

This equation says that the optimal policy is the one that, for every state, returns the action that maximizes my expected value function and this allows us to write the value function in a much more famous form known as **Bellman Equation**:

$$\mathbb{V}^{\pi}(s) = \mathcal{R}(s) + \gamma \sum_{s'} \mathbb{P}(s_{t+1} = s' | s_t = s, a_t = a)\ \mathbb{V}(s') \tag{3.4}$$

This equation has two terms and says that the expected sum of discounted return is equal to the sum of two terms: the initial, or immediate, reward received immediately simply for starting in state s, and a discounted sum of the future expected rewards (rewards after the first step) and can be rewritten as

$$\mathbb{V}^{\pi^{\star}}(s) = \mathcal{R}(s) + \gamma\, max_a \sum_{s'} \mathbb{P}(s_{t+1} = s' | s_t = s, a_t = a)\ \mathbb{V}(s') \tag{3.5}$$

to describe the optimal value function $\mathbb{V}^{\star}$.

3.2 Action-value function

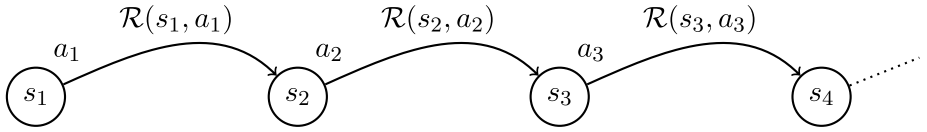

Figure 3.1: A representation of Bellman equation.

The Bellman equation is fundamental and is a necessary condition for optimality. Visually, our system goes like the one depicted in figure 3.1 and the Bellman Equation (3.3) describes this system immediately after state s_1, before taking action a_1, so when we talk about *value* we talk about a state and eventually we will get into another state that can also be represented with another value function. This is why the Bellman equation can be depicted in a recursive way.

The purpose of explaining the Bellman equation graphically is to show that, as the value function shows the infinite recursion of the system starting from state s, we can actually represent this recursion, starting immediately after the action is taken.

This equation is called **action-value function** and is defined in two ways. The first one in the Bellman equation form:

$$\mathbb{Q}^{\pi}(s,a) = \mathcal{R}(s,a) + \gamma \sum_{s'} \mathbb{P}(s_{t+1} = s' | s_t = s, a_t = a) \ \ max_a \mathbb{Q}(s', a') \tag{3.6}$$

and the second one in the expectation form:

$$\mathbb{Q}^{\pi}(s,a) = \mathbb{E}\Big[\sum_t \gamma^t \mathcal{R}(s_t) \mid \pi, \ s_0 = s, \ a_0 = a\Big] \tag{3.7}$$

This equations can be explained as we are in state s, we take action a, we get the first reward that is the first term, and then we go on with the summation.

The optimal value and action value functions are related in the following way.

$$\begin{aligned} \mathbb{V}^{\star}(s) &= max_a \mathbb{Q}^{\star}(s,a) \\ \mathbb{Q}^{\star}(s,a) &= \mathcal{R}(s,a) + \gamma \sum_{s'} \mathbb{P}(s_{t+1} = s' | s_t = s, a_t = a) \ \mathbb{V}^{\star}(s') \end{aligned}$$

One way of thinking about this is: we always go further in the cycle. For the value function, we need the best possible action and the state-action function has already taken an action, so I just get the max state-action function. Same way for the second function, state-action function starts after the action is taken, so I sum the reward and I continue at the landing state so with the value function.

Someone may ask themselves *why are we bothering of creating this action-value function when we already have the value function?*
Well, the *action-value function* is key in reinforcement learning as it allows to take expectations of $\mathbb{Q}^{\pi}(s, a)$ without knowing the transition function or the reward function.

3.3 Algorithms

Now that we have defined two important functions, we can describe a reinforcement learning algorithm.

3.3.1 Introduction

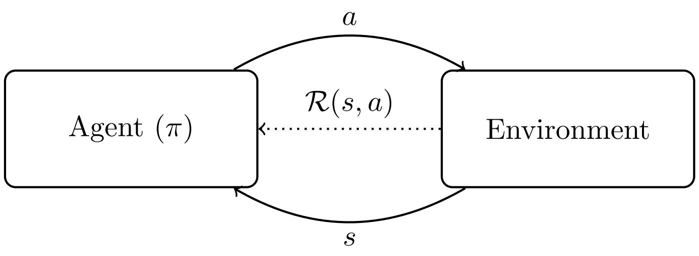

Figure 3.2: The concept of reinforcement learning.

To understand better reinforcement learning, it's better to see Figure 3.2. The environment reveals to the agent in forms of states and the agent replies with actions and the environment sends back a reward. All the computation or, in a more reinforcement learning way, the *thinking*, is done in the agent *mind* through the policy.
An important concept is that the information regarding the environment is only available through this interaction, meaning that the environment is not available in the agent's brain (the policy), but the agent is *experiencing* the environment only by interacting with it.

3.3.2 Different families of algorithms

We can think of a RL algorithm as a method that, given a sequence of action-state-reward triplets, is able to find an optimal policy π as depicted in Figure 3.3. The RL algorithm block can be split in different ways according

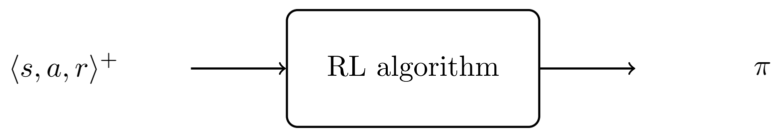

Figure 3.3: Reinforcement learning algorithm

to the way the policy is learned. We can divide algorithms in three main families.

Model-based algorithms

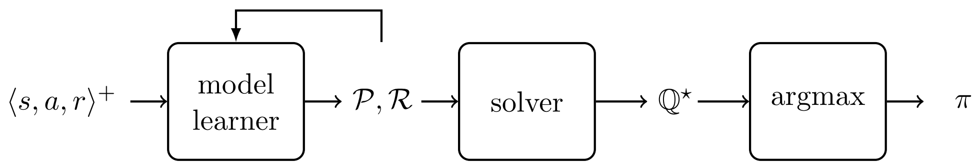

Figure 3.4: A model-based algorithm

A **model-based** algorithm takes the action-state-reward triples and sends them to a *model learner* which learns the transition probability function $\mathcal{P}$ and the reward function $\mathcal{R}$. Once learned, a solver can compute an optimal action-value function. Once you have an optimal action-value function, for each state you just get the argmax to get the best action, and that's the policy. The reason we have an arrow going back in the model learner is that the transition probability and the reward functions are initially estimated and are continuously updated with previous values of the functions.

Value-function based algorithms

The **value-function** based algorithms are almost the same as the previous family of algorithms except that, as we can see from figure 3.5, one block is missing. The main difference is that this family of algorithms tries to learn a

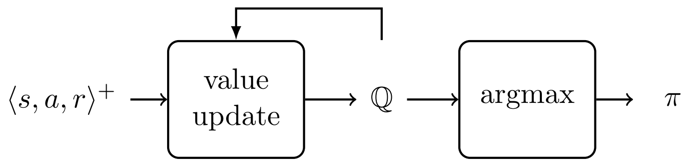

Figure 3.5: A value-function based algorithm

policy without explicitly learning the model or better, the policy is learned without the assumption of knowing the model, whereas in the first family, to learn the policy, the model *must* be learned (the two functions that define a model are the transition probability function and the reward function). This family of algorithm is in fact also called **model-free algorithms**.

In this family, the state-action function is directly estimated from the sequence of tuples from the environment and previous values of $\mathcal{Q}$. The $\mathcal{Q}$ doesn't have a $\star$ since it's just an estimation and it's not an optimal value but, with continuous iterations, it is proven that it will reach an optimal value.

Policy search algorithms

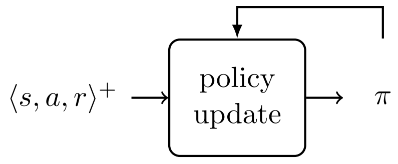

Figure 3.6: A policy search algorithm

The third family of algorithms updates directly the policy by feeding it back to the policy update that modifies it based on the sequence that it receives. This algorithm is much more direct, but the learning problem is much more difficult because the feedback you get from the policy usually is not that useful to update it. For example, in a game, knowing that an action moves you forward isn't sufficient to know when to use it.

As depicted in figures 3.4, 3.5 and 3.6, we can see that the three algorithms are getting simpler as going down, meaning that the policy learning is more direct but in the opposite way, going up, the algorithms become more supervised. Most of the time of this semester project has been spent in model-free value-based algorithms that provide a good balance between learning techniques and learning difficulties.

3.3.3 Off-policy and on-policy algorithms

A further distinction between algorithms can be made through another aspect, if they are *off-policy* or *on-policy*.

- **off-policy** algorithms are those algorithms that learn about the greedy strategy (policy), while following a different policy that allows a better exploration of the state space;
- **on-policy** algorithms are those algorithms that estimate functions according to the current policy.

An on-policy algorithm is influenced by the exploration policy and can get stuck in a local maximum whereas an off-policy algorithm is independent from the exploration policy and reaches the global optimum.
Algorithmically, an on-policy algorithm updates its $\mathcal{Q}$-values using the $\mathcal{Q}$-value of the next state s' and the current policy's action a' so it estimates the return for state-action pairs assuming the *current policy* continues to be followed. An off-policy algorithm updates its $\mathcal{Q}$-values using the $\mathcal{Q}$-value of the next state s' and the greedy action a' so it estimates the return for state-action pairs assuming a *greedy policy* was followed despite the fact that it is not following a greedy policy.
This distinction disappears when the followed policy is a greedy policy but an agent that works like this wouldn't be as good since it would never explore the space of solutions.

3.3.4 Deep Q-Network

Deep Q-Network, or **DQN**, is the first popular reinforcement learning algorithm, proposed by DeepMind in 2013 [8].

It's an updated version of the Q-learning algorithm first proposed by Watkins in 1992 [18] and uses deep learning to estimate the action-value function. The Q-network is the neural network function approximator with parameters θ.

Since it approximates and updates the action-value function, DQN is a *model-free*, *value-function based* algorithm: it solves the reinforcement learning task directly using samples from the emulator, without explicitly constructing an estimate of the model.

The two main characteristics regarding this algorithm are how it approximates the action-value function and the **experience replay**'s concept.

Deep Reinforcement Learning

Deep reinforcement learning can be defined as the connection between reinforcement Learning and neural networks, started in the late 90s [15] but with DQN it started to get exciting achievements and visibility. The deep reinforcement learning part is in the approximation of the action-value function with a convolutional neural network with 3 hidden layers. The input is an $84 \times 84 \times 4$ image, the first hidden layer convolves 16 8×8 filters with stride 4, the second hidden layers convolves 32 4×4 filters with stride 2, the final hidden layer is a fully-connected layer that consists of 256 rectifier units and the output layer is a fully-connected linear layer with a single output for each valid action.

The input image represents the state and the output actions are the actions that the agent can perform. It is interesting to see that this same network has been applied to seven Atari games and no preprocessing has been applied to the images (the algorithm worked directly with raw images from the games).

Experience Replay

The concept of experience replay was first introduced in 1992 [6] as a way to reuse past experiences and replay them to the network in order to refresh them to it. The *re-learning problem* says that if an input pattern has not been presented for quite a while, the network typically will forget what it has learned for that pattern and thus need to re-learn it when that pattern is seen again later. In DQN, a *replay memory* is used to store past experiences and show them again in a random way.

Algorithm 1 Deep Q-learning with Experience Replay

```
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
for episode $= 1, M$ do
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image
$x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
    end for
end for
```

Algorithm 1 shows the pseudo code of DQN. We can see that after a random initialization of the convnet and of the replay memory, for each episode and for each time step (image in the episode), to the corresponding state s_t an ϵ-greedy policy is applied (off-policy learning).
Optimization is then performed using stochastic gradient descent based on the $\mathcal{Q}$-learning target and the $\mathcal{Q}$-network.

Note that the preprocessing on the sequence is just a rescaling of the image, for lower memory utilization, no other preprocessing has been performed.

This implementation has been tested with 7 Atari games. It showed incredible results, obtaining for every game higher results than previous algorithms and sometimes even outperforming humans.

3.3.5 Asynchronous Advantage Actor Critic

A3C [7] is a framework developed by the same team as before, DeepMind, that outperforms their first model, DQN, with the only use of CPU (DQN instead relies heavily on GPU).
The authors present four famous reinforcement learning algorithms in an asynchronous way ($\mathcal{Q}$-learning, SARSA, n-step $\mathcal{Q}$-learning and A3C). What does asynchronous mean?
It means that parallel agents are executed in parallel, on multiple instances of the environment. This parallelism allows a wider view since at any given time-step the parallel agents will be experiencing a variety of different states and after a certain period policies are combined. It has been shown that experience replay is no longer needed and that the same results can be reached by increasing the number of instances running in parallel.
The asynchronous architecture is achieved using multiple CPU threads on a single machine, allowing every consumer computer to use this implementation.

The best performing algorithm of the four presented is the **Asynchronous advantage actor-critic (A3C)**.
First of all, it's called actor-critic since actor and critic are synonyms of policy-based and value-based respectively [5]. This means that this algorithm aims to combine the strengths of both families of algorithms: from the policy-based family of algorithms it gets the simplicity of the problem, optimizing the policy directly with no need to get the action-value function, whereas from the value-based algorithms it gets the ability to *learn*, since, when working with the policy, when it gets updated, the gradients are computed independently from the previous estimates so there is no *learning* in the sense of an accumulation and consolidation of informations. Actor-critic methods also have good convergence properties in contrast to critic-only methods.

This algorithm uses a so called *forward view* meaning that the algorithm selects actions using its exploration policy for up to a certain number of

Algorithm 2 Asynchronous advantage actor-critic - pseudo code for each actor-learner thread.

```
// Assume global shared parameter vectors $\theta$ and $\theta_v$ and global shared counter $T = 0$
// Assume thread-specific parameter vectors $\theta'$ and $\theta'_v$
Initialize thread step counter $t \leftarrow 1$
repeat
    Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
    Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
    $t_{start} = t$
    Get state $s_t$
    repeat
        Perform $a_t$ according to policy $\pi(a_t|s_t;\theta')$
        Receive reward $r_t$ and new state $s_{t+1}$
        $t \leftarrow t + 1$
        $T \leftarrow T + 1$
    until terminal $s_t$ or $t - t_{start} == t_{max}$
    $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
    for $i \in \{t-1, \ldots, t_{start}\}$ do
        $R \leftarrow r_i + \gamma R$
        Accumulate gradients wrt $\theta'$: $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
        Accumulate gradients wrt $\theta'_v$: $d\theta_v \leftarrow d\theta_v + \partial\left(R - V(s_i;\theta'_v)\right)^2 / \partial\theta'_v$
    end for
    Perform asynchronous update of $\theta$ using $d\theta$ and of $\theta_v$ using $d\theta_v$.
until $T > T_{max}$
```

steps in the future and the agent will then receive up to the same number of rewards from the environment. Forward view in a sentence explains how far ahead you need to look to figure out the value of a state.
Deep learning techniques are used to estimate both the policy and the value function where for the policy there is a softmax output instead for the value function there is one linear output with all other layers of a convolutional neural network shared.
The updates on the policy and on the value function are then performed according to the gradient of an estimate of the advantage function that takes into account both the value function parameters and the policy parameters.

In algorithm 2 the pseudo code for the A3C algorithm.

Chapter 4

OpenAI Gym and Universe

Hinton *set the baseline for neural networks*
LeCun *set the baseline for convolutional neural networks*
ImageNet *set the baseline for visual recognition competitions*
DeepMind *set the baseline for deep RL algorithms*
OpenAI *set the baseline for deep RL environments*

After all the initial chapters regarding the reinforcement learning theory, it is a must to introduce **openAI** that, as written on their website [9], *is a non-profit AI research company, discovering and enacting the path to safe artificial general intelligence.*

OpenAI has published many works, especially in reinforcement learning, and has released two main tools to improve and forward research and publication in this field.
The two main tools are **Gym** [2], a toolkit for developing and comparing reinforcement learning algorithms and **Universe**, a software platform for measuring and training an AIs general intelligence across the worlds supply of games, websites and other applications.

For this project, the Universe environment of **Atari Pong** has been used, one of the first ever commercialized games, together with the **Universe Starter Agent**, a learning environment developed by openAI and available in their github repository [10].

Figure 4.1: Some of the environments available in Universe.

4.1 Starter Agent

Universe Starter Agent is the implementation of the A3C [7] algorithm adapted for Atari games and Universe environments. This implementation allows the utilization of all the environments available in Universe simply by specifying the interested one from command line.

In this implementation, the policy is estimated through a long short term memory (LSTM) network with 256 hidden states.

The command to start the agent is:

```
python train.py --num-workers 2
    --env-id PongDeterministic-v3 --log-dir /tmp/pong
    --visualise
```

where `--num-workers` is the number of threads spawned by the command, each one running its own policy, `--env-id` sets the environment, `--log-dir` sets the folder where to save the TensorBoard [1] files and `--visualise` is an optional parameter to view the agent playing the game in a docker screen.

When this command is executed, a new tmux session is launched with $n + 3$ terminals where n is the number of parallel agents. The 3 more terminals are dedicated to TensorBoard (one) and to check interactively the processes (two).

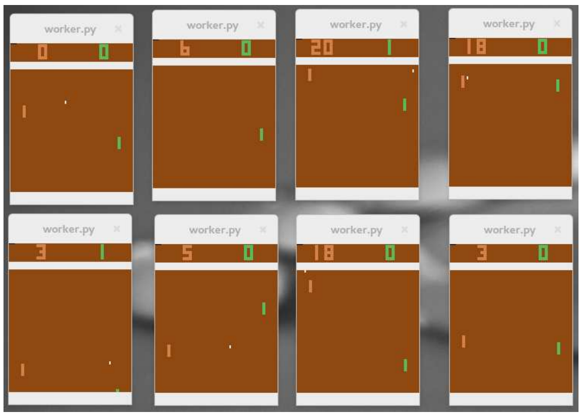

Figure 4.2: Atari Pong being played by 8 agents in parallel.

4.2 Pong

Atari Pong is one of the most simple games where the two players have to move a board in order to catch the ball and send it back to the other player. A point is made when you manage to get the ball over the opponent's board. A game finishes when a player reaches 20 points.

At the beginning, I made a test with the *universe-starter-agent* default parameters and obtained, obviously, incredible results:

- as expected, initially the policy was random and the agent was continuously losing the game since it didn't know what to do (the board was not moving at all);
- as time went by, the agent managed to move the board and catching the ball but still losing the game;
- it later understood that bumping the ball on one of the borders would make the ball bounce on the opposite direction making it harder for the opponent to catch it;
- after a sensible amount of iterations and almost 15 hours of training, each one of the 8 spawned agents was trying to make the same winning move at each step, managing to win every time with a score of 20-0.

The 2 main hyper-parameters that can be easily modified in the code are:

- the number of hidden layers in the LSTM policy estimator, default set to 256;
- `num_local_steps` which represents the amount of steps that each thread performs on its own before having policy and value functions updated. By default, this parameter is set to 20 steps.

I decided to perform various tests changing the second parameter, the essential one in an *asynchronous* algorithm.

The results were the following [10]:

- increasing the number of local steps lowers the variance in the estimation;
- increasing the number of local steps, will perform less frequent parameter updates slowing down the learning;
- decreasing the number of local steps below 20 makes the algorithm unable to learn.

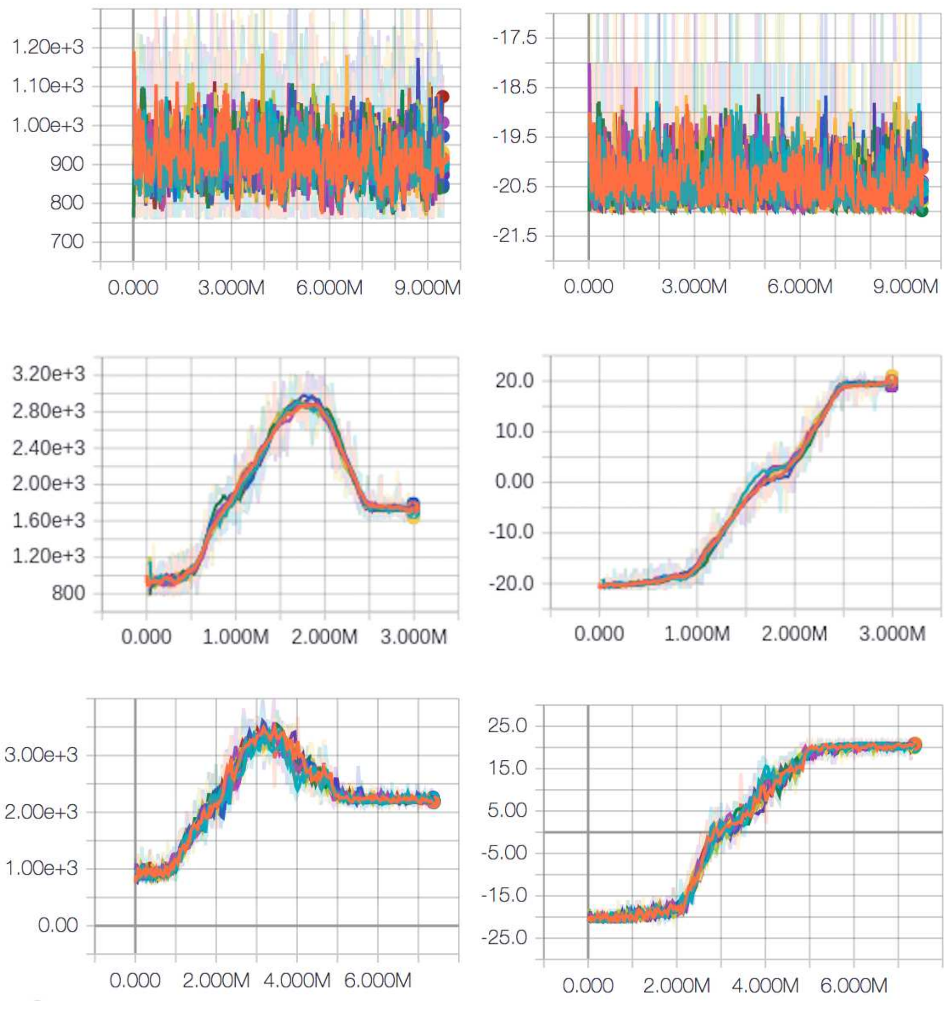

Figure 4.3: Above are shown the results for `num_local_steps` = 5, first row, 20, second row, 50, third row. The first plot in each row represents the episode length while the second represents the reward per episode.

Chapter 5

Conclusions and Future Works

After four months of continuous researching and exploring of the reinforcement learning field, it is essential to draw some conclusions and think about future works.

An important consideration to make is that at the beginning, I knew nothing about reinforcement learning and it has been an incredible exploration of the subject.

Putting hands on the universe-starter-agent and managing a high diversity of environments has been very helpful for the understanding of the subject.

Hyper-parameter selection is, as always, a cumbersome task and finding the correct set up, brute forcing the initialization, is possible only for big companies that have a lot of computing power. Still, there is no clear explanation of why some parameters are set in a certain way and why a Convnet or an LSTM must be built in a given way.

Future works certainly include a deeper understanding of different algorithms and implementation of improved function approximators.
A suggestion comes directly from DeepMind that considers Schulman et

al.'s generalized advantage estimation [12] to improve the estimation of the advantage function.

Certainly this field is one of the most actives in research groups recently, being that it could be improved in so many ways and that hypothetically could be used in so many applications, from self-driving cars to robotics, from inventory and delivery management to financial pricing and much more.

Chapter 6

Resources

An incredible amount of resources has been used during this semester in order to understand reinforcement learning, LSTMs, convolutional neural networks and all the other material that has been presented in this report. I would like to post here some courses, blogs, papers useful for this journey.

- Reinforcement learning book by Richard S. Sutton and Andrew G. Barto [11] available at
  `http://incompleteideas.net/sutton/book/the-book.html`

- Algorithms for Reinforcement Learning by Csaba Szepesvári [14] available at
  `https://sites.ualberta.ca/~szepesva/RLBook.html`

- Deep learning book by Ian Goodfellow, Yoshua Bengio, and Aaron Courville [4] available at
  `http://www.deeplearningbook.org/`

- On the origin of Deep Learning by Haohan Wang and Bhiksha Raj [17] available at
  `https://arxiv.org/abs/1702.07800`

- Sergey Levine, John Schulman and Chelsea Finn, CS 294: Deep Reinforcement Learning course from UC Berkeley [13] available at `http://rll.berkeley.edu/deeprlcourse/`

- Emma Brunskill, CS 234: Reinforcement Learning course from Stanford University, available at
  `http://web.stanford.edu/class/cs234/`

- Charles Isbell, Michael Littman and Pushkar Kolhe, Udacity: Machine Learning: Reinforcement Learning [3], available at
  `https://www.udacity.com/course/reinforcement-learning--ud600`

- Fei-Fei Li, Andrej Karpathy and Justin Johnson, CS231n: Convolutional Neural Networks for Visual Recognition course from Stanford available at
  `http://cs231n.stanford.edu`

- Richard Socher, CS224d: Deep Learning for Natural Language Processing course from Stanford available at
  `http://cs224d.stanford.edu`

- Andrej Karpathy blog post on Deep Reinforcement learning available at
  `http://karpathy.github.io/2016/05/31/rl/`

- Christopher Olah blog post on LSTMs available at
  `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`

- John Schulman talk at Machine Learning summer school available at
  `https://youtu.be/aUrX-rP_ss4` and the 3 next videos

- David Silver, Reinforcement Learning course from UCL available at
  `http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html`

Bibliography

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I. J., Harp, A., Irving, G., Isard, M., Jia, Y., Józefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D. G., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P. A., Vanhoucke, V., Vasudevan, V., Viégas, F. B., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR abs/1603.04467* (2016).

[2] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. Openai gym. *CoRR abs/1606.01540* (2016).

[3] Charles Isbell, M. L. Reinforcement learning on udacity. `https://www.udacity.com/course/reinforcement-learning--ud600`.

[4] Goodfellow, I., Bengio, Y., and Courville, A. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[5] Konda, V. R., and Tsitsiklis, J. N. On actor-critic algorithms. *SIAM journal on Control and Optimization 42*, 4 (2003), 1143–1166.

[6] Lin, L.-J. *Reinforcement Learning for Robots Using Neural Networks.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1992.

[7] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., and Kavukcuoglu, K. Asynchronous methods for deep reinforcement learning. *CoRR abs/1602.01783* (2016).

[8] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. A. Playing atari with deep reinforcement learning. *CoRR abs/1312.5602* (2013).

[9] openai website. `https://openai.com/`.

[10] openAI. Universe starter agent. `https://github.com/openai/universe-starter-agent`.

[11] Richard S. Sutton, A. G. *Introduction to Reinforcement Learning*, 1st ed. MIT Press, Cambridge, MA, USA, 1998. `https://sites.ualberta.ca/~szepesva/RLBook.html`.

[12] Schulman, J., Moritz, P., Levine, S., Jordan, M. I., and Abbeel, P. High-dimensional continuous control using generalized advantage estimation. *CoRR abs/1506.02438* (2015).

[13] Sergey Levine, Chelsea Finn, J. S. Cs 294: Deep reinforcement learning. `http://rll.berkeley.edu/deeprlcourse/`.

[14] Szepesvari, C. *Algorithms for Reinforcement Learning.* Morgan and Claypool Publishers, 2010. `https://sites.ualberta.ca/~szepesva/papers/RLAlgsInMDPs-lecture.pdf`.

[15] Tesauro, G. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Comput. 6*, 2 (Mar. 1994), 215–219.

[16] openai universe. `https://blog.openai.com/universe/`.

[17] Wang, H., Raj, B., and Xing, E. P. On the origin of deep learning. *CoRR abs/1702.07800* (2017).

[18] Watkins, C. J., and Dayan, P. Technical note: Q-learning. *Machine Learning 8*, 3 (1992), 279–292.