UNIVERSITÉ NICE SOPHIA ANTIPOLIS

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

Erasmus Mundus Consortium "MathMods"

Double Master's Degree Programme in
Mathematical Modelling in Engineering: Theory, Numerics, Applications

Master Sciences, Technologies, Santé à finalité Recherche Mention Mathématiques et Interactions Spécialité Mathématiques Pures et Appliquées	Laurea Magistrale in Ingegneria Matematica
UNIVERSITÉ NICE SOPHIA ANTIPOLIS	UNIVERSITÀ DEGLI STUDI DELL'AQUILA

In the framework of the
Consortium Agreement and Award of a Joint/Multiple Degree 2013-2019

Master's thesis

Reinforcement Learning for Self-Driving Cars

Supervisor
Dr. Antonio López

Co-Supervisor
Prof. François Delarue

Candidate
Diego Porres

Matricola: 243867

2016/2017

Abstract

Reinforcement Learning (RL) is one of the future players in the long-term for self-driving vehicles. Although Deep Learning (DL) has succeeded in solving many of the sub-problems of autonomous driving (such as object recognition), it is still not guaranteed that the union of these sub-problems will lead us into self-driving vehicles. Indeed, recognizing every object in the field of vision is not what people normally do as they drive, and a better understanding of how *we* drive will surely make sure that we do not let our machine counterparts to make the same mistakes.

We will learn and apply one of the latest state-of-the-art Reinforcement Learning algorithms, the A3C, and apply it to many different environments, in particular to video games, both old and new. We also present the latest (up to the date of the presentation) Reinforcement Learning algorithms, both their results if they are applied to self-driving cars, as well as many promising ones in the field.

Acknowledgements

I would like to offer my special thanks to Dr. Antonio López of the Computer Vision Center. Without his offer for me to participate in this project and guidance, I would not have been able to enter or learn from the exciting field of self-driving cars.

My special thanks are extended to the staff at both the Computer Vision Center and MathMods committee, without whom I would not have been able to push through hard times.

Finally, I wish to thanks my family for their eternal support, without which I would not have been able to finish this project or my Master studies.

To everyone involved:

Thank you.

Table of Contents

Tables

Table **Page**

Figures

List of Algorithms

Chapter 1

Introduction

The world of self-driving vehicles is evolving rapidly. Car manufacturers are not the only ones developing their own hardware and software in order to be the industry leaders. Startups and technological giants, such as Google and Uber, are also developing their own self-driving car. The Society of Automotive Engineers (SAE) International have defined the new standard J3016_201609, in which basically they identify six levels of driving automation, from SAE Level 0 or *no automation* to SAE Level 5 or *full automation* (SAE International 2016).

In the past, we have become accustomed to SAE Level 1 of driving automation, namely the cruise control present in many automatic vehicles. The idea of many car companies—such as Tesla—is to slowly add more and more features to the software in their self-driving vehicles as time progresses, until reaching the much-coveted SAE Level 5. On the other hand, the Google's self-driving team at Waymo has said this is a bad strategy, and they should focus entirely on building, from scratch, an SAE Level 5 self-driving vehicle [1]. Whichever strategy one might deem is better (whether on the point of view of design or customer satisfaction), both seem to be working well so far for the companies.

Though still halfway through the internship, we hope that the work presented in this report may eventually lead to equally powerful technologies down the road, as teaching a car to self-drive via Reinforcement Learning is a powerful tool to combine with the other branches of Machine Learning, but it has not yet been successfully applied in automotive applications (Sallab et al. 2017). Indeed, Reinforcement Learning will not depend only on the data available to one company in particular, aiding to democratize the creation of self-driving vehicles in the near future.

1.1 Advanced Driver Assistance Systems

The Advanced Driver Assistance Systems (ADAS) is a research group from the Centre de Visió per Computador (CVC), based at the Autonomous University of Barcelona (UAB). Their aim, implied by their name, is to combine sensors and algorithms to understand the vehicle's environment so that the driver can receive assistance or be warned of potential hazards.

In the world of self-driving cars, much has been improved since Stanley (Thrun et al. 2006), but as technology has improved, so has the computational and hardware requirements to make better estimates of detection and predictions. While other self-driving cars use advanced systems for detecting their environments, such as LIDAR, radars, or lasers, the car being developed at the CVC is being built based on low-cost technology, such as cameras (Figure 1.1b). The CVC is automating an italian electric car, the *Tazzari Zero*, within the context of the project Automated and Cooperative Driving in the City (ACDC). This is having the long-term goal of launching faster to the market, than e.g. continuing Stanley's line of technology.

To achieve the goal of designing and developing a self-driving system for urban scenarios, the group ADAS has decided to break it down into the following sub-goals:

1. Design and develop navigation software based on the use of vision, radar, GPS, vehicle state, and maps.
2. Design and develop object detection software specialized in dynamic traffic participants, with the capability of anticipating their intentions. This software will rely on vision, radar, vehicle state, and ultrasound data.

[1] See Chris Urmson's TED Talk, *How a driverless car sees the road*, at `https://goo.gl/xfb1Qt`

3. Design and develop software to detect unknown obstacles lying on the road (hazards).
4. Design and develop software for detecting road marks (lane markings, traffic signs, zebra crossings, and traffic lights), in order to respect traffic rules, using vision.
5. Design and collect representative data sets for benchmark the self-driving capabilities.
6. Design and develop a simulation platform that can playback the benchmark data and evaluate the vehicle's behavior off-line.
7. Design and execute self-driving demonstrations in controlled urban areas (in the UAB campus of Bellaterra).
8. Incorporating cooperative information into the self-driving procedure by collaborating with other groups.

(a) (b)

Figure 1.1 a) The Advanced Driver Assistance Systems logo. b) The autonomous vehicle developed by ADAS, the Elektra, within the context of the project Automated and Cooperative Driving in the City (ACDC).

1.2 Objectives of the Internship

With these sub-goals in mind, our project will be aiding in accomplishing ADAS's goal of designing and developing a self-driving system. While much effort has been put into deep learning and Deep Neural Networks (DNN), the world of Reinforcement Learning is vastly unexplored and has much to offer to the world of self-driving vehicles. Indeed, if there is one thing always missing is students in order to help accomplish these projects. As such, it is our goal to lower the learning curve needed to enter into the Reinforcement Learning world for new students in ADAS.

Concretely, the objectives of the project are:

Main Objective

Implement the state-of-the-art algorithm A3C (Mnih et al. 2016) in Reinforcement Learning gaming environments with similarities to self-driving cars, such as Rally games for example. This was at the moment of the start of the internship; see Chapter 7 onwards for the latest algorithms that have since been released.

Secondary Objectives

- Test and implement the A3C algorithm in Tensorflow (Abadi et al. 2016) so as to be easily transferable to the Unity environment that the ADAS team is currently building.
- If possible, apply the A3C algorithm to a real-world RC car for it to find the best path in a go-kart circuit recently repurposed for the ADAS team to use.
- Provide a starting point and a guide for the ADAS team to easily understand and implement the techniques and algorithms of Reinforcement Learning in present and future projects.
- Provide a summary of the different frameworks that have been used in the past to implement reinforcement learning in self-driving vehicles.

Chapter 2

Reinforcement Learning

As any other field in Machine Learning, Reinforcement Learning has a promising future, but it is one of the less known. Reinforcement Learning has its differences from the more known areas of Machine Learning, Supervised and Unsupervised Learning. As such, it would be perhaps necessary to first introduce this type of problem and its solutions so that the reader may comprehend the many applications that it has.

This chapter is a synthesis, taken from (Sutton & Barto 1998), (Szepesvári 2009), and (Silver 2015), as well as many other papers in the area, as a way of introducing the reader to this subject. In no way is this meant to be a full introduction, as we will skip most parts that do not pertain to the area we will focus on in the A3C Algorithm (Chapter 3). As such, it is highly encouraged to the reader to seek these resources which can be found online, as well as Udacity's free course on Reinforcement Learning (Littman et al. 2012).

2.1 Short History

Before starting on the theory behind Reinforcement Learning (RL), we should take a look at its roots, at the history that brought it to us. In a broad scale, RL has three main 'sources': one being the psychology of animal training and the research into learning by trial and error, another being the problem of optimal control and its solution using value (or utility) functions and Dynamic Programming (DP), and finally the one concerning Temporal-Difference (TD) methods. Although these sources all had independent research behind them since their inception, they all came together in the late 1980's to form what is now known as Reinforcement Learning (Sutton & Barto 1998).

In a more specific scale, Reinforcement Learning sits at the intersection of many fields of science (Figure 2.1a). The problem that Reinforcement Learning tries to attack and solve is the science of decision-making, and hence why it intersects with many fields, such as in Economics where Game Theory is the tool used to try to tackle this problem.

More specifically, Game Theory studies the mathematics of conflict or conflicts of interest when trying to make optimal choices. The goal is to first model a single agent and then multiple agents interacting, taking into account explicitly the goals and desires of each. The Nash Equilibrium is perhaps one of the most known results outside of this area of science, and Game Theory itself is becoming increasingly a bigger part of Artificial Intelligence and Reinforcement Learning nowadays (Littman et al. 2012).

In the area of Psychology, Edward Thorndike tackled early on the problem of learning and lead the foundations of what later became operant conditioning within behaviorism (Thorndike 1898). The main difference between classical conditioning (i.e., developing associations between events) and operant conditioning is that the latter involves learning from consequences of our behavior (McLeod 2007).

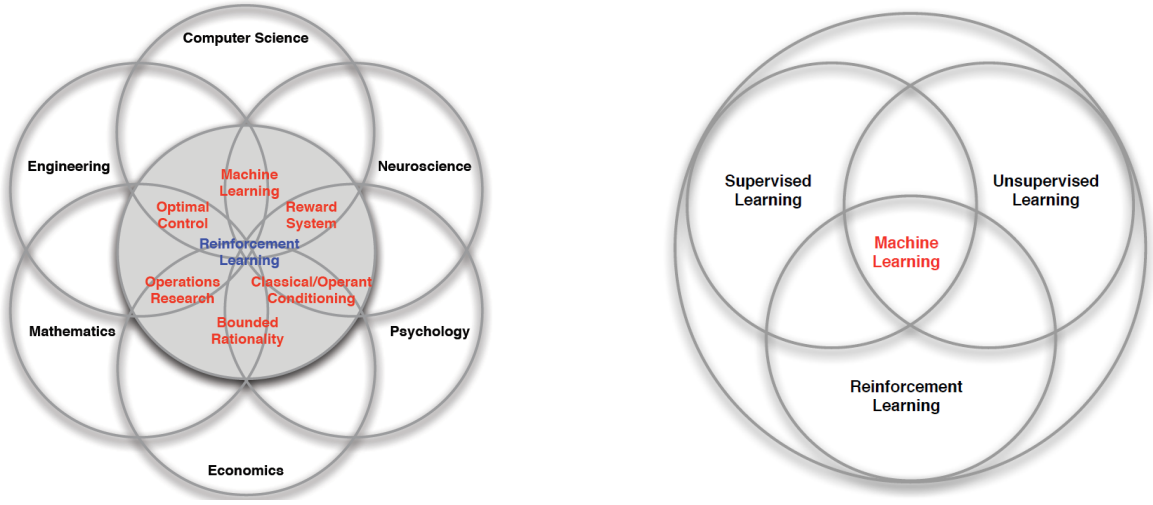
Thorndike proceeded to study how animals learned. He did this by means of experimentation by placing a cat in a puzzle box, and food was placed outside to incentivize the cat to escape the box. It was not until the cat pushed the right lever that the door would open, letting the cat escape so that it could eat the food located outside. Thorndike measured the time it took for the cats to escape, and noted that the cats would learn that pressing the specific lever had favorable consequences, and hence adopted this behavior, every time becoming increasingly faster at pressing the right lever.

Thorndike named this the *Law of effect*. More specifically:

> Responses that produce a satisfying effect in a particular situation become more likely to occur

again in that situation, and responses that produce a discomforting effect become less likely to occur again in that situation.

There are, of course, other interesting areas regarding the origin of both Reinforcement Learning and Artificial Intelligence, such as Neuroscience, but we won't delve into these subjects. The main point of this section is to perhaps emphasize how the different fields eventually merged together in the late 1980's, when Christopher Watkins developed Q-learning (Sutton & Barto 1998, Watkins & Dayan 1992).



(a) (b)

Figure 2.1 Venn diagrams depicting a) where Reinforcement Learning sits with respect to other fields of science, and b) how the different fields of Machine Learning compare. Reproduced, with permission, from (Silver 2015).

2.2 Reinforcement Learning

Reinforcement Learning, like many areas in Machine Learning, is both a problem, a class of solution methods, and the field that studies both these problems and solution methods (Figure 2.1b). In the end, what really differentiates Reinforcement Learning from Supervised and Unsupervised Learning can be summarized in the following four points: there is no supervisor (or supervising agent), only a reward signal given by the environment, the feedback is delayed (usually at the end of each session), time matters (the data is sequential, but not i.i.d.), and the agent's actions influence the subsequent data it receives (Silver 2015).

In broad terms, Reinforcement Learning deals with learning what to do in order to maximize a numerical reward, i.e., there is a ***learner*** (or goal-seeking ***agent***) that interacts with its ***environment*** to achieve a certain goal. The ***policy*** will define how our agent will behave at a given time. It is the mapping from *perceived* states of the environment to the actions that will be taken by the agent when in any of those states. The policy alone can determine the behavior of our agent, and thus may be stochastic to, e.g., better mimic the actions of a person or when our environment may drastically change due to other factors. Therefore, Reinforcement Learning promises to find 'a way of programming agents by reward and punishment without needing to specify *how* the task is to be achieved' (Kaelbling et al. 1996).

The ***reward signal***, or ***reward function***, will define numerically the goal of our Reinforcement Learning problem. On each time step, the environment will send our agent a single number, the ***reward*** (whether deterministic or stochastic), and it is the purpose of our agent to maximize the total reward it receives until the last time step. Due to them being given by the environment, our agent cannot and should change it, as this would simply be it solving a different problem. Therefore, Reinforcement Learning is a way of

The ***value*** of a state is the total (or expected) reward that an agent can expect to accumulate in the future if it starts in said state. Therefore, the ***value function*** will specify our agent what is good in the long run, and as such their only purpose is for the agent to obtain a higher reward. In other words, the rewards

are given by the environment, whereas the value function is estimated and re-estimated by the agent each time it visits different states in the environment.

Finally, the ***model*** of the environment mimics the environment's behavior. It is used for planning or deciding which course of action to take by considering plausible scenarios before they occur. Therefore, the methods to solve Reinforcement Learning problems that use planning are called ***model-based*** methods, whereas simpler trial-and-error learners are called ***model-free***. In short, Reinforcement Learning is the problem of getting an agent to act in its environment so as to maximize its rewards.

As a concluding note, it should be noted that there are other methods to solve Reinforcement Learning problems than the ones we will see in the following Sections. These are called ***evolutionary methods*** and include genetic algorithms, genetic programming, and simulated annealing, among others. We will see the results of such algorithms in Section 7.5. Their advantage is that they do not deal with value functions, and are better suited whenever the space of policies is sufficiently small, or when the learning agent cannot sense accurately the state of the environment it's in (Sutton & Barto 1998).

2.3 Finite Markov Decision Processes

This section will provide us with a mathematical definition of the terms we have previously seen in Section 2.2, as well as others that are useful in the area of Reinforcement Learning problems. We will define what is a Markov Decision Process (MDP) and focus on finite MDPs, but note that under some technical conditions, the same results can be extended to continuous state-action MDPs as well (Szepesvári 2009).

2.3.1 General Setting

Since we are formally defining the problem of learning, we call the ***agent*** the learner and decision-maker. The thing the agent interacts with is called the ***environment*** $\mathcal{E}$, where the general rule is that anything that cannot be changed by the agent is considered to be outside of it and hence part of its environment. We do not assume that everything in the environment is unknown to the agent, but this will become clearer as we introduce the notation for the terms we have previously defined.

As the agent and environment interact at the sequence of discrete time steps $t = 0, 1, 2, \ldots$, the agent will have some representation of the environment's ***state***, $S_t \in \mathcal{S}$, where $\mathcal{S}$ is the set of possible states in the environment $\mathcal{E}$. Having this, the agent will select an ***action*** $A_t \in \mathcal{A}(S_t)$, where $\mathcal{A}(S_t)$ is the set of actions that are available at the state S_t. At the time step $t + 1$, the environment will, in turn, give a numerical ***reward***, which we denote by $R_t \in \mathcal{R} \subset \mathbb{R}$, and the agent will find itself in the new state S_{t+1} (ref. Figure 2.2). This process continues until the agent reaches a ***terminal state***, usually denoted by S_T, after which the process restarts. We assume that the environment's state is not visible to our agent, but even if it were, it would usually contain irrelevant information (Silver 2015).

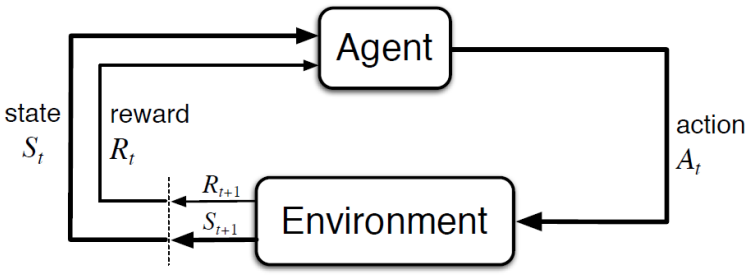


Figure 2.2 General representation of the interaction between a learning agent and its environment. Reproduced, with permission, from (Sutton & Barto 1998).

At each time step, our learning agent maps the state it currently findts itself in to probabilities of selecting each of the possible actions available at said state. We call this the agent's ***policy*** and denote it by π. If our policy is ***stochastic***, then $\pi : \mathcal{S} \times \mathcal{A}(S_t) \to [0, 1]$, where $\pi(a|s)$ is the probability that we take action $A_t = a$ in state $S_t = s$.

$$\pi(a|s) = \mathbb{P}\left[A_t = a | S_t = s\right] \tag{2.1}$$

On the other hand, if our policy is deterministic, then $\pi : \mathcal{S} \to \mathcal{A}(S_t)$, so $\pi(s) = a$, for $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$. Thus, RL methods specify how the agent changes its policy due to its experience. Note that a policy will define the full behavior of our agent, and depend only on the current state, not on the states, rewards, or actions taken beforehand (Silver 2015). We will explore this more in depth in Section 2.6.

2.3.2 Rewards and Returns

Since our agent's goal is to maximize the total amount of reward it receives, a distinction must be made between *immediate reward* and *long-term cumulative reward*. For example, while a particular move in chess might be good in the short term (e.g., eating the Queen), it might lead to more states in which we have our King exposed, making it more likely for us to lose the game.

This is called the ***reward hypothesis***, and constitutes the basis of Reinforcement Learning (Silver 2015). It states the following:

> *All* goals can be described by the maximization of expected cumulative reward.

Therefore, it is critical that the rewards are made in such a way that they indicate to our learning agent *what* we want it to achieve, but not *how*. Otherwise, this might lead to larger issues such as negative side effects and reward hacking, among others, as we will see in Section 6.4. We want the agent's goal to be something which it does not have perfect control of, so we consider the rewards as coming from outside of the agent (this does not stop the agent from formulating some sequence of *internal* rewards).

Let's denote $R_{t+1}, R_{t+2}, R_{t+3}, \dots$ as the sequence of rewards received by our agent after the time step t. The ***return***, denoted by G_t, is defined in general as a function of the reward sequence:

$$G_t = F(R_{t+1}, R_{t+2}, R_{t+3}, \dots) \tag{2.2}$$

Most likely, we won't be able to break down the interactions between our agent and its environment into identifiable ***episodes***, i.e. subsequences where the interaction of the agent with the environment stops. We denote the state where each episode ends, the terminal state, by S_T, where T is the final time step. Since $\mathcal{S}$ is the set of all non-terminal states, then we define $\mathcal{S}^+$ as the set of all states plus the terminal state.

It is possible that the interactions can go on without limit, without ever reaching S_T, i.e. $T = +\infty$, and if we don't choose our function F in Equation 2.2 correctly, the return G_t could also be infinite. This would lead our agent into entering a loop and never exit, since it would continue to gain (positive) rewards *ad infinitum* and hence gain the maximum return possible (see Section 6.4).

This is why we introduce the notion of ***discounting***: the ***discount rate*** $\gamma \in [0, 1]$ represents the future value of present rewards such that more immediate rewards weigh more than those further in the future. The total accumulated ***discounted return*** starting from time step t is then defined as:

$$G_t \doteq \sum_{k=0}^{+\infty} \gamma^k R_{t+k+1} \tag{2.3}$$

In other words, a reward received l time steps in the future is worth γ^{l-1} times what it is currently worth (if it where to be received immediately) (Szepesvári 2009). Our agent will choose, at the time step t, the action A_t that maximizes the expected discounted return. Setting $\gamma = 0$ means that our strategy will be short-sighted, i.e., only the immediate reward matters, but this might, in turn, reduce our return. On the other hand, setting $\gamma = 1$ only makes sense if our environment is deterministic, all the rewards are always the same, and $T < +\infty$. We can then see that as γ approaches 1, it means that our agent will take into account the future rewards more strongly.

If $R_t \leq R_{max} < +\infty, \forall t = 0, 1, 2, \dots$ and $\gamma < 1$, then:

$$G_t = \sum_{k=0}^{+\infty} \gamma^k R_{t+k+1} \leq \sum_{k=0}^{+\infty} \gamma^k R_{max} = R_{max} \sum_{k=0}^{+\infty} \gamma^k = R_{max}/(1-\gamma)$$

On a final note, we can generate a recursive relationship as follows:

$$\begin{aligned} G_t &= \sum_{k=t}^{+\infty} \gamma^{k-t} R_{k+1} \\ &= R_{t+1} + \gamma \sum_{k=t+1}^{+\infty} \gamma^{k-t-1} R_k \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{2.4}$$

Or, equivalently,

$$G_{t+1} = (G_t - R_{t+1})/\gamma \tag{2.5}$$

Hence, we can just remember the value of the new reward and update the return accordingly. The final goal for our agent, as discussed before, will be to maximize its ***expected return*** from each state S_t.

2.3.3 The Markov Property

The state at time step t, S_t, is constructed on immediate sensations by the agent and previous states or some other information of past sensations. However, this state signal should not be expected to carry all the information of the environment, or more specifically, to carry all the information necessary to make a useful decision. Thus, our agent should only be punished if it learns something and then forgets it, but not if it does not know something about its environment.

We seek that the state signal contains all the relevant information for the agent. The response at time $t+1$ that an environment may have to an action from an agent at time t may depend on all that has occurred up to that moment. But in a game of chess, for example, it is irrelevant *how* we got to the current state of the chessboard. All that matters is the current state itself and the possible moves we can make in order to determine the best course of action in order to win the game. These types of states are called ***Markov*** or said to have the ***Markov property*** if:

$$\begin{aligned} \mathbb{P}\left[S_{t+1} = s', R_{t+1} = r | S_0, A_0, R_1, \ldots, S_{t-1}, A_{t-1}, R_t, S_t, A_t\right] &= \mathbb{P}\left[S_{t+1} = s', R_{t+1} = r | S_t = s, A_t = a\right] \\ &\doteq p(s', r | s, a) \end{aligned} \tag{2.6}$$

where a is the action we take in the current state s leading us to state s', and the sequence $S_0, A_0, R_1, \ldots, S_{t-1}, A_{t-1}, R_t, S_t, A_t$ is called the ***history***; we use $p(s', r|s, a)$ as a shorthand. Using this equation, we should be able to predict all future states and expected rewards. Even if our state signals are not Markovian, it is still considered appropriate to think of it as an approximation to a Markov state.

2.3.4 Markov Decision Processes

If a Reinforcement Learning task satisfies the Markov property, i.e. *all* the states are Markovian, then it is called a ***Markov Decision Process*** or ***MDP***. An MDP is a framework designed to make decisions in problems full of uncertainty and complexity (Piñol 2014). We formally define it as the tuple $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$, where $\mathcal{S}$ is the ***state space***, the set of all possible states an MDP can be in, $\mathcal{A}$ is the ***action space***, the set of all possible actions available to our agent, $\mathcal{P}$ is the ***state transition probability matrix*** whose entries are given by $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$, $\mathcal{R}$ is the reward function, with $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$, and γ is the previously discussed discount factor such that $0 \leq \gamma \leq 1$.

If both $\mathcal{S}$ and $\mathcal{A}$ are finite sets, then we call $\mathcal{M}$ a ***finite MDP***. These are perhaps of the most important types of MDPs, and the ones we will focus on this project. Given Equation 2.6, we can therefore specify the dynamics of our environment. We define the ***expected rewards for state-action pairs*** (s, a) as:

$$\mathcal{R}(s, a) \doteq \mathbb{E}\left[R_{t+1} | S_t = s, A_t = a\right], \tag{2.7}$$

the ***state-transition probabilities***,

$$T(s, a, s') \doteq \mathbb{P}\left[S_{t+1} = s' | S_t = s, A_t = a\right], \tag{2.8}$$

where another notation commonly used for the state-transition probability is $p(s'|s, a)$, and the ***expected rewards for state-action-next-state triplets*** (s, a, s'),

$$\mathcal{R}(s, a, s') \doteq \mathbb{E}\left[R_{t+1} | S_t = s, A_t = a, S_{t+1} = s'\right], \tag{2.9}$$

Here a distinction must be made: in the literature, even though Equations 2.7 and 2.9 seem different, they are equivalent. Indeed, we can even write them shorter as simply $\mathcal{R}(s) = \mathbb{E}\left[R_{t+1} | S_t = s\right]$, where we simply state what is the reward of arriving at state s. To get this reward, it is implicit that we were in another state beforehand, and we took an action a to get to state s. We will interchangeably use these notations but hope no confusion arises from them (Littman et al. 2012).

Whenever possible, a picture of the environment will vastly improve our understanding of what exactly is happening and what decisions our agent is taking. This is what a ***transition graph*** is for. It has two kinds of nodes, the ***state nodes*** and the ***action nodes***. In Figure 2.3, we have an example for the transition graph: on state s, we have as possible actions $\mathcal{A}(s) = \{a, a', a''\}$, so we will have three state-action pairs: $(s, a), (s, a'), (s, a'')$.

In particular, on the state-action pair (s, a), we have a probability of $T(s, a, s')$ of transitioning to state s' and get a reward $\mathcal{R}(s, a, s') = r$. The probability of arriving to the terminal state s_T is equal to $T(s, a'', s_T)$, whereas if we take action a', we have a probability of $T(s, a', s')$ of arriving to state s' and a probability of $T(s, a', s)$ of staying in state s. Note that for each state-action pair, the sum of all the transition probabilities should sum up to 1.

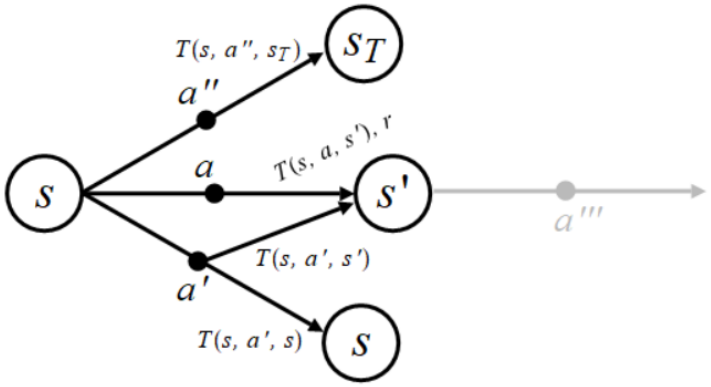


Figure 2.3 Example of a transition graph. Note that taking action a' on state s will bring us back to state s with probability $T(s, a', s)$ (e.g., the action could be *standing still* for a robot) so a loop can be used to represent this.

As a final note, we remember that an MDP assumes that an agent can always observe the state of the environment. This is not always the case, so to generalize the notion of an MDP we define the ***Partially Observable Markov Decision Process*** or ***POMDP***. In it, we do not assume that the state is perfectly observable and hence introduce an uncertainty over the state of the environment. Of course, this makes learning and decision making for our agent more complicated, but it can help to account for noise in our detection systems/sensors (such as LIDARs) in our self-driving vehicles, as is used in (Ulbrich & Maurer 2013).

2.4 Value Functions

One possible way to solve an MDP, to find the optimal strategy to take for every possible state, would be to list *all* the possible behaviors of our agent and simply identify those with the highest expected return. For small action and state spaces, this seems logical, but for continuous spaces, this is impossible.

Value functions are functions of the states, or state-action pairs, that estimate how good it is for the agent to be in a given state, or how good it is to take an action in a given state in the state-action pair case. In other words, we see what is the expected return to be in a given state or to take the action in said state.

For MDPs, the value of a state under a policy π is called the ***state-value function for policy*** π and denoted by $v_\pi : \mathcal{S} \to \mathbb{R}$. It is the expected return when starting in state s and following the policy π henceforth. In other words, if at the time step t we are in state s, then:

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s\right] \tag{2.10}$$

We define the ***action-value function*** for policy π as the value of taking action a in state s under a policy π, and denote it by Q_π, such that $Q_\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. This function is the expected return starting from s, taking action a, and thereafter following policy π:

$$Q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a\right] \tag{2.11}$$

What both of these value functions tell us is how *good* it is to be in a particular state or state-action pair in terms of the expected return. We will then choose certain states or certain state-action pairs, depending on their value.

One important feature of value functions is that they follow a recurrence relation. Continuing with Equation 2.10 and using Equation 2.4, it can be shown that (Silver 2015):

$$v_\pi(s) = \mathbb{E}_\pi\left[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s\right], \tag{2.12}$$

and likewise for the action-value function, using Equations 2.11 and 2.10:

$$Q_\pi(s, a) = \mathbb{E}_\pi\left[R_{t+1} + \gamma Q_\pi(S_{t+1}, A_{t+1})|S_t = s, A_t = a\right] \tag{2.13}$$

Equations 2.12 and 2.13 are called the ***Bellman equations*** for v_π and Q_π, respectively. On the finite case, we would have $|\mathcal{S}|$ equations with $|\mathcal{S}|$ unknowns. From here it can easily be seen that applying a brute force approach or tabular methods would not be very efficient as our state space grows considerably, as is the case even for finite spaces such as board games (ref. Section 2.6).

2.4.1 Optimal Value Functions

In order to finally solve our Reinforcement Learning problem, we must find a policy that achieves the maximum reward possible in the long run. We can differentiate which policies are better using the value functions defined in the previous section. We define it like so:

$$\pi \geq \pi' \Rightarrow v_\pi(s) \geq v_{\pi'}(s), \forall s \in \mathcal{S} \tag{2.14}$$

In other words, a policy π is better than (or equal to) another policy π' if its expected return is greater than or equal to that of π', for all states in the state space. We define the ***optimal state-value function***, $v_\star : \mathcal{S} \to \mathbb{R}$, as the maximum value function over all policies, so:

$$v_\star(s) \doteq \max_\pi v_\pi(s), \forall s \in \mathcal{S} \tag{2.15}$$

Likewise, the ***optimal action-value function*** $Q_\star : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is defined as:

$$Q_\star(s, a) \doteq \max_\pi Q_\pi(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s) \tag{2.16}$$

Theorem 2.1 *For any Markov Decision Process,*

- *There exists the* **optimal policy** $\pi_\star$ *that is better than or equal to all other policies:* $\pi_\star \geq \pi, \forall \pi$.
- *All optimal policies achieve the optimal value function, that is,* $v_{\pi_\star}(s) = v_\star(s)$.
- *All optimal policies achieve the optimal action-value function, that is,* $Q_{\pi_\star}(s, a) = Q_\star(s, a)$.

It can be proven that we can write the optimal action-value function in terms of the optimal state-value function (Sutton & Barto 1998), like so:

$$Q_\star(s, a) = \mathbb{E}\left[R_{t+1} + \gamma v_\star(S_{t+1})|S_t = s, A_t = a\right], \tag{2.17}$$

Furthermore, $v_\star$ must satisfy Equation 2.12, and since it's the optimal value function, it should be able to be written without specifying a policy in particular. This is called the ***Bellman optimality equation***,

and it basically says that the value of a state under an optimal policy must be equal to the expected return for the best action from that state, that is:

$$\begin{aligned} v_\star(s) &= \max_{a \in \mathcal{A}(s)} Q_{\pi_\star}(s,a) \\ &= \max_a \mathbb{E}_{\pi_\star}\left[G_t | S_t = s, A_t = a\right] \\ &= \max_a \mathbb{E}_{\pi_\star}\left[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a\right] \\ &= \max_a \mathbb{E}\left[R_{t+1} + \gamma v_\star(S_{t+1}) | S_t = s, A_t = a\right] \end{aligned} \tag{2.18}$$

Likewise, the Bellman Optimality equation for $Q_\star$ is:

$$Q_\star(s,a) = \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} Q_\star(S_{t+1}, a') \Big| S_t = s, A_t = a\right] \tag{2.19}$$

In conclusion, in order to obtain the optimal policy $\pi_\star$, we simply maximize over $Q_\star$:

$$\pi_\star(a|s) = \begin{cases} 1, & \text{if } a = \arg\max_{a \in \mathcal{A}(s)} Q_\star(s,a) \\ 0, & \text{otherwise} \end{cases} \tag{2.20}$$

or, if our policy is deterministic:

$$\pi_\star(s) = \arg\max_{a \in \mathcal{A}(s)} Q_\star(s,a) \tag{2.21}$$

We can do the same but using the state-value function. Thus, the easiest way to learn the optimal policy is to learn the optimal value function first.

Equations 2.18 and 2.19 represent systems of $|\mathcal{S}|$ equations which are nonlinear (due to the max) and, sadly, there does not exist a closed form solution for them. If we know the dynamics of the environment $p(s', r|s, a)$, then we should be able to solve the Bellman Optimality Equations. However, this is not always the case, and calculating these probabilities would be another computation altogether, adding more and more complexities for our solution.

Thus, we seek to avoid calculating these probabilities altogether, along with the transition probabilities. For this there are several types of iterative solution methods, but we won't see them all here. On the next section, we will focus on Q-learning, but other algorithms include Policy Iteration, Value Iteration, Sarsa, and Temporal-Difference. We refer the reader to (Sutton & Barto 1998), (Szepesvári 2009), (Gosavi 2015) and (Watkins & Dayan 1992) for the full account of these and many other algorithms.

2.5 Q-Learning

A foreword must be made before delving into this type of algorithm. When developing a learning agent, there is usually a distinction between two phases, the ***learning phase*** and the ***deployment phase***. In the former, the agent will try different actions available and will infer a good (or good enough) policy for the environment that it will follow later during the deployment phase.

Another point to make is that in Reinforcement Learning, there is a distinction between what is called an ***estimation policy*** and a ***behaviour policy***. The former is the policy that is used to generate behavior to the agent, while the latter is the policy that is evaluated and improved upon. If they are the same policy, then the Reinforcement Learning algorithm is said to be ***on-policy*** while if they are different, then the algorithm is said to be ***off-policy***.

Finally, one can claim there are two types of Reinforcement Learning: online and offline. ***Online (reinforcement) learning*** basically means that the agent is performing the training as the data comes in, whereas in ***offline (reinforcement) learning*** one has a static dataset to train our agent. More formally, in online reinforcement learning one measures the performance of the behavior policy, and in offline reinforcement learning the agent learns a policy during the learning phase and then measure the learned policy's performance during deployment (application).

2.5.1 Exploration versus Exploitation Dilemma

In online learning, the agent must make a choice: to continue exploring more actions to find better policies or to exploit the known actions that have the best-known results. This is known as the ***exploration versus exploitation dilemma***. Maybe a known Chess strategy might lead us into sacrificing some of our pieces, but it might lure our opponent into a trap and hence we would win the game. In other words: the best long-term strategy may involve short-term sacrifices (Silver 2015).

The question is *when* to act with the best-known action (***exploitation***) and when to try new ones (***exploration***). If our agent always chooses the best-known action, i.e. the one that produces the largest rewards, then we call these types of policies ***greedy*** which exploit the current knowledge of the values of the available actions. The fault in these is that there are no resources spent trying to find better policies; indeed, the agent might be stuck in a local optimum and not even realize it.

A typical solution is for our agent to randomly select from all the possible actions available with a probability of ϵ, and with probability $1 - \epsilon$ the agent should behave greedily, with $0 < \epsilon < 1$. We call these methods ***ϵ-greedy*** and these ensure us that our agents will visit all of the states, and hence assure convergence *in the limit* to our optimal value functions.

Finally, in order to increase the certainty with which our agent takes actions, one common thing to do is to gradually decrease the value of ϵ. This will result in a highly explorative behavior from our agent at the start and a highly exploitative behavior at the end. This method is known as ϵ ***decay***.

2.5.2 Q-Learning Algorithm

Q-Learning (Watkins 1989) is a model-free algorithm. It has similarities to other class of methods called Temporal Difference (TD) Methods found in (Sutton & Barto 1998), some of which include TD(0), TD(λ) and Sarsa. Q-Learning works as follows: an agent tries an action a at a particular state s and evaluates its consequences in terms of the immediate reward it receives, as well as its *estimate* of the value of the next state s' to which it arrives. It will thus try all actions repeatedly and learn which is best with respect to the long-term discounted reward.

The ***Q-Learning algorithm*** will thus seek to approximate $Q_\star$, Equation 2.17, via solving the Bellman's equation, Equation 2.13 iteratively like so: the agent's experience is divided into episodes in which, for each episode, the agent observes its current state $S_t = s$, selects and performs an action $A_t = a$, observes the subsequent state $S_{t+1} = s'$, receives an immediate reward $R_{t+1} = r$, and finally adjusts its Q values using a learning factor α_t according to the following:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha_t \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right] \tag{2.22}$$

which has the general form:

$$NewEstimate \leftarrow OldEstimate + StepSize\,[Target - OldEstimate] \tag{2.23}$$

We assume the initial Q values are given for all actions and states. If high enough initial values are given, then they are called ***optimistic initial values*** and are a way to encourage exploration; they may outperform e.g. $\epsilon-$greedy methods (Sutton & Barto 1998).

For fully deterministic environments, it suffices to have $\alpha_t = 1, \forall t$. However, for stochastic environments, we need to use the following theorem which sets the conditions for the convergence of the Q-Learning algorithm:

Theorem 2.2 (Robbins-Monro sequence of time steps) *Given bounded rewards $R_t \leq \Re < +\infty$ and learning rates $0 \leq \alpha_t < 1$ satisfying*

$$\sum_{t=1}^{+\infty} \alpha_t = +\infty \qquad\qquad \sum_{t=1}^{+\infty} \alpha_t^2 < +\infty,$$

then $Q(s,a) \to Q_\star(s,a)$ as $t \to +\infty, \forall s, a$, with probability 1, with Q given by Equation 2.22.

Some examples of the learning rate include $\alpha_t = \frac{A}{B+t}$ and $\alpha_t = \log(t)/t$. The convergence proof is found in (Watkins & Dayan 1992) from where we took this theorem from and adapted it for our notation. Note that this will be a model-free solution, given that we avoid completely the transition probabilities. The pseudocode for Q-learning can be found in Appendix A.

2.5.3 *n*-step Q-Learning with Function Approximation

For value-based model-free reinforcement learning methods such as Q-Learning, the action-value function can be represented using a function approximator such as linear or nonlinear functions. This is necessary when the state-action space is large (Szepesvári 2009). For example, we can use neural networks (NN). Thus, we use $Q(s, a; \theta)$ to represent the approximate action-value function with parameters θ and the updates to this parameter will be derived from Q-Learning, i.e., $Q_\star(s, a) \approx Q(s, a; \theta)$.

The parameters θ are learned by iteratively minimizing the following sequence of loss functions:

$$L_i(\theta_i) = \mathbb{E}\left(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i)\right)^2, \tag{2.24}$$

with the same notation as before. Using the Q-Learning algorithm (Equation 2.22) and this loss function (Equation 2.24) is referred to as ***one-step Q-Learning***. This is due to it updating the action-value function towards the one-step return $r + \gamma \max_{a'} Q(s', a'; \theta)$ (remember the general form in Equation 2.23). Note that this will make the learning process slow, as the only state-action pairs are the ones that led to this reward in particular. Thus, all other action-value pairs will only be indirectly changed, which might lead to slower calculations.

One way of accelerating calculations is by using ***n-step returns*** which is defined as $R_t + \gamma R_{t+1} + \cdots + \gamma^{n-1} R_{t+n-1} + \gamma^n \max_a Q(S_t, a)$. This will be our new target value towards which our action-value function will be updated. Using this, along with our Q-Learning algorithm, results in what is called ***n-step Q-Learning***. This has the potential to make our updates more efficient, as changes will be propagated $n-$step backwards each iteration. Reasonable values of n must be used as there is a higher variance and this might put the convergence at risk (Mnih et al. 2016).

2.6 Notable Applications of Reinforcement Learning

An interesting example of an application of Reinforcement Learning was done by a team at Stanford, where they succeeded in creating an autonomous RC helicopter that did several aerobatic maneuvers: tic-tocs, loops, hurricane, forward flip and sideways roll at low speed, tail-in funnel and nose-in funnel, as well as an airshow, which requires a transition between each of these maneuvers (Abbeel et al. 2007, Coates et al. 2008). This is particularly impressive given that autonomous helicopter flying is a highly challenging control problem, and this is even more clear in the reward function that they used: it contained 24 features (including squared error state variables, squared inputs, squared change of inputs between consecutive time step, and the squared integral of the error state variables), then they proceeded to use Apprenticeship Learning via Inverse Reinforcement Learning algorithm on a professional pilot flying the RC helicopter. This provided them with the reward weights and hand-chose the ones that brought them closer to how the expert demonstration went (Figure 2.4).

Figure 2.4 The RC helicopter doing a pirouetting stall turn. Adapted from the accompanying videos of (Abbeel et al. 2007, Coates et al. 2008).

One of the most recent *in the know* examples of Reinforcement Learning in the news is that of the team at Google's DeepMind beating the world champions in Go (Silver et al. 2016). Go is a particularly difficult environment to set a Reinforcement Learning agent in: you may solve each game by recursively computing the optimal value function (ref. Section 2.4) in a search tree containing approximately 250^{150} possible sequences of moves (the number of possible moves is around 250 and the games last for around 150 moves). It has been estimated that the total number of possible games of Go is around 10^{761} (ref. Figure 2.5).

With this in mind, it is folly to do an exhaustive tree search for every possible game the machine is playing. DeepMind's solution to this is by using Supervised Learning on expert human moves to train a policy network p_σ. At the same time, they train a fast policy p_π that can rapidly sample actions during

rollouts. Thirdly, they train a Reinforcement Learning policy p_ρ that improves the policy p_σ by optimizing the final outcome of the games by playing itself. Lastly, they train a value network v_θ that predicts the winner of games played by the policy p_ρ network against itself. Their program AlphaGo combines the policy and value networks via Monte Carlo Tree Search (Silver et al. 2016).

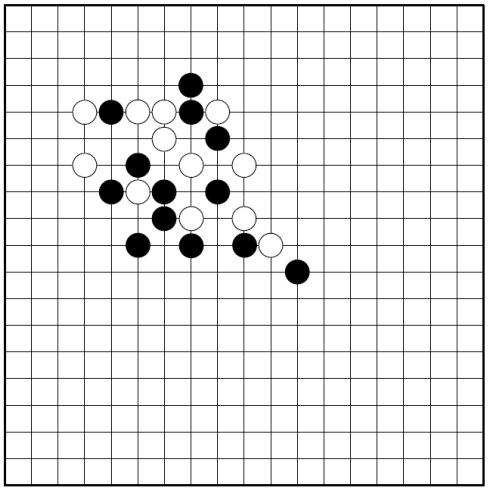


Figure 2.5 A typical configuration of the board game Go. Reproduced, with permission, from (Sutton & Barto 1998).

Lastly, in the last month (August 2017), the team at OpenAI has created and tested a bot that defeats the best players in the videogame Dota 2 (OpenAI 2017*a*). They have managed to beat the best $1v1$ players during *The International*, the pro stage for Dota 2 where teams compete for large cash prizes. Specific details of the implementation of this bot are scarce, but from what is released, we know that the bot learned by playing an exact copy of itself during two weeks of training, without using imitation learning or tree search.

Dota 2 is a complex game itself, mainly due to much of the information is hidden: the agent does not know where are its opponents, or the items and monsters that are scattered throughout the map. Because of this, the agent must learn some sort of intuition to plan, attack, trick, and deceive its opponents.

As such, it is extremely interesting how it has come up with the same strategies that many Dota 2 players (particularly professional players) also use, like using the other player's in-game bots to achieve its goals, dodging the attacks of the other player, improvise when encountering unfamiliar situations, predicting where the enemy player will be in order to know where to attack, among others (ref. Figure 2.6). The OpenAI team's next steps will be to play a full match of $5v5$, that is, 5 bots versus 5 human players, where each player can choose from a hundred heroes and hundreds of items for weapons, equipment, and health and mana regeneration (OpenAI 2017*a*,*d*).

Figure 2.6 The OpenAI bot doing the maneuver Creep Blocking, where it zigzags in front of the other smaller in-game bots (creeps) in order to slow them down and better control their position. Adapted from the accompanying videos of (OpenAI 2017*a*).

From these examples, it can be inferred that not one algorithm is sufficient to overcome a particular task at hand, whether it is playing a particularly complex game or to maneuver a helicopter. A combination of different algorithms is needed to overcome the complex world that the agent is positioned in, and this is most likely what the future of training self-driving cars via Reinforcement Learning will hold.

Chapter 3

Asynchronous Advantage Actor-Critic (A3C)

Google's DeepMind group released in 2016 an algorithm that has achieved state-of-the-art results. Its name is Asynchronous Advantage Actor-Critic, or A3C as they refer to it (Mnih et al. 2016). It works on both discrete and continuous action spaces, and has become the algorithm to use whenever encountering Reinforcement Learning problems with complex action and state spaces.

Since the previous state-of-the-art algorithm was released, the Deep Q-Network or DQN (Mnih et al. 2015), the importance of the A3C comes from its use of fewer resources, that it was more robust, and was able to achieve better scores in the Atari 2600 games used to test the DQN. Mainly, it no longer relied on the use of a GPU to train its agents, only on the CPU. This is due to its architecture, as we will see in the following sections, but this was the key point on why it was selected for this project.

We will divide the long name of this algorithm to explain how it works, starting with the Actor Critic Methods, the main architecture that it is based on, then moving on to the Advantage function, and finally to the asynchronous part. We will finish by explaining how these all come together into the A3C algorithm. The reader can find the pseudocode for each algorithm as they were presented in the original paper in Appendix A.

3.1 Actor-Critic Methods

So far we have seen in Section 2.4.1 Value-based models for solving our Reinforcement Learning problems. There is another group of models that deal directly with improving policies and are named accordingly: these are the ***policy-valued models***. Specifically, these will be methods that learn a parametrized policy that will select actions without the need of using a value function.

We call the ***policy weight (parameter) vectors*** the primary learned weight vector $\boldsymbol{\theta} \in \mathbb{R}^n$, so we will rewrite the policy as the probability of taking an action a given that the agent is in state s with weight vector $\boldsymbol{\theta}$ at time t:

$$\pi(a|s;\boldsymbol{\theta}) = \mathbb{P}\left[A_t = a|S_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta}\right] \doteq \pi_{\boldsymbol{\theta}}(a|s) \tag{3.1}$$

The actor-critic algorithm can be seen as a hybrid between the policy method and the value function method (ref. Figure 3.1a). Thus, the policy function will be known as the ***actor*** and the value function will be known as the ***critic***. The actor will produce an action a, given the current state s, and the critic will produce a signal to essentially criticize these actions taken by the actor. Thus actor-critic methods are temporal difference (TD) methods. Learning is made on-policy as the critic will need to learn and critique the actions taken by the agent using the current policy (ref. Figure 3.1b). This critique will take the form of the so-called ***Temporal Difference (TD) error***, defined as:

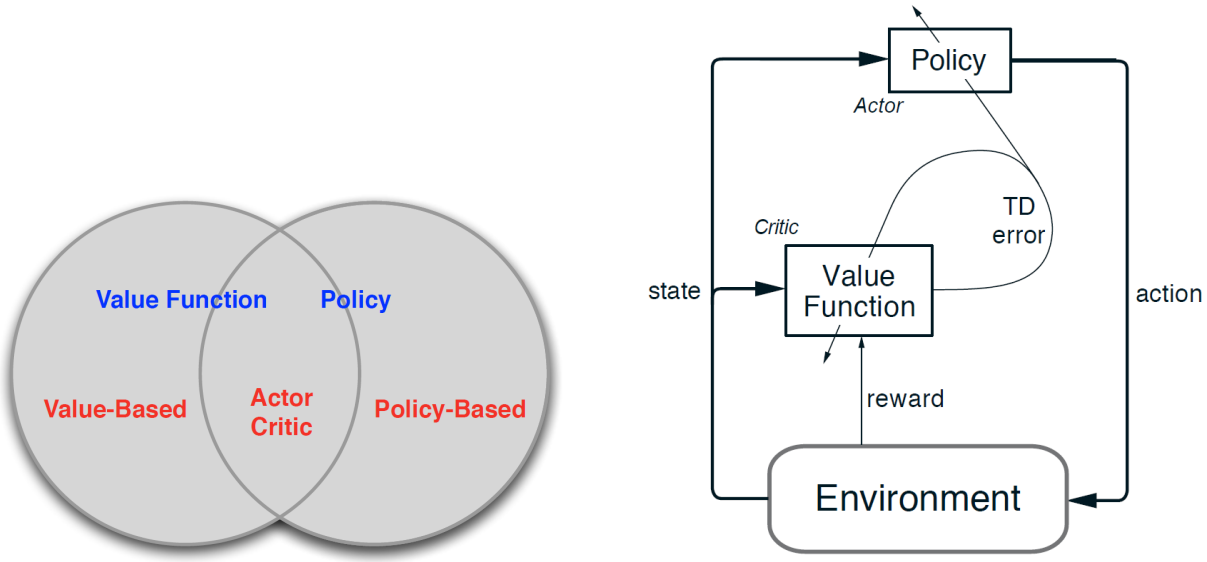
$$\delta_t \doteq R_{t+1} + \gamma V_t(S_{t+1}) - V(S_t) \tag{3.2}$$

where V_t is the value function implemented by the critic at time t. The TD error will indicate whether or not things have improved or gotten worse, i.e., we should select the action A_t at time t more if the TD error is positive, and less if it's negative. We call the ***parametrized numerical preferences*** as the parametrization

we form for each state-actor pairs. We denote these by h, with $h : \mathcal{S} \times \mathcal{A}(S_t) \times \mathbb{R}^n \to \mathbb{R}$. What we wish to do with these is that the most preferred actions in each state are given the highest probability of being selected, e.g. using the Gibbs softmax distribution:

$$\pi(a|s;\boldsymbol{\theta}) = \frac{\exp h(s,a;\boldsymbol{\theta})}{\sum_{b\in\mathcal{A}(s)} \exp h(s,b;\boldsymbol{\theta})} \tag{3.3}$$

which has the advantage that the approximate policy can approach determinism. As a reminder of $\epsilon-$greedy methods, there is a probability of selecting a random action which might become troublesome. Other advantages of policy-based RL are its better convergence properties, that they are effective in high-dimensional or continuous action spaces, and that they can learn stochastic policies, which the action-value methods cannot do. The disadvantages are that they typically converge to a local optimum rather to the global optimum as we will see in Chapter 5 and that evaluating a policy is typically inefficient and has a high variance (Silver 2015, Mnih et al. 2016).



(a) (b)

Figure 3.1 a) Actor-critic algorithms can be seen as a hybrid of value-based and policy-based methods. Reproduced, with permission, from (Silver 2015). b) Architecture of the actor-critic algorithm. The TD error is the sole output of the critic, driving all learning in the actor and the critic. Reproduced, with permission, from (Sutton & Barto 1998).

Thus policy-based model-free methods will parametrize the policy π and update the parameter (weight) $\boldsymbol{\theta}$ by performing gradient ascent on $\mathbb{E}[G_t]$. This is done with a function approximator, such as linear functions or Neural Networks (NN). In other words, we will start with a policy, we will evaluate its value function and then move on to a better policy (Szepesvári 2009). Note that the parametrization $\pi(a|s;\boldsymbol{\theta})$ can be done in any way as long as $\nabla_{\boldsymbol{\theta}}\pi(a|s;\boldsymbol{\theta})$ exists and is always finite.

REINFORCE methods (Williams 1992), for example, update $\boldsymbol{\theta}$ using the gradient $\nabla_{\boldsymbol{\theta}} \log \pi(A_t|S_t;\boldsymbol{\theta})G_t$, which in turn is an unbiased estimator of $g = \nabla_{\boldsymbol{\theta}}\mathbb{E}[G_t;\boldsymbol{\theta}]$ (Shalev-Shwartz et al. 2016) thanks to the Policy Gradient Theorem (Sutton & Barto 1998, Schulman, Moritz, Levine, Jordan & Abbeel 2015).

The downside of using this is the high variance and inefficiency (Silver 2015), but we can do something about the former: we can reduce it by subtracting a ***baseline*** $b_t(S_t)$, a learned function of the state, to the estimator. Thus, we will use the gradient $\nabla_{\boldsymbol{\theta}} \log \pi(A_t|S_t;\boldsymbol{\theta})(G_t - b_t(S_t))$. A common baseline to use is an estimate of the value function, that is, $b_t(S_t) \approx v_\pi(S_t)$.

3.2 Advantage Function

We now define the ***advantage function***. It will measure whether or not the action is better or worse than the policy's default behavior (Schulman, Moritz, Levine, Jordan & Abbeel 2015). We denote it by $A : \mathcal{S} \times \mathcal{A}(S_t) \to \mathbb{R}$ and is defined as:

$$A_\pi(s, a) \doteq Q_\pi(s, a) - v_\pi(s) \tag{3.4}$$

We can then see the quantity $G_t - b_t$ on the gradient as an estimate of the advantage function we just defined. Indeed, we can take G_t as an estimate of $Q_\pi(s, a)$ and as discussed previously, we will take b_t as an estimate of $v_\pi(S_t)$. Taking all of this into consideration, we conclude that this approach can be seen as having an actor-critic architecture, with the policy π as the *actor* and the baseline b_t as the *critic* (Mnih et al. 2016).

Finally, the TD error defined in Equation 3.2 can be used as an estimator of the advantage function. Indeed, it is an unbiased estimator:

$$\mathbb{E}_{\pi_{\boldsymbol{\theta}}} \left[\delta_{t,\pi_{\boldsymbol{\theta}}} | s, a\right] = \mathbb{E}_{\pi_{\boldsymbol{\theta}}} \left[R_t + \gamma v_{t,\pi_{\boldsymbol{\theta}}}(s') | s, a\right] - v_{\pi_{\boldsymbol{\theta}}}(s) = Q_{\pi_{\boldsymbol{\theta}}}(s, a) - v_{\pi_{\boldsymbol{\theta}}}(s) = A_{\pi_{\boldsymbol{\theta}}}(s, a)$$

3.3 Asynchronous Reinforcement Learning Framework

The point of having asynchronous versions of many value-based models is to find algorithms that can reliably train Deep Neural Networks (DNN) policies without large resources. The authors realized and used two ideas to make their algorithms practical:

1. Use asynchronous learners via multiple CPU threads on a single machine (saving resources).

2. Observe that multiple actors-learners running in parallel will be exploring different parts of the environment, which make the updates being done by the actors to be less correlated in time than e.g. a single actor applying online updates.

The second point is important, given that it tells us that asynchronous frameworks bypass the need for an experience replay that DQN (Mnih et al. 2015) uses by using multiple agents to explore the environment in parallel. As a note, experience replay is a windowed buffer of the last N transitions ($N = 10^6$ in the original paper), so instead of updating from the last transition, the agents store it in the experience replay and update from a batch of randomly sampled transitions of the same experience replay.

Besides stabilizing learning, using multiple parallel actor-learners helps obtain a reduction in training time that is roughly linear in the number of parallel actor-learners. Also, given that no experience replay is used, on-policy Reinforcement Learning methods can be used such as the actor-critic, aiding in the stability of our training of the neural networks.

Thus, for ***asynchronous one-step Q-Learning***, each thread interacts with its own copy of the environment and computes a gradient of the Q-Learning loss at each time step. The gradients are accumulated over multiple minibatches, reducing the chance of multiple actor-learners overwriting each other's updates. To make the exploration policies differ, the authors used $\epsilon-$greedy exploration with ϵ periodically sampled from some distribution by each thread. Pseudocode for this algorithm can be found in Appendix A.

For ***asynchronous one-step Sarsa***, the same is applied as in asynchronous one-step Q-Learning, except that the target value for $Q(s, a)$ is now $r + \gamma Q(s', a'; \boldsymbol{\theta}^-)$, a' being the action taken in state s'.

For ***asynchronous n-step Q-Learning***, the algorithm performs in the forward view by explicitly computing the n-step returns. This is easier when training NN with momentum-based methods (ref. Section 3.4.3) and backpropagation through time. The algorithm will select actions using its exploration policy for up to t_{max} steps or until a terminal state is reached, hence it will receive up to t_{max} rewards from the environment since its last update.

Finally, the algorithm will compute the gradients for n-step Q-Learning updates for each state-action pairs that have been encountered since the last update. Each of these n-step updates uses the longest possible n-step return and the accumulated updates are applied in a single gradient step. Pseudocode for this algorithm can be found in Appendix A.

3.4 Asynchronous Advantage Actor-Critic

We conclude by joining all the past sections into one, the A3C algorithm. But first, we must define some final terms to add both to the loss functions, as well as how to optimize these gradients that we will be calculating (and specifically, the gradients of *what*).

3.4.1 Entropy

The ***entropy*** of the policy π corresponds to the spread of the action probabilities that the policy will calculate. Its purpose is to push the learning process into a random selection, but only slightly so and whenever the model doesn't know how to act optimally, i.e. it is added to balance exploration and exploitation by discouraging premature convergence to sub-optimal deterministic policies. If the policy suggests a single action with a large probability, then the entropy should be low, and if the policy suggests many actions with low probabilities, then the entropy will be high (Pereyra et al. 2017). We define it as follows:

$$H(\pi(S_t;\boldsymbol{\theta})) = -\sum_a \pi(a|S_t;\boldsymbol{\theta}) \log \pi(a|S_t;\boldsymbol{\theta}) \tag{3.5}$$

Thus if we have a fully deterministic policy, the entropy will be zero, whereas a uniformly distributed policy will maximize it. Or conversely, higher values of entropy imply a more spread distribution, and lower values imply less spread. Measuring the entropy will tell us how sure the agent is of the actions it is taking, or of the policy it has learned and implemented. By adding this term, we prevent our model from being too deterministic, but it is also helpful on tasks requiring hierarchical behavior (Mnih et al. 2016, Lapan 2017).

3.4.2 Loss Functions and Gradients

There will be, effectively, two loss functions associated with the DNN outputs: the policy and value loss functions. We will consider the entropy term as a part of policy function since it favors exploration during the training process. Therefore we have the policy loss:

$$L_\pi(\boldsymbol{\theta}') \doteq \log \pi(A_t|S_t;\boldsymbol{\theta}')(G_t - v(S_t;\boldsymbol{\theta}_v)) + \beta H(\pi(S_t;\boldsymbol{\theta}')) \tag{3.6}$$

where β is a hyperparameter that controls the strength of the entropy regularization term and $\boldsymbol{\theta}_v$ are the parameters of the value function. Meanwhile, the value loss function is defined as:

$$L_v(\boldsymbol{\theta}) \doteq (G_t - v(S_t;\boldsymbol{\theta})^2 \tag{3.7}$$

Hence, we will perform the following respective updates to the parameters: for $\boldsymbol{\theta}$:

$$\partial\boldsymbol{\theta} \leftarrow \partial\boldsymbol{\theta} + \nabla_{\boldsymbol{\theta}'} \log \pi(A_i|S_i;\boldsymbol{\theta}')(G_i - v(S_i;\boldsymbol{\theta}_v) + \beta \nabla_{\boldsymbol{\theta}'} H(\pi(S_i;\boldsymbol{\theta}')) \tag{3.8}$$

where $\boldsymbol{\theta}'$ are thread-specific parameters which we synchronize with the global parameters at the beginning of each thread start. The first part tells us how to improve our policy: we push the probability of the action taken towards the total reward minus the predicted value. If we were pessimistic and got a good outcome, we should perform the action more often; if we were optimistic and got a bad outcome, we should avoid it in the future (Lapan 2017).

We perform the updates to $\boldsymbol{\theta}_v$ like so:

$$\partial\boldsymbol{\theta}_v \leftarrow \partial\boldsymbol{\theta}_v + \frac{\partial(G_i - v(S_i;\boldsymbol{\theta}'_v))^2}{\partial\boldsymbol{\theta}'_v} \tag{3.9}$$

where again $\boldsymbol{\theta}'_v$ are thread-specific parameters which we synchronize with the global parameters at the beginning of each thread start. Basically, we do an MSE update on the value function parameter updates.

3.4.3 Gradient Descent Optimization

The authors investigated two different gradient descent optimization algorithms for the asynchronous framework. These where Momentum Stochastic Gradient Descent (SGD), RMSProp (Tieleman & Hinton 2012) without shared statistics, and RMSProp with shared statistics (gradients) g. They found that the latter is more robust than the other two methods.

The standard non-centered RMSProp update is given by the following:

$$g = \alpha g + (1 - \alpha)\Delta\theta^2 \tag{3.10}$$

$$\theta \leftarrow \theta - \eta \frac{\Delta\theta}{\sqrt{g + \epsilon}} \tag{3.11}$$

where α is the RMSProp decaying factor, η is the learning rate, g are the shared statistics, and ϵ is the exploration rate that is added so as to avoid division by zero. All of these operations will be performed elementwise, and the vector g is shared among threads and updated asynchronously (this will reduce the memory requirements).

3.4.4 A3C

In conclusion, the A3C algorithm will maintain a policy $\pi(A_t|S_t;\theta)$ and an estimate of the value function $v(S_t;\boldsymbol{\theta})$. As in Section 3.3, it will use the n-step returns previously defined to update both the policy and value function. These will be updated every t_{max} actions (or when a terminal state is reached).

As seen in the previous subsection, the updates performed by the A3C can be seen as in Equations 3.8 and 3.9: $\nabla_{\boldsymbol{\theta}'} \log \pi(A_t|S_t;\boldsymbol{\theta}')A(S_t, A_t;\boldsymbol{\theta},\boldsymbol{\theta}_v) + \beta\nabla_{\boldsymbol{\theta}'} H(\pi(S_t;\boldsymbol{\theta}'))$, where $A(S_t, A_t;\boldsymbol{\theta},\boldsymbol{\theta}_v)$ is an estimate of the advantage function. Thus:

$$A(S_t, A_t;\boldsymbol{\theta},\boldsymbol{\theta}_v) = \sum_{i=0}^{k-1} \gamma^i R_{t+i} + \gamma^k v(S_{t+k};\boldsymbol{\theta}_v) - v(S_t;\boldsymbol{\theta}_v)$$

where k can vary from state to state and is upper-bounded by t_{max}. We will rely on parallel actor-learners and accumulated updates for improving the training stability. In practice, the parameters of the policy and value functions are shared, even if before we took them as different. The architecture for the A3C can be seen in Figure 3.2 and its pseudocode can be found in Appendix A.

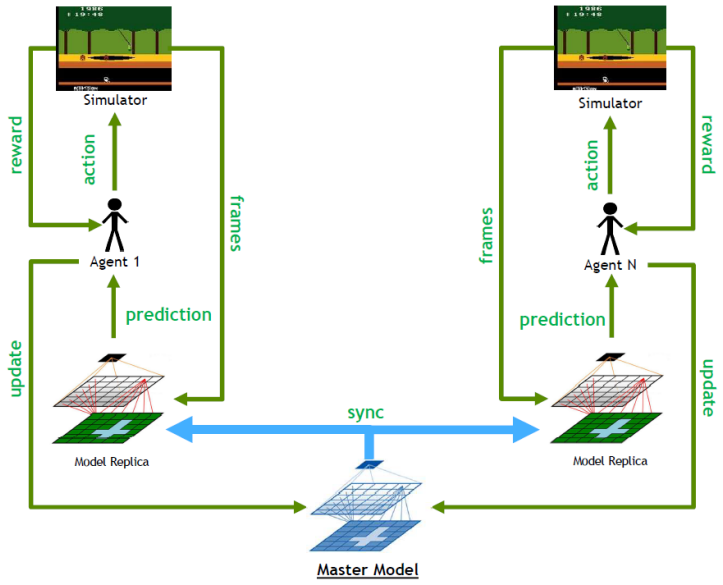


Figure 3.2 Architecture of the A3C algorithm. The agents act concurrently and have replicas of the model. Reproduced, with permission, from (Babaeizadeh et al. 2017).

The final, perhaps key piece of information in this is that the main reason for using a CPU and not a vanilla GPU implementation on the architecture of the A3C is due to the sequential nature of Reinforcement Learning in general, but in particular for A3C. In Reinforcement Learning, the training data are generated whilst learning, meaning that the training and inference batches are *small* and thus a GPU would be for the most part idle during the training, waiting for new training data to arrive. Given that the A3C does not use replay memory, it is completely sequential and thus a CPU implementation is as fast as a vanilla GPU implementation (Babaeizadeh et al. 2016).

3.4.5 Neural Network, Hyperparameters, and Results

Each experiment done in (Mnih et al. 2016) used 16 actor-learner threads running on a single machine and without using GPUs (16 agents on a 16-core CPU). The methods performed updates every 5 actions, i.e., $t_{max} = 5$ and $I_{Update} = 5$, and a shared RMSProp was used for optimization (as discussed in Section 3.4.3). After each update, the central server propagates new weights to the agents to guarantee they share a common policy (Babaeizadeh et al. 2016). The authors tested the three asynchronous value-based algorithms discussed in Section 3.3 using a shared target network that was updated every 4×10^4 frames.

The preprocessing of the image was done as in (Mnih et al. 2015), with an action repeat of 4 (i.e., Deterministic environments; see Section 5.3): for Atari 2600 games, the windows have dimensions 210×160 pixels with a 128 color palette. We reduce this high dimensionality by first converting the RGB images to grayscale and downsampling to a 110×84 image. The final representation is obtained by cropping an area of the image and obtaining a 84×84 region (capturing the playing area of course).

So, in (Mnih et al. 2016), the authors used in their setup a single DNN to approximate both the policy and value function. This DNN has two convolutional layers: the first one has 16 filters of size 8×8 with a stride of 4, followed by the second convolutional layer with 32 filters of size 4×4 with a stride of 2. These were followed by a fully connected layer with 256 hidden units. Each hidden layer is followed by a ReLu defined as $f(x) = \max(0, x)$. For value-based methods, the output was a single linear unit for each action representing the action-value. For actor-critics, the two outputs are a softmax layer (defined in Equation 3.3) which approximates the policy function $\pi(A_t|S_t;\boldsymbol{\theta})$ and a linear layer to output an estimate of $v(S_t;\boldsymbol{\theta})$, with all non-output layers shared. Another version was tested, named A3C LSTM, by adding 256 LSTM (Long-Short Term Memory) cells after the final hidden layer.

The experiments used a discount factor of $\gamma = 0.99$ a RMSProp decay factor of $\alpha = 0.99$, and the exploration rate ϵ was sampled from a distribution taking three values $\epsilon_1, \epsilon_2, \epsilon_3$ with respective probabilities $0.4, 0.3, 0.3$. The values of these were annealed from 1 to 0.1, 0.01, 0.5 respectively over the first 4×10^6 frames. The entropy regularization weight $\beta = 0.01$ for all Atari 2600 and TORCS experiments. The initial learning rate was sampled from a $LogUniform(10^{-4}, 10^{-2})$ distribution and annealed to 0 over the course of the training. The authors also applied the A3C algorithm to the MuJoCo environment (ref. Section 4.1), but we won't explore these environments, though the ideas applied to these continuous environments will prove to be useful for the future environments we will use (ref. Section 7.1).

The authors compared the results using the A3C algorithm to the Atari environments to other previous state-of-the-art algorithms, such as DQN, Gorila, Dueling Double DQN, among others (exploring all of these algorithms is beyond the scope of this project). We present these in the same format as the authors presented it, in Table 3.1. All algorithms trained on a GPU environment were trained in a Nvidia Tesla K40 GPU. Perhaps it suffices to note that 1 day of training the A3C on a CPU reaches the level of a Double DQN, saving not only 7 days of training but the computational use of the GPUs. We reserve the discussion of the results in the TORCS environment for Section 6.1.

Table 3.1 Results of the A3C algorithm compared to other state-of-the-art algorithms. The results are normalized with respect to a human player on 57 Atari games. The human starts where used as the evaluation metric. Reproduced from (Mnih et al. 2016).

Method	Training Time	Mean	Median
DQN	8 days on GPU	121.9%	47.5%
Gorila	4 days, 100 machines	215.2%	71.3%
D-DQN	8 days on GPU	332.9%	110.9%
Dueling D-DQN	8 days on GPU	343.8%	117.1%
Prioritized DQN	8 days on GPU	463.6%	127.6%
A3C, FF	1 day on CPU	344.1%	68.2%
A3C, FF	4 days on CPU	496.8%	116.6%
A3C, LSTM	4 days on CPU	623.0%	112.6%

Chapter 4

Environments

In this section, we will present the different environments we have used and will plan to use in order to test the algorithm A3C (Section 3). Two of these software tools are Gym and Universe which were built by OpenAI, a non-profit AI research company whose mission is to build safe Artificial General Intelligence (AGI), and ensure that AGI's benefits are as evenly distributed as possible. They believe AGI is going to be the most significant technology ever created by humans, and hence why they open-source their software tools, as well as publish via blog posts or research papers their contributions.

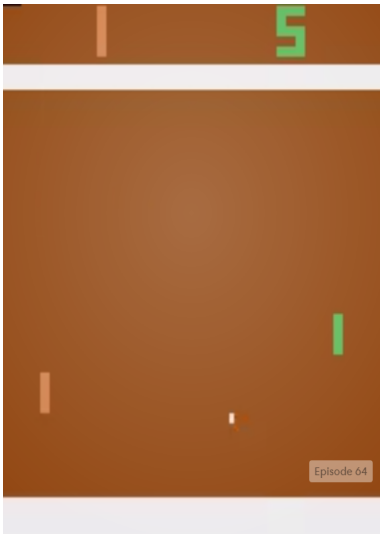
The third and last software environment we plan to use is Unity, where we will use the open-sourced distribution of Udacity's self-driving simulator. We present these more in depth in the following sections.

4.1 Gym

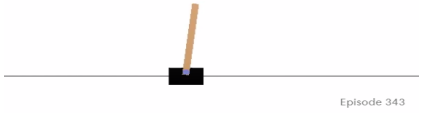
Gym (Brockman et al. 2016) is a toolkit for developing and comparing reinforcement learning algorithms. The purpose is to facilitate this process: OpenAI's team develop and provide the environment, and the user provides the algorithm, usually implemented using either Tensorflow or Theano. Gym also lets the user upload their results on each environment and evaluate them, to see if the agent has indeed solved the environment (on the specifications that each has on their respective descriptions). Gym supports teaching agents in many environments, which are separated into the following 13 different types:

- **Classic Control** Classic control problems from RL literature. There are several available, but perhaps the most widely known of these is the Cart Pole problem, where the agent must balance a pole on top of the cart, which moves horizontally on a frictionless track.
- **Algorithmic** The agent learns to imitate computations, i.e., reversing a text, copying a text, learn to add two or three multi-digit numbers, among others.
- **Atari** Atari 2600 games. What the agent sees (the input) can be either the screen images or the RAM. Famous games such as Pong, Ms Pacman, Breakout, and Asteroids can be found. Note, however, that every time an action is taken by the agent, it is performed for a duration of k frames, where $k \sim \mathcal{U}(\{2, 3, 4\})$.
- **Board Games** Play classic board games. Currently available are Go 9×9, Go 19×19 and Hex 9×9.
- **Box2D** The agent learns continuous control tasks in the Box2D simulator. Known environments include the Bipedal Walker (which has two versions, the normal and the *Hardcore*), the Lunar Lander and the Car Racing. Most of these are experimental, specifically the one we will use mostly, which is the Car Racing environment (Figure 4.1c).
- **MuJoCo** The agent learns continuous control tasks run in a fast physics simulator. Perhaps most notable of these are the 'animals' or 'people' that consist of sticks and the agent must learn to run or walk through moving various parts of the body.
- **Parameter Tuning** The agent learns to tune parameters of costly experiments to obtain better results. For example, it can learn to select the architecture of a CNN classifier and its training parameters to obtain a higher accuracy.

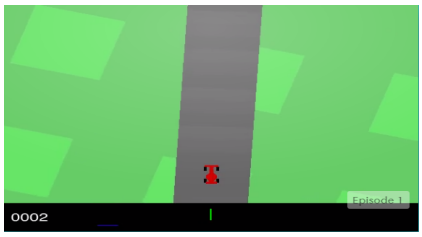
- **Toy Text** Simple text environments, perhaps the environments to start practicing with your agent. The classical environment of this type is the Frozen Lake, where your agent must find a safe path across a frozen lake towards the goal, all represented by letters.
- **Safety** Environments to test various AI safety properties, such as saying what they predict to do before acting, or the Cart Pole environment seen before, but with the variation of, while travelling more than 1 units to the right, shutting off. In other words, the goal is to design an agent that will not learn to turn left, that is, to avoid being turned off.
- **Minecraft** Minecraft environments based on Malmo. There are many environments, such as in the vanilla game: survive, find gold and mine ores.
- **PyGame Learning Environment** Adapted games from the PyGame Learning Environment (PLE), such as Snake, Flappy Bird, Pixel Copter, among others.
- **Soccer** Half-field offense soccer games.
- **Doom** Doom environments based on VizDoom. There are basic ones, such as shoot an opponent at the other side of the room to get a reward, run as fast as possible to grab a vest at the other side of the room, or even play Missions 1 to 9 of the original game.



(a) Pong-v0



(b) CartPole-v0



(c) CarRacing-v0

Figure 4.1 Some of the environments available in Gym.

The advantage of these Gym environments is that they are well documented, maintained, and updated by the community, meaning that we have access to the reward function, the observation data, as well as the actions available for the agent. For example, for the Cart Pole environment (Figure 4.1b), the agent may apply a force of $+1$ or -1 to the cart. The pendulum will start upright, and the goal is to not let it fall over. The agent is given a reward of $+1$ for every time step that the pole is kept upright, and the episode ends when the pole is more than 15 degrees from the vertical, or the cart moves more than 2.4 units from the center. The version available, *CartPole-v0*, considers that the agent 'solves' the environment when obtaining an average reward of 195 out of 100 consecutive trials. Our solution to this environment, without using any complex algorithm, can be found over at `https://github.com/PDillis/CartPole-Cases`.

4.2 Universe

Universe is a software platform for measuring and training an AI's general intelligence. It does this by providing different environments, like software, games, and other applications (OpenAI 2016). While, roughly speaking, Gym provides a way to organize and manage arbitrary dynamic environments, Universe is a big set of dynamic environments which have a common interface.

The agents will be able to interact with the environments like a human normally does: with both the keyboard and mouse, as well as the pixels on the screen. Thus, Universe will allow us to train an agent into performing any task in the computer as a human would. With Universe, then, any program can be turned into a Gym environment. The goal, then, is to develop a single AI agent that can flexibly apply its past experience on Universe environments to quickly master unknown, different and hard environments. This would be a major step towards Artificial General Intelligence.

So far, more than 1000 environments have been launched, with much more to come, due to partnership and/or permission from many companies, including Valve, NVidia, EA, and Microsoft. These environments are separated into four main branches:

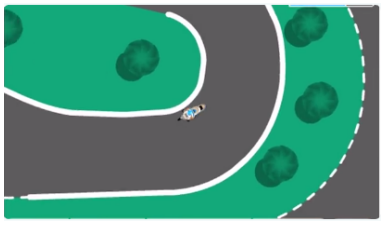
- **Flash Games** Perhaps the immediate extension of Gym, these are environments of online flash games.
- **Internet** Let's your agent play online against other players. So far only one game is available: the much-acclaimed Slither.io
- **PC Games** These environments are run on high budget AAA games from the gaming industry. As of this date, it has not been launched, but will include games like Portal, Starcraft, and Kerbal Space Program, among others.
- **World of Bits** In these environments you can test the versatility of your agent, as it must learn to interact with websites and fulfill 'normal' tasks, like filling out a form, book a flight, click specific options, etc.

It must be noted that, since the majority of the environments are for RL purposes, they differ from some found at Gym in that there is no threshold on when to decide that the environment has been solved. Therefore, whichever solution is found by our agent can be compared with a human performing the same task, and then compare the scores received with the human demonstrator.

The agents in Universe operate a remote desktop by observing pixels of a screen and producing keyboard and mouse commands. The environment exposes a VNC server and the Universe library turns the agent into a VNC client (OpenAI 2016).

For the Flash Games, which is the only environment we will be interested in Universe, the rewards are the points obtained in whichever game you are currently training in. As such, this makes things more obscure to the trainer, since the black-box increases: you can no longer tweak the rewards obtained in order to get the desired behavior you wish your agent to display. As such, it is recommended to first research each game in depth beforehand in order that it suits the particular needs of the researcher's project, if possible.

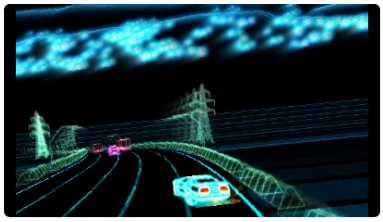
In particular, we have used the following (clicking on the name of each game will take open the respective games in the browser, should the reader wish to do so). As of turning this report, only the *v0* version of each game exists and as such are the ones we use, but other newer versions are likely to come out:



(a) **Wheelers**

(b) **DriftRunners**

(c) **CoasterRacer**



(d) **NeonRace**

(e) **DuskDrive**

Figure 4.2 Flash games we will use from the Universe set of environments.

Chapter 5

Preliminary Results

This is because I do not always do what I am told. And this is because when people tell you what to do it is usually confusing and does not make sense.

— Mark Haddon, *The Curious Incident of the Dog in the Night-time*

For our preliminary implementation of the A3C algorithm we will use one released by OpenAI called ***universe-starter-agent***. It can be found in `https://github.com/openai/universe-starter-agent`, along with its documentation and how to install and run it. It is based on *Tensorflow* (Abadi et al. 2016) and uses the Gym and Universe environments released by OpenAI. To install, it suffices to git clone it into your local directory and proceed as per the documentation online.

It is recommended for it to be installed in a virtual environment with the name of your choice. For some of the additional packages required, such as Go and OpenCV, we recommend to create the virtual environment with Anaconda (or mini Conda) as it is easier to find and install these dependencies. Not all dependencies can be installed in Windows (most important of all, the *complete* Gym and Universe environments), so as of now it only fully works in Linux or MacOS.

Once installed and activated, we run it via the following commands in a Terminal whilst in the directory where we have cloned the universe-starter-agent (and having activated our virtual environment):

```
python train.py -w (No. of Workers) -e (Name of Env.) -l (Log Dir. Path)
tmux a -t a3c
```

We can add the option to see what our agent sees via adding the toggle `--visualise` to the first command, though this may consume too much resources. The recommended number of workers (agents) must not exceed the number of available CPU cores. However, the FLash Games are run at 5 fps by default, so it may be possible to run at 2× or even 3× of the number of available CPU cores.

We ran all the agents in a high-end laptop, on a VirtualMachine running Ubuntu 16.04. However, it was not possible to use all of the available RAM due to large consumption of RAM by the Flash Games, so we limited to only running 4 agents at once. For the Gym environments, however, we managed to run 8 agents at once without problem. The specifics of our computer is found in Table 5.1.

Table 5.1 Specification of the system used for the preliminary results we have obtained.

	System
Processor	Intel Core i7-4790
	3.60 GHz
RAM (Virtual Machine)	8 GB DDR3
Available Cores	4 cores
Software	Python 3.5.3, TensorFlow r1.3.0
	Universe 0.21.5, Gym 0.9.2

5.1 Tensorboard

For monitoring various metrics of all our asynchronous agents, we open `http://localhost:12345/` in a browser, which leads us to Tensorboard, a suite of visualization tools that is included in Tensorflow. We can use it to display plots (line plots, histograms, among others), images of what our agent sees, even the graph of the Neural Network and the tensors flowing between nodes.

For the line plots that we will display later on, Tensorboard uses a smoothing factor w that ranges from 0 to 1 which the user selects (0 being the original data). By default, we use a smoothing of $w = 0.6$, unless otherwise noted. The smoothed lines will be shown, while the original data will be 'ghosted' in the background. The smoothing algorithm is a moving average: given a point p and a window w, the algorithm replaces p with the average in the range $[p - \lfloor w/2 \rfloor, p + \lfloor w/2 \rfloor]$. When there aren't enough points to the left, the window is reduced to cover the exact number of points available. When there aren't enough points on the right, the line will stop being rendered, as we will see in the following sections.

Another feature is that we can manipulate the plots: zoom in or out, have a logarithmic scale on the vertical axis, expand the graphs, change the ticks in the horizontal axis from timesteps, relative time, to clock hour. At least 9 plots where generated for each environment, but we will not be using all of them, though they are useful (perhaps even necessary) when debugging and observing how our agents train.

5.2 Universe-starter-agent vs. A3C

While the unvierse-starter-agent is a great way to easily start with both Gym and Universe environments, as well as implementing the A3C algorithm, it is not precisely the same as the A3C. The major differences are:

- The images are sized down to 42×42 pixels, whereas the A3C sized down the images to 84×84 pixels.
- There are four convolutional layers with 32 filters of size 3×3 with stride 2. This smaller size may result in faster convolution, and it may have been chosen due to the larger amount of parameters (and hence features).
- It uses Exponential Linear Units (ELU) instead of ReLUs between hidden layers (Clevert et al. 2015).
- The number of timesteps that the policy is run before updating the parameters is increased form 5 to 20, that is, $t_{max} = 20$. The authors claim in the source code that making local steps be much smaller than 20 makes the algorithm more difficult to tune and to get to work.

The ELUs are defined and used like so in Tensorflow: `tf.nn.elu(features)`, which has as output `exp(features)-1` if `features<0`, `features` otherwise. Many of the changes we just listed in the universe-starter-agent may be due to simply experience by the programmers, so we can only speculate the real reason why they where chosen.

Lastly, the gradient descent optimization that the universe-starter-agent uses is not the RMSProp optimization algorithm we have previously discussed in Section 3.4.3. Instead, it uses what is known as the Adam optimizer, which is another algorithm like RMSProp that computes adaptive learning rates. We present it more formally in the following subsection.

5.2.1 Adam - Gradient Descent Optimization

The ***Adam optimizer*** (Kingma & Ba 2014) stores both an exponentially decaying average of past squared gradients v_t and gradients m_t defined as follows:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \qquad v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

where $\beta_1, \beta_2 \in [0, 1)$ and g_t is the gradient at time step t. In other words, m_t is the estimate of the mean and v_t is the estimate of the variance of the gradient. They will have a bias towards zero (due to the initialization values), so to correct that, we set the following estimates:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1} \qquad \hat{v}_t = \frac{v_t}{1 - \beta_2}$$

Finally, we use these for the Adam update rule:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t \tag{5.1}$$

where η is the learning rate and θ are the parameters we are updating. Note that effectively we have a decaying learning rate defined as $\eta_t = \eta\sqrt{1-\beta_2^t}/(1-\beta_1^t)$. This is recommended as learning progresses so as to not let the parameter vector bounce around needlessly as training progresses.

We can use Adam via the command in Tensorflow: `tf.train.AdamOptimizer`. Its arguments (and default values that (Kingma & Ba 2014) recommend) are: `learning_rate=0.001`, `beta1 = 0.9`, `beta2=0.999`, `epsilon=1e-8`, `use_locking=False`, `name='Adam'`. Its thorough documentation can be found at `https://www.tensorflow.org/api_docs/python/tf/train/AdamOptimizer`. We will be using different learning rates on some experiments in the following subsections.

5.3 Gym

In all of Atari 2600 games, the observations of the environment are arrays of shape $(210, 160, 3)$, specifically, `Box(210,160,3)`. On the regular version of the games (e.g., `Pong-v4`), all actions are repeatedly performed for a duration of k frames, where $k \sim \mathcal{U}(\{2,3,4\})$. On the Deterministic versions of the games (e.g., `PongDeterministic-v4`), all actions are repeatedly performed for a duration of 4 frames. In the documentation, this is referred to as `frameskip`. All games are simulated through the Arcade Learning Environment (ALE), which uses the Stella Atari emulator (Brockman et al. 2016).

Atari 2600 games have a shimmering effect that draws some objects only every second frame, so there may be a 50% chance of seeing some objects with the `Deterministic` versions of the games. Additionally, another version can be found named `NoFrameskip` which reports every frame.

Following the documentation of Gym in their repository, in order to obtain the shape of our observation space, we run `env.observation_space`, whereas to obtain the action space shape we run `env.action_space`. Most likely it will be of the type `Discrete` which, unlike the `Box` type, is only a discrete amount of actions. For example, `Discrete(6)` means there are a total of 6 actions, ranging from 0 to 5.

Depending on the game being trained, there will be a different set and number of actions used. These are all subsets from the full list of actions defined in the documentation as the following dictionary:

```
ACTION_MEANING = {0 : "NOOP",1 : "FIRE", 2 : "UP", 3 : "RIGHT", 4 : "LEFT", 5 : "DOWN", 6
↪  : "UPRIGHT", 7 : "UPLEFT", 8 : "DOWNRIGHT", 9 : "DOWNLEFT", 10 : "UPFIRE", 11 :
↪  "RIGHTFIRE", 12 : "LEFTFIRE", 13 : "DOWNFIRE", 14 : "UPRIGHTFIRE", 15 : "UPLEFTFIRE",
↪  16 : "DOWNRIGHTFIRE", 17 : "DOWNLEFTFIRE",}
```

If we so wish, we can obtain what each action performs in each environment (or what the agent can perform in that specific game) by running the command `env.env.get_action_meanings()`, which will give us an array of available actions for the environment. So, for our `Discrete(6)` example, this e.g. could give `['NOOP', 'FIRE', 'UP', 'RIGHT', 'LEFT', 'DOWN']`. Two things should be noted: these smaller subsets of actions do not have to go in order as in the dictionary `ACTION_MEANING`, and that the agent does not necessarily know what the actions mean, just that when it performs an action, it will get a certain reward. As such, we won't show what each Atari environment has as possible actions in order to keep the image of the environment as a black box more consistently.

Using all of this, we proceed with the following classical Atari 2600 games to test the power of the A3C algorithm, as well as to analyze what is happening and whether or not we can advance to the more complex environments.

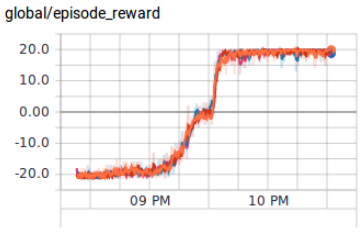
5.3.1 Pong

Pong was one of the first arcade video games. It is a 2-dimensional sports game that simulates the table tennis game. The agent receives a reward of 1 each time it scores against the other player, and a reward of -1 (penalty) is given each time the other player scores a point. On the plots reported by Tensroflow, only the final sum of rewards is printed, i.e., the minimum it can obtain is -21 and the maximum is 21.

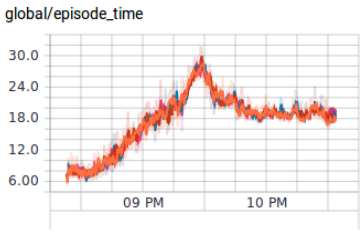
Due to its simplicity, we will test more on this game, as it quickly converges to a solution.

Deterministic vs. Normal Game

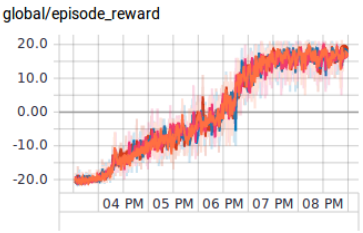
We will first compare the cases of Deterministic and 'normal' games, that is, we will compare `PongDeterministic-v4` against `Pong-v4`. We run the agents until they find a solution, or in other words, until they are the ones consistently winning with 21 points to 0 for the other computer player. We will train 4 agents for each type of game and present the results in Figure 5.1.



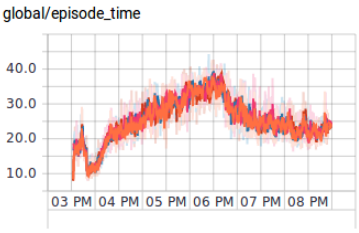
(a) Reward per episode for 4 agents in `PongDeterministic-v4`.



(b) Time length of the system for 4 agents in `PongDeterministic-v4`.



(c) Reward per episode for 4 agents in `Pong-v4`.



(d) Time length of the system for 4 agents in `Pong-v4`.

Figure 5.1 Comparison of results between using 4 and 8 agents. Each plotline represents the result obtained by each agent.

Note then the more random behavior of the agent in the `Pong-v4` environment due to it not knowing for how long will its actions be repeated. A clear indicator of this is the time it took to reach a reward of 0 (meaning that our agent starts to win) and a reward of 21 (meaning that our agent wins 21-0). For the `PongDeterministic-v4` environment, the former happens at around 1.8 Megasteps, or 1 hour 5 minutes of training. For the `Pong-v4` environment, this happens at around 5 Megasteps, or 3 hours of training.

The latter case, the 'always-winning' case, happens for the `PongDeterministic-v4` environment 2 Megasteps, or 1 hour 15 minutes of training. For the `Pong-v4` environment, on the other hand, this happens at around 7 Megasteps, or 4 hours of training. Thus, the `Deterministic` environments of the future games we test will be more likely to converge faster and aid us in not unnecessarily wasting resources.

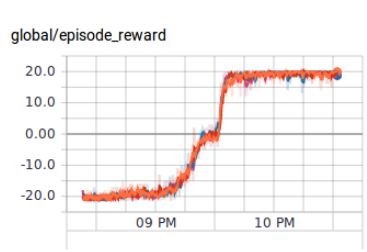
Increasing the number of agents

We now test how much it affects to have more agents than CPU cores. We thus run the `PongDeterministic-v4` environment with 4 and 8 agents, respectively.

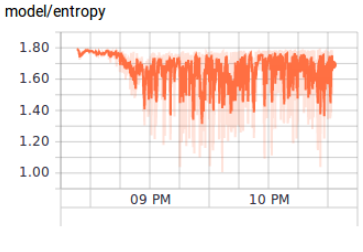
The first thing to note is the time it takes for our agent to start winning: for 4 agents, it happens at around 1 hour 5 minutes of training, or 1.8 Megasteps. For 8 agents, this happens at around 1 hour and a half of training, or 1.9 Megasteps. It must be noted that at this stage in the game, each episode lasts the longest, for the agent and the opposing player are basically in a tie and are both struggling to win.

For the 4 agents training asynchronously, it took 1 hour 15 minutes, or 2 Megasteps, for our agent to consistently win 21-0 against the other player, whereas it took around 2 hours or 2.9 Megasteps for our 8 agents to converge to the final strategy.

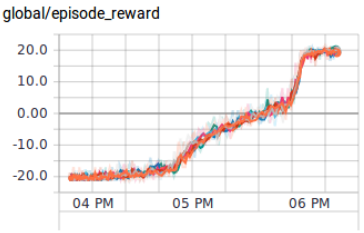
We note that the learning curve for our 4 agents is much steeper than that for 8 agents. Indeed, running more agents than there are of available CPUs/threads result in interference and hence we will refrain of doing so whenever possible, unless otherwise recommended by the universe-starter-agent documentation.



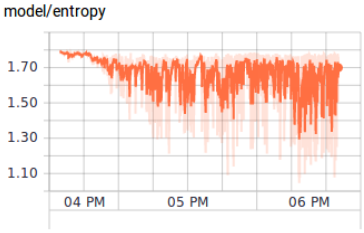
(a) Reward per episode for 4 agents.



(b) Entropy of the system for 4 agents.



(c) Reward per episode for 8 agents.

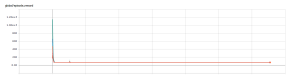


(d) Entropy of the system for 8 agents.

Figure 5.2 Comparison of results between using 4 and 8 agents in the `PongDeterministic-v4` environment. Each plotline represents the result obtained by each agent.

5.3.2 Ms Pacman

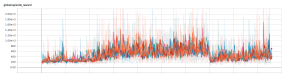
Perhaps one of the most widely-known arcade games, the player (agent) gains 10 points by eating orbs around the maze whilst avoiding the ghosts. If the player eats some special pellets, the power pellets, the player gains 50 points and is able to eat the ghosts and return them to their initial cage. The first ghost eaten awards 200 points, and then each additional one doubles this until reaching 1600 points. Since this is a step-up of complexity from Pong game, we will explore the `Deterministic` version only, and will thus vary the value of the entropy regularization term β and the initial learning rate η. We cross reference these with the graphs of the entropy given to us in Tensorboard. The respective results are given in Figures 5.3 and 5.4.



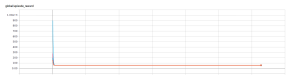
(a) $\beta = 10^{-1}, \eta = 10^{-2}$

(b) $\beta = 10^{-1}, \eta = 10^{-3}$

(c) $\beta = 10^{-1}, \eta = 10^{-4}$



(d) $\beta = 10^{-1}, \eta = 10^{-5}$

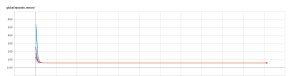


(e) $\beta = 10^{-2}, \eta = 10^{-2}$

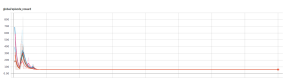
(f) $\beta = 10^{-2}, \eta = 10^{-3}$

(g) $\beta = 10^{-2}, \eta = 10^{-4}$

(h) $\beta = 10^{-2}, \eta = 10^{-5}$

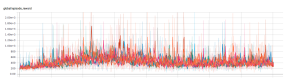


(i) $\beta = 10^{-3}, \eta = 10^{-2}$



(j) $\beta = 10^{-3}, \eta = 10^{-3}$

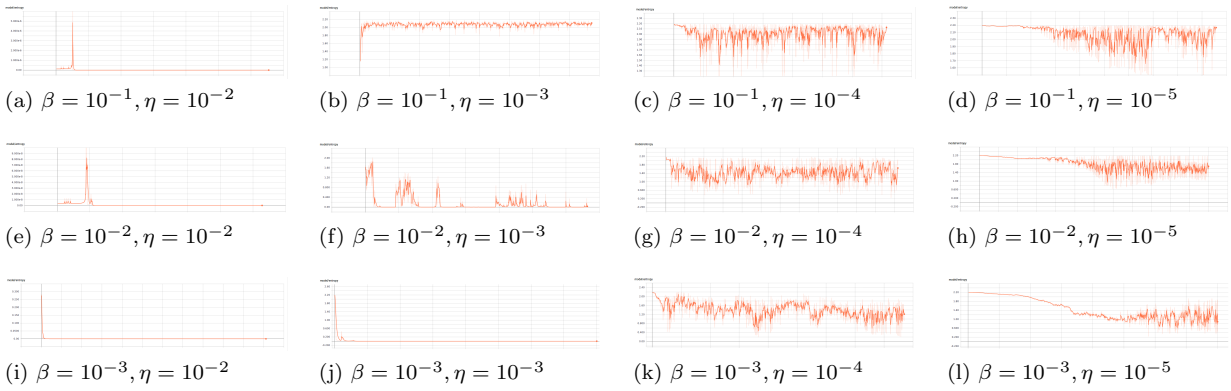
(k) $\beta = 10^{-3}, \eta = 10^{-4}$



(l) $\beta = 10^{-3}, \eta = 10^{-5}$

Figure 5.3 Rewards per episode per time step for different combinations of strength of the entropy regularization term (β) and learning rate (η) for the `MsPacmanDeterministic-v4` environment using the A3C algorithm. Each plotline represents a different agent.

We note that for large values of initial learning rate there is barely any exploration being done, and the agent quickly converges to a local maximum of simply turning two times and staying in a corner, attaining a maximum of 70 points. Decreasing the initial learning rate, we get the same problem if our entropy regularization term is not high enough. This makes sense as our agent is being incited to explore more with a larger entropy regularization term. We see that, of the combinations explored, the one that presents more



(a) $\beta = 10^{-1}, \eta = 10^{-2}$ (b) $\beta = 10^{-1}, \eta = 10^{-3}$ (c) $\beta = 10^{-1}, \eta = 10^{-4}$ (d) $\beta = 10^{-1}, \eta = 10^{-5}$

(e) $\beta = 10^{-2}, \eta = 10^{-2}$ (f) $\beta = 10^{-2}, \eta = 10^{-3}$ (g) $\beta = 10^{-2}, \eta = 10^{-4}$ (h) $\beta = 10^{-2}, \eta = 10^{-5}$

(i) $\beta = 10^{-3}, \eta = 10^{-2}$ (j) $\beta = 10^{-3}, \eta = 10^{-3}$ (k) $\beta = 10^{-3}, \eta = 10^{-4}$ (l) $\beta = 10^{-3}, \eta = 10^{-5}$

Figure 5.4 Entropy of the model per time step for different combinations of strength of the entropy regularization term (β) and learning rate (η) for the `MsPacmanDeterministic-v4` environment using the A3C algorithm.

stability throughout is that of Figure 5.3 and 5.4 (g), i.e., for $\beta = 0.01$ and $\eta = 0.0001$, the standard values of the universe-starter-agent. More training is needed in order for our agent to solve the environment, but at least we can rule out some combinations of learning rates and entropy regularization terms thanks to these experiments.

Each model was trained for at least 1 Megasteps (in the case for high value of initial learning rate, as we see that they quickly converge to a deterministic model and hence it is unnecessary to keep training them). For the rest, the models where trained for an average of 4 Megasteps (or 1 epoch in (Mnih et al. 2016)) and a maximum of 9.5 Megasteps for the case of Figure 5.3 (d), or when $\beta = 0.1$ and $\eta = 0.00001$. Just this one represents 5 hours and a half of training on our local computer but as noted before, far more training is needed in order to converge to a solution.

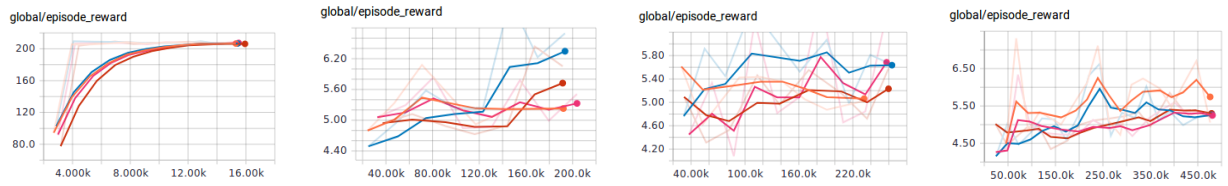
5.4 Flash Games

5.4.1 Wheelers

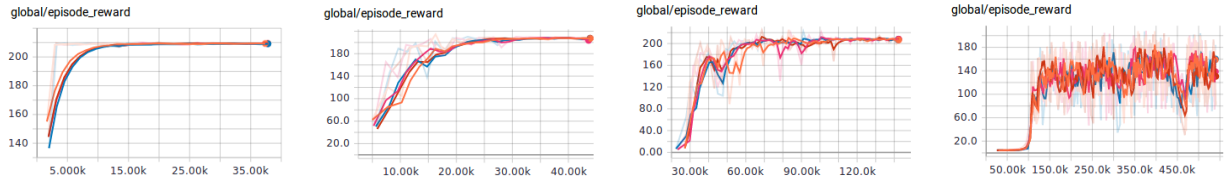
Starting now with track racing games, `flashgames.Wheelers-v0` lets the player control a motorbike to race along a 2D track. Though there are objects in the environment such as trees and stands, the player does not crash into anything. The only negative effect is the grass, where the player will travel at a slower pace. There is a pit stop in order to refuel the gasoline (which, if the player runs out of, the game ends) and the point of the game is to win the race, that is, get 1st place. We present the results in Figures 5.5 and 5.6

The solution found by our agent for which the entropy reaches zero (or very close to) means that it will simply go straight ahead, without regard for the race or the stage, it will continue on ahead until it runs out of gas and then the game resets. Increasing our entropy and decreasing our learning rate ($\beta = 0.1$ and $\eta = 0.00001$ respectively), we see that there is some intention of exploring other paths to take, but the rewards it obtains pales in comparison to simply going straight ahead until it runs out of gas. In particular, this scenario was run for 4.8×10^5 timesteps, that is, around 6 hours and 30 minutes of training, but it still could not find a better alternative. In contrast, decreasing the entropy regularization term to $\beta = 0.01$ takes us back to the solution of accelerating straight ahead, but every now and then turning, which is almost irrelevant when it still runs out of gas at the end, far away from the track itself. This scenario was trained for 5.4×10^5 timesteps, or 7 hours 50 minutes of training.

The reward system in this environment is not entirely understood, as it is clear that the agent receives a positive reward by merely going straight, and a large one for that. Perhaps it is not an intended event considered by the makers of the game, but it is the result we obtain nonetheless. An interesting point is how fast it finds the solution to accelerate straight ahead, as we can see for the low values of $\beta = 0.1$ and $\eta = 0.01$.

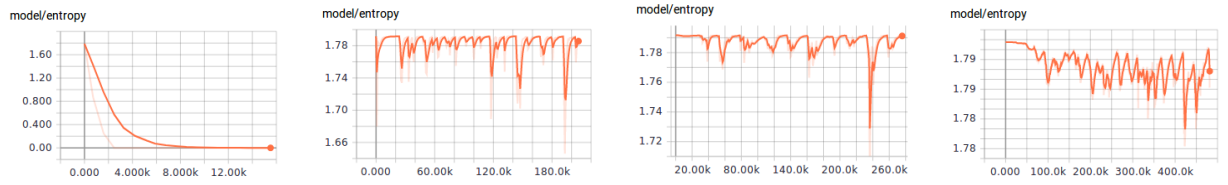


(a) $\beta = 10^{-1}, \eta = 10^{-2}$ (b) $\beta = 10^{-1}, \eta = 10^{-3}$ (c) $\beta = 10^{-1}, \eta = 10^{-4}$ (d) $\beta = 10^{-1}, \eta = 10^{-5}$

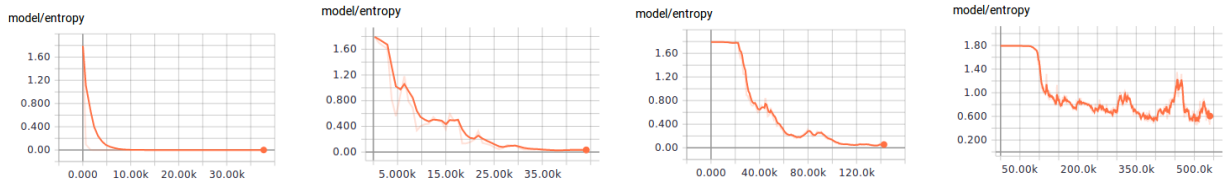


(e) $\beta = 10^{-2}, \eta = 10^{-2}$ (f) $\beta = 10^{-2}, \eta = 10^{-3}$ (g) $\beta = 10^{-2}, \eta = 10^{-4}$ (h) $\beta = 10^{-2}, \eta = 10^{-5}$

Figure 5.5 Rewards per episode per time step for different combinations of strength of the entropy regularization term (β) and learning rate (η) for the `flashgames.Wheelers-v0` environment using the A3C algorithm. We ran 4 agents, with a different plotline representing each agent.



(a) $\beta = 10^{-1}, \eta = 10^{-2}$ (b) $\beta = 10^{-1}, \eta = 10^{-3}$ (c) $\beta = 10^{-1}, \eta = 10^{-4}$ (d) $\beta = 10^{-1}, \eta = 10^{-5}$



(e) $\beta = 10^{-2}, \eta = 10^{-2}$ (f) $\beta = 10^{-2}, \eta = 10^{-3}$ (g) $\beta = 10^{-2}, \eta = 10^{-4}$ (h) $\beta = 10^{-2}, \eta = 10^{-5}$

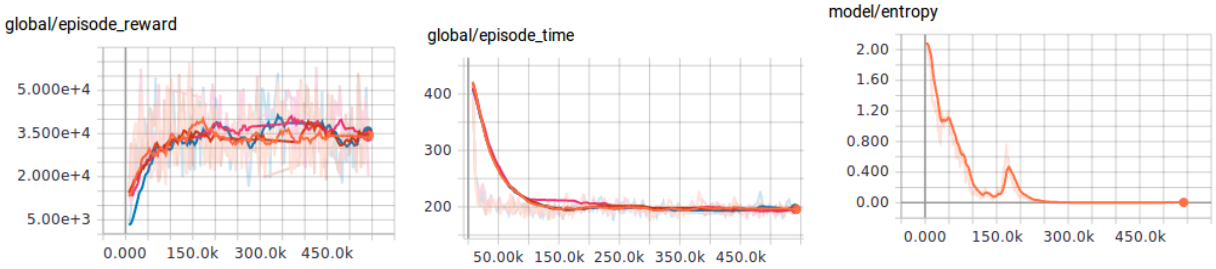
Figure 5.6 Entropy of the model for different combinations of strength of the entropy regularization term (β) and learning rate (η) for the `flashgames.Wheelers-v0` environment using the A3C algorithm.

5.4.2 Neon Race

A more colorful racing game, the `flashgames.NeonRace-v0` simulates a street racing car whose objective is to reach the end of the track in the least amount of time possible, while collecting coins and crashing into the red neon vehicles. We run 4 agents asynchronously, and present the results of this in Figure 5.7.

We see that the agent reaches a reward of 3.5×10^4, effectively taking around 200 seconds to complete each lap. Due to the gray scaling of the images, the other red and pink vehicles are indistinguishable, and hence why the agent does not try to crash into only the red ones and gain points from this. More importantly, the agent has just learned that, due to the mechanics of the game, simply pressing the Up key will yield the best result. As such, this becomes entirely deterministic and why the Entropy reaches zero.

The game was run for a long time due to the entropy growing in some instances, indicating that the agent was trying other actions. The total training time was around 5.4×10^5 time steps, or 8 hours 40 minutes of training, due to the limitation imposed by the universe-starting-agent on the Flash Games to run at 5 frames per second maximum.



(a) Reward per episode for 4 agents.

(b) Total time in seconds per episode for 4 agents.

(c) Entropy of the model.

Figure 5.7 Results for the `flashgames.NeonRace-v0` environment using the A3C algorithm.

5.5 Conclusions

Given that the A3C algorithm is on-policy (i.e., the policy will gradually improve), then our agent needs a feedback on what is a good action and what isn't. Many games we played gave their reward at the end of the game, which will only raise problems to our algorithm. This was precisely the problem in the MsPacman game. On the other hand, in the Wheelers flash game the agent received a reward, but the problem laid in it being always positive and hence why it never found 'bad' actions to never take, like accelerating straight outside of the track.

One possible solution might be to increase even more to our entropy term β so that our agent manages to actually finish the games and hence improve the policy π. Another is to standardize the rewards, or more importantly, to have more control of our environment. At the end, the main problem with the racing Flash Games was due to how the rewards were given to our agent, which lacked the engineering necessary to reward the good actions it took and punish the bad ones in order for it to not crash on the walls or to at least finish a lap.

Tuning hyperparameters may seem tedious, but it is necessary in order to obtain the desired result (if any). A list of possible hyperparameters to tune include the learning rate η, the amount of gradient norm clipping, the entropy regularization weight β, the discount factor γ, the gradient descent hyperparameters such as the RMSProp decay factor α or the Adam the ones found in Adam (ref. Section 5.2.1), and the exploration rate ϵ. Some have their own logic on whether or not they should be tuned, such as e.g. the discount factor, since changing this will imply whether or not actions from the past influence those in the future.

Finally, while some of our experiments seemed to be going nowhere, this might be simply because they needed more time to train. Indeed, while we were limited to using only 4 agents (or 8 depending on the setting), (Mnih et al. 2016) found that the A3C had a sub-linear speedup with the number of threads being used. While not being particularly accurate, a 12.5 speedup by having 16 threads compared to a 3.7 speedup we currently have by using 4 threads seems quite encouraging. Indeed, this is why we need more CPU cores, as we will discuss in Section 7.2. Incorporating experience replay may substantially improve the data efficiency by reusing old data, which could lead to faster training times in all domains, including the more complex ones (like TORCS; see Section 6.1).

We should add that we have experimented with many other games in the Atari 2600 environment such as Space Invaders, Assault, Breakout, Amidar, Berzerk, and Skiing (many using both the normal and the Deterministic versions), but they added little to the discussion, and hence will be omitted altogether from this report. The same applies for the Flash Games, where we also experimented on Drift Runners and Dusk Drive, but the result was the same: on all the available racing games let the agent reach a local maximum by simply taking the Up action. This is why we must need a more complex and well-controlled environment, like the `CarRacing-v0` or Udacity's simulator, seen in Sections 7.1.

As noted before, all of the code of our project (up to date) can be found at the Repositories tab in the following link: `https://github.com/PDillis`. We hope to be updating it as quickly as possible, and as often as a breakthrough is achieved.

Chapter 6

Virtual to Real Self-Driving Cars using RL

Reinforcement Learning presents to us a unique way to turn the much sought-after task of autonomous vehicles. Indeed, since the notion of trial and error is how we learn how to perform new tasks, it is only natural for us to push machines into learning through experience rather than through carefully labeled data, as this will lead to better systems that can cope with never-before-seen scenarios and react similarly to how we would react—and even better. It is only logical that we wish to transfer this ability to those agents that will be taking our place in decision making and of course improve it, since 1.25 million people die every year from car accidents around the world, and 94% of crashes between vehicles are caused due to human error (see `https://x.company/waymo/`).

Evidently, we cannot apply the notion of learning by failure to real-life vehicles whilst they are being trained. Not only would it be costly to do such an endeavor, but also extremely dangerous to both the public and the research team. As such, the logical way is to first train our learning agent in a virtual environment and then test it with real-world data we may have, or even in a real vehicle in a controlled environment.

There have been many attempts to do this type of training, of which we will see the latest in the following sections. Perhaps it should be noted that many aspects will be necessary to advance this particular line of work: from improving the algorithms used to train the agents, to better resources (such as sensors or CPU/GPU), to better integration of high-level data and low-level data (e.g. cameras and LIDARs respectively). Indeed, we do not wish for self-driving vehicles to have the same limitations that humans possess, but to excel in the job and save as many lives as possible.

We proceed with some of the latest efforts in using Reinforcement Learning to train self-driving cars, starting with the type of environments these have used, to how they dealt with the reward function and the continuous domain, to finally on how to solve some and perhaps prevent of the issues of safety in Artificial Intelligence, especially since we wish to apply them in self-driving cars.

6.1 Environments and Frameworks

TORCS, The Open Racing Car Simulator, is perhaps the most popular open-source simulator for self-driving car research as of now. It can be found at `http://torcs.sourceforge.net/`. As previously discussed in Section 3.4.5, the A3C was also tested using the TORCS game in (Mnih et al. 2016). This was not only because it has more complex graphics than the Atari 2600 games, but also because the agent is required to learn the dynamics of the car it is controlling. The authors performed experiments with the same neural network architecture and hyperparameters as used in the Atari 2600 game environments (ref. Section 3.4.5) and they did so in four different settings: the agent was controlling a slow or a fast car in the racetrack, each case with and without opponent bots. Overall, the A3C performed best, obtaining between 75% and 90% of the score obtained by a human tester on these same configurations. The reward function is examined in the Section 6.2.

Likewise, (Lau 2016) uses TORCS to test the Deep-Deterministic Policy Gradient (DDPG) algorithm as we will see later on, and obtained fairly good results. The author used the Aalborg track as the main training environment and the Alpine 1 for validation (to avoid overfitting). See Section 6.2 for how the author chose to reward the agent, as well as the results obtained with different types of reward functions.

On the other hand, (Perot et al. 2017) argue that, while the TORCS engine has been proven to be useful, its physics and graphics lack the realism for the results to be efficiently translated into the real world. Thus, the authors resort to a realistic car racing game, World Rally Championship 6 (WRC6): the environment has more realistic physics like grip and drift for example, better graphics (illuminations and animations), and a variety of environments with sharp turns, slopes, cliffs, and snow. The author's results are seen ahead along with the reward function being used.

We proceed with more complex ways to use these environments in the following two subsections.

6.1.1 Deep RL framework for Autonomous Driving

(Sallab et al. 2017) tested their Deep Reinforcement Learning Framework for Autonomous Driving in the TORCS environment. The authors propose this framework as they note the difficulties in posing autonomous driving as a supervised learning problem, mainly taking into account the interactions of multiple agents, i.e., interactions with other vehicles, pedestrians, and objects on the road. Their framework incorporates Recurrent Neural Networks (RNN) for integrating information and attention models to focus on relevant information and reduce the computational complexity.

The authors divide into three the tasks in creating an autonomous driving agent:

- **Recognition** Identifying components of the surrounding environment, e.g., pedestrians or traffic signs. Convolutional Neural Networks (CNN) are best used in this area, so a combination of these and DNN is desired.

- **Prediction** Building internal models that predict future states of the environment, e.g., tracking an object's movement. For this, we must be able to integrate past information, so RNNs are essential (in particular LSTM networks).

- **Planning** Generating an efficient model that incorporates both recognition and prediction to plan for a future sequence of actions by the agent to reach the desired destination. This is the hardest of the three according to the authors since the first two tasks must be combined with giving a reward or a penalty to the agent as it traverses the environment.

Autonomous driving will require integrating the information from multiple sensors, both from low dimensional data (such as the LIDAR) and high dimensional data (such as cameras). Note though that usually the memory bandwidth of the problem can be lowered, as we are not interested in *all* of the high dimensional data. For example, it is unnecessary to allocate resources to get the fine details of other vehicles; it suffices that our system is able to recognize them.

Attention models mix both Reinforcement Learning and RNNs in order to obtain the parts of an image that the agent truly cares about. These can then be extended and integrated to DQN and Deep Recurrent Q Networks (DRQN) models. This is the purpose of the framework proposed by the authors: to have as an input raw sensor inputs and as output the driving actions.

Perhaps most illuminating is the point of view of the authors: that current approaches decouple autonomous driving into isolated sub-problems (such as using supervised learning for object detection), and then having a post-processing layer to combine the results of these previous steps. The two main issues that the authors claim this approach has are the following:

1. The sub-problems being solved may be *more difficult* than autonomous driving on its own, e.g., object detection via semantic segmentation is both difficult and (perhaps) unnecessary. Their argument for this lies in the fact that human drivers do not detect and classify all the objects that they see whilst driving, only the most relevant ones in a specific area of interest.

2. The isolated sub-problems may not combine in a coherent way to achieve the goal of driving. We can address this issue in RL via the reward signal (classifying good and bad driving), but all this training must be done in virtual environments.

The author's pipelined framework for end-to-end training of a DNN for autonomous driving is divided into three parts (reminiscent of the three tasks seen before for creating autonomous driving):

- **Spatial aggregation** This stage contains two networks: sensor fusion and spatial features. The car state includes its position, orientation, velocity, and acceleration. We also need information on the

environment, but usually the environment state (and of the surrounding objects) is not directly observed but deduced by the algorithm using the sensor readings. Thus, we will need to fuse this sensory information and this is where the CNN comes to shine with the features it can extract from the observations.

  - **Sensor fusion** All the sensors' information encoding the environment should be grouped together in a raw input vector and presented like so to the DNN. Thus, the DNN will worry about adjusting the weights using, for example, SGD. For this type of job, a CNN is best suited.
  - **Spatial features** The fused sensors representation will enable the data to be further processed by focusing on the important part that is relevant to the driving task. The CNN will extract deeper representations, followed by an attention filter to direct the kernels of the convolution into the parts of the data that are of actual interest. An attention filter will reduce the dimensionality of the data, whereas an action and glimpse networks may help in learning to deploy the gaze of the kernel to parts of the data that are of interest, in turn reducing the computations dramatically.

- **Recurrent temporal aggregation** Since the environment state is not directly accessible to the agent, and single snapshots of the sensors are not sufficient to convey the whole information, information integration will be mandatory as the agent traverses the environment, thus revealing the state. Adding recurrence to the pipeline will enable handling POMDP situations, which is precisely what driving is. RNN are able to model long term dependencies on previous and current states, but in practice, LSTM are used due to them being able to control which information is kept from the previous state, as well as which to forget.

- **Planning** This is the Reinforcement Learning planning part: for discrete actions, a DQN may be used, whereas for continuous actions, a Deep Deterministic Actor-Critic (DDAC) may be used. The error of the MSE is backpropagated through the network to update the parameters.

The authors conclude by using the TORCS environment with its virtual sensors, such as *trackPos*, which gives the position of the track borders and the car speed in the $x-$position, among many others (steering, velocity, etc.). The output will be the steering, gear, acceleration and brake values. The actions $(steer, gear, brake, acceleration)$ are tiled and discretized (when the authors used the DQN) and kept continuous (when using the DDAC). The former will converge faster thanks to the replay memory but will have more abrupt steering actions, whereas the latter will have smoother actions and will provide a better performance.

This framework was tested for the lane keep assist algorithm and also tested via apprenticeship learning. In this test, there is a proportional controller that is trying to keep the lane whilst limiting the speed. Thus the level of supervision is reduced gradually until it reaches zero, i.e., the model is completely in control. Hence, this type of framework may lead successfully to an end-to-end Deep Reinforcement Learning. All that is left is to test it in a simulated environment where the sensors and actuators are controlled artificially with a labeled ground truth.

6.1.2 Virtual-to-real-world image translation

While certainly a better option to train models is to simply use more realistic environments, (You et al. 2017) propose a network translate models trained in virtual environments to the real world. This network converts non-realistic virtual images into realistic ones with a similar scene structure. Thus, given a realistic frame as an input, it should be able to adapt to real world driving using a policy trained via Reinforcement Learning on the virtual environment.

Given that all the most recent research on Reinforcement Learning applied to self-driving cars is in the stage of simulation, the authors propose a realistic translation network that converts virtual images rendered in the simulators to a realistic one. They use scene parsing representation to achieve this, which includes two modules: a virtual-to-segmentation module that pursues a scene parsing representation of the virtual image as input, and a parsing-to-real network to translate these scene parsing representations into realistic images. Thus, Reinforcement Learning can be applied to the real-world videos and hence applied to real-world driving. See Figure 6.1.

The authors used two other models for comparison: the first one being a Reinforcement Learning model with only a virtual image as input having never before seen real world images, and the second a supervised learning model that maps real images to their corresponding actions. They found out that their model, the

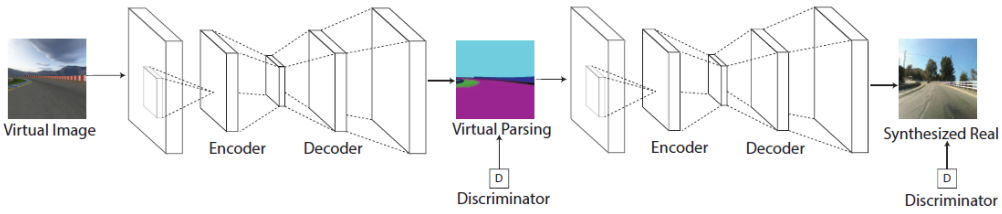


Figure 6.1 Network structure of the virtual-to-real image translation: the virtual images where given as input, which in turn where segmented by the segmentation network and translated to a real image. Reproduced from (You et al. 2017).

Reinforcement Learning trained with translated real images, had better performance than the other two. For training, the authors used 1673 images from the TORCS environment and segmented 45569 images from the CityScape dataset found at `https://www.cityscapes-dataset.com/`.

The results are quite promising: the accuracy of the action prediction of the author's model is 43.40%, while the accuracy of the normal Reinforcement Learning model was of 28.33% and for the supervised learning model was of 53.60%. While of course less accurate than the supervised learning one, the power resides in combining available datasets along with the simulations we are using to make better prediction models than purely Reinforcement Learning, as it is being done in many applications we have found.

6.2 Reward Functions

(Perot et al. 2017) do not use the in-game score as it is too sparse to train the agents. Instead, they propose and use the following short, but succinct, reward function for their self-driving agent:

$$R_t = V_t(\cos\theta_t - d_t) \tag{6.1}$$

where, at time t, d_t is the distance to the middle of the road (taken as a penalty), θ_t is the angle difference between the car's heading and the road, and V_t is the speed of the car. Thus, the agent will seek to maximize the speed and keep the distance to the center of the road as small as possible, as well as minimizing the angle difference between the car's heading and the road. They did this to compare with (Mnih et al. 2016), where the reward function was simpler, i.e., the velocity of the car projected along the track direction:

$$R_t = V_t \cos\theta_t \tag{6.2}$$

The results of (Perot et al. 2017) using the A3C algorithm are promising; they are presented in Table 6.1. The authors reasoned that it was because the reward function 6.2 that the car tends to slide along the guard rail, which of course is more dangerous and the car is more likely to crash, as well as pushing the car to go slower. A non-numeric result is that the driving is *smoother*. After 1.9×10^8 steps of training using the reward function 6.1, the agent learned to drive on different types of roads and rely on road edges for control of the vehicle.

Table 6.1 Results of (Perot et al. 2017) for different reward functions, using the algorithm A3C in the environment WRC6.

		After 8×10^7 steps of training	
Reward function	Steps to clear 80% of track	Average Speed	No. of crashes
$V_t \cos\theta_t$	1.5×10^7	82.9 km h^{-1}	6.2/km
$V_t(\cos\theta_t - d_t)$	6.0×10^7	88.0 km h^{-1}	0.9/km

On the other hand, (Lau 2016) used the TORCS environment. The reasons for using this environment where mainly to better visualize both whenever the agent is stuck in a local optimum as well as how the

neural network trains over time and not just look at the final result. The author didn't use DQN because discretizing the steering angles would quickly lead to the ***curse of dimensionality***. Instead, the author used the Deep Deterministic Policy Gradient (DDPG) from (Lillicrap et al. 2015), an Actor-Critic algorithm, with the continuous Q-learning (SARSA) as the critic model and a policy gradient method as the actor model. We won't delve too much into this algorithm, as we didn't apply or study it.

The author argues that the reward function used in (Lillicrap et al. 2015), which is equal to Equation 6.2, makes training unstable (i.e., sometimes it didn't train at all), due to the form of the reward function itself. We can see why: it would encourage the agent to accelerate as much as possible which made it more likely to crash and hence reach some sorts of local optimum. The author then proposes the following reward function:

$$R_t = V_t \cos\theta_t - V_t \sin\theta_t - V_t|trackPos| \tag{6.3}$$

where, at time t, $trackPos$ is the distance between the car and the track axis, and the rest of the terms follow the same notation as we have used before. Note that $trackPos$ is a sensor input given by TORCS, so normalizing it with respect to the track width is recommended.

What Equation 6.3 implies is that the agent should reach the maximum longitudinal velocity possible, while minimizing its transverse velocity, and we penalize every time the agent drives away from the center of the track. However, it must be noted that the author also used a reward function that did not include the $trackPos$ term and the results where similar. This is something to consider to try when we use our own versions of these reward functions.

6.3 Exploration in the Continuous Domain

To deal with the problem of exploration in the continuous domain, (Lau 2016) argues that $\epsilon-$greedy policies do not work, as having 3 different actions would imply that some combination of two opposing actions (such as acceleration and brake) would occur. Therefore, he proposes to add some noise to the exploration by using an Ornstein-Uhlenbeck process (selected due to its mean-reverting properties):

$$dx_t = \theta(\mu - x_t)dt + \sigma dW_t \tag{6.4}$$

The values of each parameters for each type of action is given in the Table 6.2. The positive mean acceleration will ensure us that the agent won't get stuck in a local optimum, where it will be constantly pressing the brake. The author also used a ***stochastic brake*** that would activate 10% of the time during the exploration phase. This taught the agent to brake on turns, mimicking better how a human drives.

Table 6.2 Suggested values for the parameters used in the Ornstein-Uhlenbeck process. Reproduced from (Lau 2016).

Action	θ	μ	σ
Steering	0.6	0.0	0.30
Acceleration	1.0	$[0.3 - 0.6]$	0.10
Brake	1.0	-0.1	0.05

The author concludes that, while DDPG from (Lillicrap et al. 2015) is a powerful tool for continuous domains, it is still different to how a human learns to drive. Indeed, if we add complexity to our vehicle, the exploration becomes more complicated and hence not so clear how we should do it. For example, changing from an automatic vehicle to a manual, which in turns add the clutch, and making the environment's information harder to obtain, e.g. we can only get pictures via cameras as in Section 7.1.2 would greatly increase the number of combinations of parameters that must be taken into account.

6.3.1 MuJoCo

Continuing our discussion of continuous action spaces, (Mnih et al. 2016) tested the A3C algorithm with the MuJoCo physics simulator. The setup used was as follows: the physical state was the input as well as how the agent tried to learn the task at hand, which consisted of the joint positions and velocities, as well as the target position (if there was one). For some tasks, the authors examined training directly from the RGB images of the screen. In this case, the input was passed through two convolutional layers without any non-linearity or pooling. For the low dimensional physical state cases, the inputs are mapped to a hidden state using one hidden layer with 200 ReLU units. The output of both encoders was fed to a single layer of 128 LSTM cells.

The two outputs of the policy network are two real number vectors that the authors treat as the mean vector $\boldsymbol{\mu}$ and scalar variance σ^2 of a multidimensional normal distribution with spherical covariance. The input is passed through the model to the output layer, where the normal distribution determined by $\boldsymbol{\mu}$ and σ^2 generates our action. In practice, $\boldsymbol{\mu}$ is modeled by a linear layer and σ^2 by a SoftPlus operation $g(x) = \log(1 + \exp x)$, and the activation is computed as a function of the output of a linear layer.

Both networks for the policy and value do not share any parameters, unlike in the discrete problems we have seen in Section 3.4.5. The authors did not use any bootstrapping in the policy or value function updates and batched each episode into a single update. An entropy cost was included that encouraged exploration: a cost on the differential entropy of the normal distribution defined by the output of the actor network, $-\frac{1}{2}(\log(2\pi\sigma^2) + 1)$ and using a constant multiplier of 10^{-4} for this cost across all the tasks examined.

While not precisely a self-driving car simulator, the techniques used may shed some light on how to use the A3C in the continuous domain, e.g., for self-driving cars. The authors found that the algorithm achieved good solutions using only the CPU in less than 24 hours, and sometimes even in a few hours. The MuJoCo simulator presents interesting environments for testing the algorithms for robotics (like moving an object or even 3D manipulation). This is extremely useful for control of robotics or the different machinery inside a self-driving car.

6.3.2 Labyrinth

In the same paper ((Mnih et al. 2016)), the authors tested the A3C algorithm in a new 3D environment named Labyrinth. The task for the agent was to find rewards in randomly generated mazes: the agent is placed at the beginning of each episode in a new randomly generated maze consisting of rooms and corridors, and each maze had two types of objects, apples and portals: picking an apple gives a reward of 1 while entering a portal gives a reward of 10. The goal of the agent is to score as many points as possible and, after 60 seconds, a new episode would begin and hence a new maze is generated. This presents the agent with a more challenging domain since it must learn a *general* strategy due to the random nature of the generated mazes. The authors trained an A3C LSTM agent using only 84×84 RGB images as input, obtaining a final average score of around 50.

This presents an interesting case, as we can view the streets that the self-driving car will traverse as a maze in a way. Thus, the random generation of apples may represent events such as collisions for example, where a negative reward may be given and see how the agent reacts (and hopefully learns to evade) to such events. A more complex environment will be needed, though it is reassuring to see that the A3C is capable of deducing a general strategy.

6.4 AI and Safety

While in theory, everything should work according to plan, (Shalev-Shwartz et al. 2016) argue differently: they argue that the objective itself of maximizing a long-term reward poses a functional safety problem. If the objective is to avoid accidents, for example, what should be the penalty given to the agent should it occur (compared to other instances where we give it a positive reward)? If high enough, it should discourage the event of it happening, but at the same time, it will make the variance of the rewards exceedingly high.

Thus, the solution lies not in performing mathematical tricks like using the baseline (as seen in Section 3.1), but on redesigning the architecture. The authors achieve this by applying two ideas: first restructure the policy function as $\pi_\theta = \pi^{(T)} \circ \pi_\theta^{(D)}$ and Temporal abstraction using option graphs as a way to breaking down the problem into semantically meaningful components. On the former, $\pi_\theta^{(D)}$ maps the state space into a set of so-called Desires for comfortable driving, and $\pi^{(T)}$ maps these Desires into a trajectory, i.e. the safety

of the passengers depends on this. On the latter reduces the effective horizon (and thus the variance of the gradient estimations) while also reducing the sample complexity and plays a similar role as LSTM that we have previously seen.

While the former application seems more formal and perhaps more practical, there is a more philosophical nature to this question of Reinforcement Learning and safety. More generally, in Artificial Intelligence and safety. This is what a group of researchers from Google, OpenAI, Stanford and UC Berkeley have released in (Amodei et al. 2016), where they discuss the potential accidents that may occur with AI and machine learning systems. Accidents are defined in the paper as "unintended and harmful behavior that may emerge from ML systems when either the wrong objective function is specified, the research team is not careful about the learning process, or the agent commits other machine learning implementation errors".

The authors argue that it is not necessary to highlight extreme scenarios in order to discuss accidents; if anything, this would just lead to discussions that lack precision. Therefore, it is more productive to frame accidents in terms of practical issues within modern machine learning technologies (and hence can be prevented and solved).

The authors categorize safety problems according to where in the process things went wrong. While each category alone is perhaps a research problem, we mention them and try to explain them as concisely as possible in the context of self-driving cars:

1. The designer of the AI or ML system may have specified the wrong formal objective function, so that maximizing this will lead to harmful results.

   - **Negative side effects** The designer specifies an objective function that focuses on accomplishing some specific task in the environment but ignores other aspects of it. Thus, indifference over the other variables in the environment may result in harmful behavior. How can we ensure that our self-driving car won't crash into objects on the road in order to get to the destination as soon as possible? Can this be done without manually specifying everything the car should ***not*** crash into?
   - **Reward hacking** The objective function admits some 'clever' solutions that maximize the rewards but perverts the intent of the designer. In other words, the objective function can be 'gamed'. If we give rewards to our self-driving car according to the number of crashes, it might just learn to disable the sensors that tell it whether it has crashed or not.

2. The designer may know the correct objective function (or has a method to evaluate it), but doing this is too expensive for the agent, leading to harmful behavior due to extrapolating from a limited dataset.

   - **Scalable oversight** How to ensure safe behavior by our agent when given a limited access to the true objective function? If our self-driving car encounters a scenario it hasn't encountered before and there are no passengers inside of it, can it manage to overcome this scenario without the need for it to ask a direct third party?

3. The designer may have specified the correct objective function, but something bad occurs due to making decisions from insufficient or poorly curated training data.

   - **Safe exploration** How do we ensure that exploratory actions in RL agents do not lead to negative (or even irrecoverable) consequences that outweigh the long-term value of exploration? Our self-driving car may attempt to explore a never-before seen route, but this might lead it into a state where it cannot escape, i.e., crashing.
   - **Robustness to distributional shift** How do we avoid that our ML systems make bad decisions when given inputs that are very different to what it has seen during training? Is our self-driving car robust enough to be trained in a city and go to the countryside without major repercussions, for example?

These are important questions, mainly due to the increased presence of Reinforcement Learning agents, the increase in complexity of both agents and the environments, and the increase in autonomy and AI systems being brought forth by the industry. The five problems we have previously discussed have been focused on the self-driving car world, but they are also applicable to many other fields, such as medicine, personal-data handling, politics, and security, among others. If small-scale accidents do occur, will this have a negative effect on the trustworthiness and perception from the public of AI and autonomous systems?

It may not be very prudent to analyze and reach a solution for each problem via case-by-case rules. A more general, end-to-end approach to prevent systems from causing (unintended) harm is a more logical

approach and is the one the authors proclaim the studies should be focused on. Indeed, lest we do not forget especially for autonomous vehicles, many lives will depend on this and accidents are far more notorious, given the mistrust of the public in faceless machines. How to ensure that the trust remains intact (or at least as high as possible) and advancement in the field is not hindered by unnecessary politics? We can only do this by studying and trying to tackle these problems before they are brought into the general view via an accident, but this will require much-needed research as our algorithms, environments, agents, and systems become more and more complex.

As a final remark, we reference (Lake et al. 2016) for the current take on AI, especially on how to have better structured model. Indeed, if we wish to mimic human thought and processing, this paper is a must-read for many outside the field, but perhaps more crucially to professionals in the area.

Chapter 7

Final Remarks and Next Steps

And I am going to finish this chapter with two interesting facts about Sherlock Holmes.

— Mark Haddon, *The Curious Incident of the Dog in the Night-time*

We will present the next and final stage in our internship, which is to apply the A3C algorithm to a racing simulator environment with a well-defined reward function. As we saw in Section 6.2, defining this reward function is of key importance, as it will indirectly tell our agent what we wish for it to accomplish.

Furthermore, although we have barely begun with the A3C algorithm, the world of Artificial Intelligence and Reinforcement Learning is quickly evolving. While we may apply the vanilla A3C algorithm, there have been many improvements to it surfacing in the last months, as well as other promising algorithms. We will quickly explore some in this section, as they may present themselves as the next step in our quest for attaining Self-Driving vehicles with Artificial General Intelligence, but more are sure to show up.

7.1 Next Environments

There are two environments we wish to use to apply what we have learned during this internship. One is from the Gym environment, and the other is a simulator made in Unity. We present both of these in the following sections as our final steps in the internship at the Computer Vision Center.

7.1.1 Gym - CarRacing-v0

The environment with most similarities to what we wish to accomplish is the `CarRacing-v0` environment by Oleg Klimov, an environment in the Gym Box2D repertoire. Even if its current state is purely experimental, it has the following documented traits, which some we find to be useful for our future application of our algorithm:

- The reward is -0.1 points for every frame and $+1000/N$ points for every track tile visited, where N are the total tiles on the track. This will incentivize our agent to finish as fast as possible, while staying on the track.
- The game is considered to be *solved* by Gym when the agent gets 900 points or more over 100 consecutive trials.
- The track is random for every episode, which will aid us avoid overfitting.
- The episode ends when all tiles are visited. If the agent goes too far off the track, then it will get -100 points and be terminated.
- At the bottom of the screen, there are 3 indicators: the true speed, four ABS sensors, steering wheel position and gyroscope, if read from left to right.
- The action space is: `A(s)=Box(np.array([-1,0,0]), np.array([+1,+1,+1]))`, and the state space is `S=Box(low=0, high=255, shape=(96, 96, 3))`. If $a \in \mathcal{A}(s)$, then this means that

`a=[steer, gas, brake]`, i.e., the first element being how much the car turns, with `-1<=steer<=1` (-1 being turning left and 1 being turning right). The second element means how much acceleration is applied, with `0<=gas<=1` (0 being no acceleration and 1 being full acceleration). Finally, the third element indicates how much of the brake is applied, with `0<=brake<=1` (0 being no brake applied and 1 being applying the full brake).

Regarding the last point, note that since both the action space and state space are Box spaces, this means that the actor may enter continuous values, so this means that we may have an action in the form $[-0.7, 0.5, 0.36354]$. Of course, this makes no sense whatsoever for a human expert driver, but is still possible to be entered. Furthermore, it is not recommended to do this by the environment's documentation, as the combination of acceleration and turning will most likely veer the car off track (ref. Figure 7.1).

It is in this better controlled environment in which we will partly conclude our application of the A3C algorithm. Particularly, we are interested in creating our own scripts of code in Tensorflow, as this will make it easier to transfer to the work being done by the ADAS team. Our first attempt at solving this environment using DQN can be found at `https://github.com/PDillis/DQN-CarRacing/`. Our current work in progress using the A3C can be found at `https://github.com/PDillis/Experiment-CarRacing`.

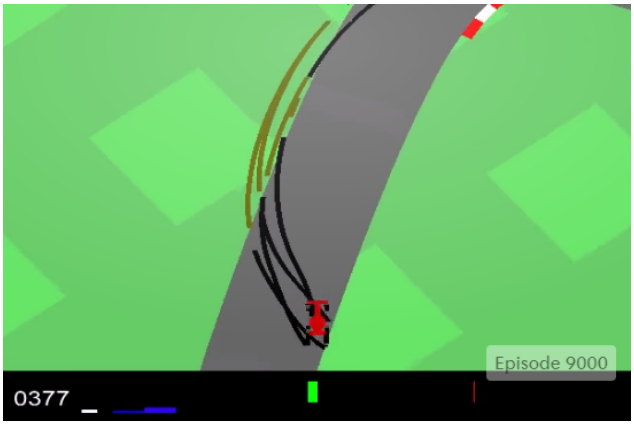


Figure 7.1 In the *CarRacing-v0* environment in Gym, it is quite easy to lose control of the vehicle due to it being a powerful rear-wheel car. It is recommended not to accelerate *and* turn at the same time for this reason.

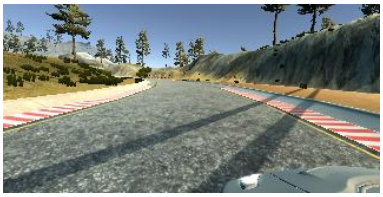
7.1.2 Unity - Udacity's Self-Driving Car Simulator

Unity is one of the leading game industry software. They offer a platform for creating 2D, 3D, VR and AR games and apps. They offer a powerful graphics engine and editor for over 25 platforms, including Windows, Linux, and macOS. This is so that virtual worlds are brought to life as easily and quickly as possible, as well as shared between platforms. As such, it has been chosen by both ADAS and Udacity to develop the virtual world where their self-driving cars will inhabit. The former's is under an NDA, and as such, we will only focus on the latter, where we hope it can be easily transferred what we learn to the former.

For the reasons mentioned before, Udacity has created their environment in which to train their self-driving car. However, due to it being built by students from around the world, they have decided to make it open-source (Cameron 2016). The source code, along with many challenges, can be found at `https://www.udacity.com/self-driving-car`.

This simulator works by positioning 3 virtual cameras on the front, left, and right sides of the car, which will record as the user drives the car along the track and record the data from the drive to try and mimic the user's driving (ref. Figure 7.2). As such, it is recommended to do from 1 to 5 laps around the course of which there are two available, the Jungle and Lake tracks. The ideal number of laps, according to other users, would be 5. The data recorded will be the associated steering angle, speed, throttle and brake for each frame in the 3 cameras. The data is saved to a CSV file, and in order for the movements to be as smooth as possible, they recommend using a joystick or steering wheel, though the keyboard suffices for our purposes.

It is this environment which we wish to edit to apply the A3C algorithm. Doing so will ensure an easier transition for it to be applied to ADAS's Unity simulator in the following time after this project is finished, in order to ensure the group at ADAS has another tool to test their environment.



(a) Left camera attached to the simulated car.

(b) Center camera attached to the simulated car.

(c) Right camera attached to the simulated car.

(d) Overhead camera; this is how the user interacts with the car and it's GUI.

Figure 7.2 Udacity's open-sourced self-driving car simulator in Unity. a)-c) are frames from the video taken at the same time that each of the virtual cameras located on top of the car will record.

7.2 CPU Cluster

In order for us to be able to fully exploit the advantages of the A3C algorithm, it would be best to have more CPU cores available in order to have more agent training asynchronously. For this reason we will use the Department of Computer Science (DCC) and Computer Vision Center (CVC) Cluster. We may run up to 30 running jobs and as many pending jobs as the cluster allows.

The DCC Cluster is formed by 28 computer nodes under a head node, all interconected by an ethernet network. Each of the nodes have 2×8 processor cores with hyperthreading enabled (32 virtual cores per node), 64 GB of memory, 1 TB disk drive, 1 GB ethernet connection, bringing to a total of 896 virtual cores with an assignment of 2 GB of memory for each core. On the other hand, the head node has the same specifications except for the hard disks and ethernet connection.

Thus, we can see that we have sufficient resources to run the experiments we wish to in the future, and not depend on a single local computer to run our experiments. There are also GPUs available, though we have not yet seen their specifications and may not even have the time to use them. Thus, we will focus only on the CPU cores so as to exploit the advantage of the A3C algorithm.

7.3 Finding the Optimal Learning Rate

Finding the correct hyperparameters is perhaps one of the most time-consuming and tedious tasks in Machine Learning, especially if there are no large resources for computing different combinations of these. While there are many techniques to find the optimal combinations of hyperparameters, the A3C algorithm needs to be fine-tuned for each individual environment that it is being trained on. That is, no two environments will be optimal with the same combination of hyperparameters, though in some cases they will be similar.

The A3C algorithm employs the so-called RMSProp (Tieleman & Hinton 2012) with shared statistics for optimization. Perhaps another option to find the best value for the learning rate would be to use what is proposed by (Schaul et al. 2013) and apply it to the RMSProp since the authors apply it to Stochastic Gradient Descent (SGD). This would remove the need for learning-rate tuning. However, we wish to keep using RMSProp with shared statistics across threads since it was proven in (Mnih et al. 2016) that it is more robust than, in particular, SGD.

7.4 Applying a GPU

One possible extension of our work is to apply the A3C algorithm to a GPU, as detailed in (Babaeizadeh et al. 2016, 2017). The authors have found that their hybrid implementation of A3C, named GA3C, can generate and consume data from $\sim 6\times$ faster using a small Deep Neural Network (DNN) up to $\sim 45\times$ for larger DNNs. The authors used two metrics, the Predictions Per Second (PPS) and the Trainings Per Second (TPS) to test and compare their algorithm to the A3C.

The main components of the GA3C are a DNN with training and prediction on a GPU, as well as a multi-process, multi-thread CPU architecture with an *agent*, *predictor*, and a *trainer* (ref. Figure 7.3); we can see that the GA3C maintains only one copy of the DNN model. When compared to the A3C, the GA3C has faster convergence towards the maximum score in a shorter time (Pong, among others), convergence towards a better score in a larger amount of time (QBert, for example), or for other games, a slower convergence (Breakout, for example).

Finally, (Mnih et al. 2016) note that asynchronous methods achieve significant speedups from using a larger number of agents (hence their testing with Asynchronous 1-step and n-step Q-Learning). The optimal configurations for the GA3C algorithm use a much larger number of agents compared to the CPU counterpart, which suggests that the GPU implementations of asynchronous learning methods may benefit from both a larger TPS and collecting experience using a higher number of agents.

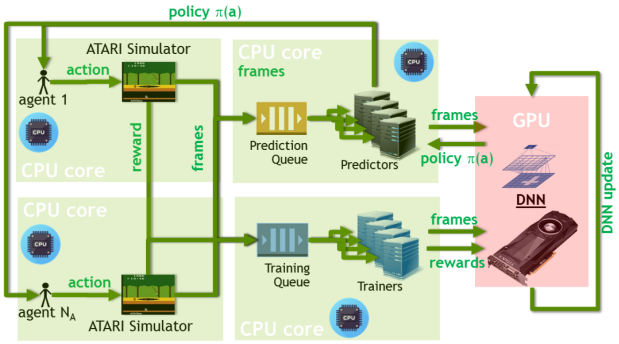


Figure 7.3 Architecture of the GA3C algorithm. The agents use predictors to query the network for policies while trainers gather experiences for network updates. Reproduced, with permission, from (Babaeizadeh et al. 2017).

On the other hand, a team of the University of Toronto and OpenAI has released a paper where they specify the baselines for two new algorithms: the ACKTR (pronounced "actor") and A2C for exploiting the use of a GPU (Wu et al. 2017, OpenAI 2017*e*). They are both actor-critic algorithms, but use different structures: ACKTR combines A3C (Mnih et al. 2016), Trust Region Policy Optimization (TRPO) (Schulman, Levine, Moritz, Jordan & Abbeel 2015), and distributed Kronecker factorization (K-FAC) (Ba et al. 2017).

At the same time, A2C is a synchronous version of A3C that waits for each actor to finish its segment of experience before doing an update. This results in larger batch sizes which in turn perform better with GPUs. When using single-GPU machines, the A2C implementation is more cost-effective than an A3C implementation, and it is faster than a CPU-only A3C implementation for large policies.

The ACKTR algorithm can learn continuous control tasks from low-resolution pixel inputs (ref. Figure 7.4). On the other hand, A3C could only obtain results from simple tasks, relatively speaking, such as in the Pendulum, Pointmass2D, and Gripper environments. Finally, the main advantage of the ACKTR algorithm comes when using large batch sizes, especially when comparing it to the A2C (Wu et al. 2017).

(a) (b)

Figure 7.4 a) ACKTR can learn to move a robotic arm to a designated position only through using low-resolution images. Adapted from the accompanying videos of (OpenAI 2017*e*).

7.5 Evolution Strategies

Another promising algorithm is the so-called Evolution Strategies (ES) which solve the Reinforcement Learning problems, as we have discussed before in Section 2.2. ES are, on the surface, similar to the Reinforcement Learning algorithms we have seen, but relatively easier to implement. They present a more scalable option, as well as being invariant to action frequency and delayed rewards, having fewer hyperparameters, and do not need temporal discounting or value function approximation (Salimans et al. 2017, OpenAI 2017*b*).

The authors have found that Evolution Strategies pose an interesting alternative to Q learning and even Policy Gradients. The only offset is that ES need from 3 to 10 times the amount of data to achieve the same levels of scoring achieved by A3C in Atari 2600 environments. However, perhaps surprisingly, 1-hour implementation of ES require equivalent computation power to a 1-day implementation of A3C, while performing better on 23 games tested, and worse on 28. In quantity of frames, this means that all games were trained for 10^9 frames using ES while using the same neural network computation as the implementation of the A3C of (Mnih et al. 2016) that trained each game for 3.2×10^6 frames. The larger the network used, the better the results obtained with ES (Salimans et al. 2017).

Since ES require negligible communication between workers, the authors were able to solve the 3D humanoid task of MuJoCo (Figure 7.5) using 1,440 CPUs across 80 machines in 10 minutes. In contrast, a setting of 32 A3C workers on one machine solved this task in roughly 10 hours. This shows that neuroevolution approaches offer competitive approaches to solving Reinforcement Learning problems, comparing to state-of-the-art benchmarks.

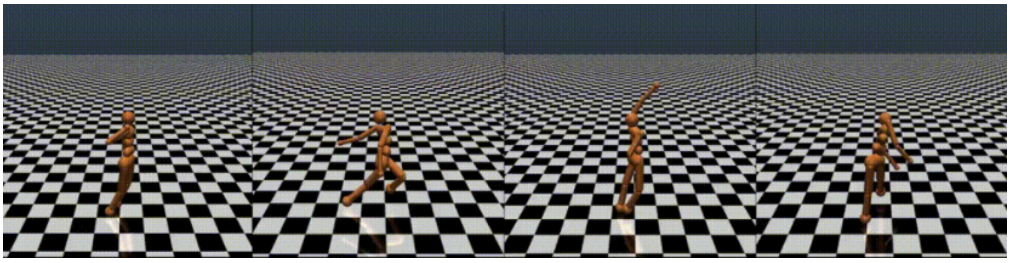


Figure 7.5 Four different local optimum attained in MuJoCo's environment of a 3D humanoid by the Evolutionary Method. Taken from (Salimans et al. 2017).

7.6 Reinforcement Learning Teacher

On our final entry to our repertoire of latest advancements in RL, there are many environments in which the reward function is too complex to determine. Perhaps there are too many variables, or the designers and programmers are not taking into account correctly all the parameters that will affect the behavior of our agent. This is precisely the case in self-driving cars, as we have discussed in Chapter 6, for which the reward functions are especially difficult to tune, mainly if we wish a particular behavior for our agent to manifest.

With this in mind, OpenAI has released ***RL-Teacher*** (OpenAI 2017*c*) which is an implementation of (Christiano et al. 2017). In it, the authors have applied their algorithm to different types of environments, one of which was the Atari 2600 game Enduro, a racing game, and found out that it outperforms the A3C algorithm, reaching the levels of the DQN algorithm where the A3C previously failed.

RL-Teacher is an open source implementation in which, using a remote server, the interface asks for human feedback in 1% of the interactions of the agent with its environment. The behaviors of the agents trained in 1 hour were considerably more complex than any other previously used. Indeed, using simple reward functions might result in complex behaviour for our agent, but at the end the agent only wishes to optimize the reward function without necessarily satisfying the whole final goal for which they were placed in the environment in the first place (ref. Chapter 5 and Section 6.4).

This algorithm would solve our problem encountered in Chapter 5, as the main problem with our environments was of it not performing the way we desired, e.g., finishing the race for the Wheelers environment, much less winning it. Indeed, many complicated behaviors have been trained using the RL-Teacher, including teaching the `Walker` environment of MuJoCo to dance like a ballerina instead of walking forward. Another example is the `Hopper` environment which the agent should learn how to hop forward, but with the RL-Teacher, the agent instead learned how to perform a backflip. We can easily see this algorithm outperforming any current result if we apply it to the `CarRacing-v0` environment.

7.7 Conclusions

The world of Reinforcement Learning is moving quickly, as can be seen from the last section. Indeed, while Reinforcement Learning is a giant field in and of itself, it is also being advanced indirectly through many other fields as seen in Section 1. Perhaps the area everyone is interested in is Artificial Intelligence, and this is where all the major breakthroughs are coming from, especially since everyone wishes to accomplish the much sought-after Artificial General Intelligence.

All of the fields in Machine Learning will benefit of this, but especially for our interest is the advancement of self-driving vehicles. It is perhaps crucial that we reach the level of AGI in this area, as vehicular safety is of paramount importance. Indeed, one cannot train an agent in all possible scenarios the self-driving car might find itself in. Some of these might even seem trivial to a human driver but not for the autonomous vehicle. This would have with disastrous consequences if trained incorrectly, in particular if the scenario is one in which a human driver might struggle.

It should be emphasized what what was discussed in Section 6.4. While Reinforcement Learning is being seen as simply another area of Machine Learning, it is also an important it would be wise to tackle problems—or at least define a *method* on how to tackle them—before they even appear. As the algorithms progress and the systems become more and more complex, these problems will become more apparent, and it should be wise to know how to deal with them before they become too big to deal with.

As seen in Chapter 6, while there are many approaches to apply Reinforcement Learning to self-driving cars, there hasn't yet been a major breakthrough in the area, at least not known to the public. While many things may be blocked behind a patent or an NDA, the major breakthroughs done by Reinforcement Learning in the last months will attract more researchers and more investment from both the public and the private sectors. It is in our best interest to advance this as efficiently and correctly as possible, and we shall make it so.

Appendix A

Pseudocodes

The pseudocode for the Q-Learning algorithm, as presented in (Sutton & Barto 1998).

Algorithm 1 One-step Q-Learning

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(terminal\text{–}state, \cdot) = 0$
repeat
  Initialize S
  repeat
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$
    $S \leftarrow S'$
  until done for each step of episode **or** S is terminal
until done for each episode

We present the pseudocode of the asynchronous algorithms as they where presented in (Mnih et al. 2016):

Algorithm 2 Asynchronous one-step Q-Learning

// Assume global shared θ, θ^- and counter $T = 0$
Initialize thread step counter $t \leftarrow 0$
Initialize target network weights $\theta^- \leftarrow \theta$
Initialize network gradients $d\theta \leftarrow 0$
Get initial state s
repeat
  Take action a with $\epsilon-$greedy policy based on $Q(s,a;\theta)$
  Receive new state s' and reward r
  $y = \begin{cases} r & \text{for terminal } s' \\ r + \gamma \max_{a'} Q(s',a';\theta^-) & \text{for non-terminal } s' \end{cases}$
  Accumulate gradients w.r.t. θ: $d\theta \leftarrow d\theta + \partial(y - Q(s,a;\theta))^2 / \partial\theta$
  $s = s'$
  $T \leftarrow T + 1$ and $t \leftarrow t + 1$
  if $T \mod I_{target} == 0$ **then**
    Update the target network $\theta^- \leftarrow \theta$
  end if
  if $t \mod I_{AsyncUpdate} == 0$ or s is terminal **then**
    Perform asynchronous update of θ using $d\theta$
    Clear gradients $d\theta \leftarrow 0$
  end if
until $T > T_{max}$

Algorithm 3 Asynchronous n-step Q-Learning Actor-Critic (A3C)

// Assume global shared parameter vector θ, target parameter vector θ^- and global shared counter $T = 0$
Initialize thread step counter $t \leftarrow 1$
Initialize target network parameters $\theta^- \leftarrow \theta$
Initialize thread-specific parameters $\theta' = \theta$
Initialize network gradients $d\theta \leftarrow 0$
repeat
  Clear gradients: $d\theta \leftarrow 0$
  Synchronize thread-specific parameters $\theta' = \theta$
  $t_{start} = t$
  Get state s_t
  repeat
    Take action a_t according to the $\epsilon-$greedy policy based on $Q(s_t, a; \theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ \max_a Q(s_t, a; \theta^-) & \text{for non-terminal } s_t \text{// } \textit{Bootstrap from last state} \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients w.r.t. θ': $d\theta \leftarrow d\theta + \partial (R - Q(s_i, a_i; \theta'))^2 / \partial \theta'$
  end for
  Perform asynchronous update of θ using $d\theta$
  if $T \mod I_{target} == 0$ **then**
    $\theta^- \leftarrow \theta$
  end if
until $T > T_{max}$

Algorithm 4 Asynchronous Advantage Actor-Critic (A3C)

// Assume global shared parameter vectors θ and θ_v and global shared counter $T = 0$
// Assume thread-specific parameter vectors θ' and θ'_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 $\pi(a_t \mid s_t; \theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// } \textit{Bootstrap from last state} \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients w.r.t. θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i \mid s_i; \theta')(R - V(s_i, \theta'_v))$
    Accumulate gradients w.r.t. θ'_v: $d\theta_v \leftarrow d\theta_v + \partial (R - V(s_i, \theta'_v))^2 / \partial \theta'_v$
  end for
  Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$
until $T > T_{max}$

Bibliography

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. & Zheng, X. (2016), 'Tensorflow: Large-scale machine learning on heterogeneous distributed systems', *CoRR* **abs/1603.04467**.
URL: *http://arxiv.org/abs/1603.04467*

Abbeel, P., Coates, A., Quigley, M. & Ng, A. Y. (2007), An Application of Reinforcement Learning to Aerobatic Helicopter Flight, *in* P. B. Schölkopf, J. C. Platt & T. Hoffman, eds, 'Advances in Neural Information Processing Systems 19', MIT Press, pp. 1–8.
URL: *http://papers.nips.cc/paper/3151-an-application-of-reinforcement-learning-to-aerobatic-helicopter-flight.pdf*

Amodei, D., Olah, C., Steinhardt, J., Christiano, P., Schulman, J. & Mané, D. (2016), 'Concrete Problems in AI Safety', *CoRR* **abs/1606.06565**.
URL: *http://arxiv.org/abs/1606.06565*

Ba, J., Grosse, R. & Martens, J. (2017), Distributed second-order optimization using Kronecker-factored approximations, *in* 'Proceedings of the 5th International Conference on Learning Representations', ICLR 2017.

Babaeizadeh, M., Frosio, I., Tyree, S., Clemons, J. & Kautz, J. (2016), 'GA3C: GPU-based A3C for Deep Reinforcement Learning', *CoRR* **abs/1611.06256**.
URL: *http://arxiv.org/abs/1611.06256*

Babaeizadeh, M., Frosio, I., Tyree, S., Clemons, J. & Kautz, J. (2017), Reinforcement Learning Through Asynchronous Advantage Actor-Critic on a GPU, *in* 'Proceedings of the 5th International Conference on Learning Representations', ICLR 2017.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J. & Zaremba, W. (2016), 'OpenAI Gym', *CoRR* **abs/1606.01540**.
URL: *http://arxiv.org/abs/1606.01540*

Cameron, O. (2016), 'We're Building an Open Source Self-Driving Car'.
URL: *https://medium.com/udacity/were-building-an-open-source-self-driving-car-ac3e973cd163*

Christiano, P., Leike, J., Brown, T. B., Martic, M., Legg, S. & Amodei, D. (2017), 'Deep reinforcement learning from human preferences', *CoRR* **abs/1706.03740**.
URL: *https://arxiv.org/abs/1706.03740*

Clevert, D., Unterthiner, T. & Hochreiter, S. (2015), 'Fast and accurate deep network learning by exponential linear units (elus)', *CoRR* **abs/1511.07289**.
URL: *http://arxiv.org/abs/1511.07289*

Coates, A., Abbeel, P. & Ng, A. Y. (2008), Learning for Control from Multiple Demonstrations, *in* 'Proceedings of the 25th International Conference on Machine Learning', ICML '08, ACM, New York, NY, USA, pp. 144–151.
URL: *http://doi.acm.org/10.1145/1390156.1390175*

Gosavi, A. (2015), *Solving Markov Decision Processes via Simulation*, Springer New York, New York, NY, pp. 341–379.
URL: *https://doi.org/10.1007/978-1-4939-1384-8_13*

Kaelbling, L. P., Littman, M. L. & Moore, A. (1996), 'Reinforcement Learning: A Survey', *Journal of Articial Intelligence Research* **4**, 237–285.

Kingma, D. P. & Ba, J. (2014), 'Adam: A method for stochastic optimization', *CoRR* **abs/1412.6980**.
URL: *http://arxiv.org/abs/1412.6980*

Lake, B. M., Ullman, T. D., Tenenbaum, J. B. & Gershman, S. J. (2016), 'Building machines that learn and think like people', *CoRR* **abs/1604.00289**.
URL: *http://arxiv.org/abs/1604.00289*

Lapan, M. (2017), Deep Reinforcement Learning: theory, intuition, code, *in* 'Proceedings of PyData - Amsterdam 2017'.

Lau, B. (2016), 'Using Keras and Deep Deterministic Policy Gradient to play TORCS'.
URL: *https://yanpanlau.github.io/2016/10/11/Torcs-Keras.html*

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D. & Wierstra, D. (2015), 'Continuous control with deep reinforcement learning', *CoRR* **abs/1509.02971**.
URL: *http://arxiv.org/abs/1509.02971*

Littman, M., Isbell, C. & Kolhe, P. (2012), 'Lecture notes in Machine Learning'.

McLeod, S. (2007), 'Edward Thorndike'.
URL: *https://www.simplypsychology.org/edward-thorndike.html*

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D. & Kavukcuoglu, K. (2016), 'Asynchronous Methods for Deep Reinforcement Learning', *CoRR* **abs/1602.01783**.
URL: *http://arxiv.org/abs/1602.01783*

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S. & Hassabis, D. (2015), 'Human-level control through deep reinforcement learning', *Nature* **518**, 529–533.

OpenAI (2016), 'Universe'.
URL: *https://blog.openai.com/universe/*

OpenAI (2017*a*), 'Dota 2'.
URL: *https://blog.openai.com/dota-2/*

OpenAI (2017*b*), 'Evolution Strategies as a Scalable Alternative to Reinforcement Learning'.
URL: *https://blog.openai.com/evolution-strategies/*

OpenAI (2017*c*), 'Gathering Human Feedback'.
URL: *https://blog.openai.com/gathering_human_feedback/*

OpenAI (2017*d*), 'More on Dota 2'.
URL: *https://blog.openai.com/more-on-dota-2/*

OpenAI (2017*e*), 'OpenAI Baselines: ACKTR & A2C'.
URL: *https://blog.openai.com/baselines-acktr-a2c/*

Pereyra, G., Tucker, G., Chorowski, J., Kaiser, L. & Hinton, G. E. (2017), 'Regularizing neural networks by penalizing confident output distributions', *CoRR* **abs/1701.06548**.
URL: *http://arxiv.org/abs/1701.06548*

Perot, E., Jaritz, M., Toromanoff, M. & de Charette, R. (2017), End-To-End Driving in a Realistic Racing Game With Deep Reinforcement Learning, *in* 'The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops', pp. 3–4.

Piñol, M. (2014), Reinforcement Learning of Visual Descriptors for Object Recognition, PhD thesis, Universitat Autònoma de Barcelona.

SAE International (2016), 'Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles'.
URL: *http://standards.sae.org/j3016_201609/*

Salimans, T., Ho, J., Chen, X. & Sutskever, I. (2017), 'Evolution Strategies as a Scalable Alternative to Reinforcement Learning', *CoRR* .
URL: *https://arxiv.org/abs/1703.03864*

Sallab, A. E., Abdou, M., Perot, E. & Yogamani, S. (2017), Deep Reinforcement Learning framework for Autonomous Driving, *in* 'IS&T International Symposium on Electric Imaging 2017: Autonomous Vehicles and Machines', Society for Imaging Science and Technology, pp. 70–76.

Schaul, T., Zhang, S. & LeCun, Y. (2013), 'No More Pesky Learning Rates', *CoRR* **abs/1206.1106**.
URL: *https://arxiv.org/abs/1206.1106*

Schulman, J., Levine, S., Moritz, P., Jordan, M. I. & Abbeel, P. (2015), 'Trust region policy optimization', *CoRR* **abs/1502.05477**.
URL: *http://arxiv.org/abs/1502.05477*

Schulman, J., Moritz, P., Levine, S., Jordan, M. I. & Abbeel, P. (2015), 'High-Dimensional Continuous Control Using Generalized Advantage Estimation', *CoRR* **abs/1506.02438**.
URL: *http://arxiv.org/abs/1506.02438*

Shalev-Shwartz, S., Shammah, S. & Shashua, A. (2016), 'Safe, Multi-Agent, Reinforcement Learning for Autonomous Driving', *CoRR* **abs/1610.03295**.
URL: *http://arxiv.org/abs/1610.03295*

Silver, D. (2015), 'UCL Course on Reinforcement Learning Lecture Notes'.
URL: *http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html*

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T. & Hassabis, D. (2016), 'Mastering the Game of Go with Deep Neural Networks and Tree Search', *Nature* **529**(7587), 484–489.

Sutton, R. S. & Barto, A. G. (1998), *Reinforcement Learning: An Introduction*, The MIT Press, Cambridge Univ Press, MA (USA).

Szepesvári, C. (2009), *Algorithms for Reinforcement Learning*, Morgan & Claypool.

Thorndike, E. (1898), 'Animal Intelligence: An experimental study of the associative processes in animals', *Psychological Monographs: General and Applied* **2**(4), i–109.

Thrun, S., Montemerlo, M., Dahlkamp, H., Stavens, D., Aron, A., Diebel, J., Fong, P., Gale, J., Halpenny, M., Hoffmann, G., Lau, K., Oakley, C., Palatucci, M., Pratt, V., Stang, P., Strohband, S., Dupont, C., Jendrossek, L.-E., Koelen, C., Markey, C., Rummel, C., van Niekerk, J., Jensen, E., Alessandrini, P., Bradski, G., Davies, B., Ettinger, S., Kaehler, A., Nefian, A. & Mahoney, P. (2006), 'Stanley: The robot that won the DARPA Grand Challenge', *Journal of Field Robotics* **23**(9), 661–692.
URL: *http://dx.doi.org/10.1002/rob.20147*

Tieleman, T. & Hinton, G. (2012), 'Lecture 6e- rmsprop: Divide the gradient by a running average of its recent magnitude'. COURSERA: Neural Networks for Machine Learning.
URL: *http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf*

Ulbrich, S. & Maurer, M. (2013), Probabilistic online POMDP decision making for lane changes in fully automated driving, *in* '16th International IEEE Conference on Intelligent Transportation Systems (ITSC 2013)', pp. 2063–2067.

Watkins, C. (1989), Learning from delayed rewards, PhD thesis, University of Cambridge, England.

Watkins, C. & Dayan, P. (1992), 'Technical Note: Q-Learning', *Machine Learning* **8**(3), 279–292.
URL: *https://doi.org/10.1023/A:1022676722315*

Williams, R. J. (1992), 'Simple statistical gradient-following algorithms for connectionist reinforcement learning', *Machine Learning* **8**(3), 229–256.

Wu, Y., Mansimov, E., Liao, S., Grosse, R. & Ba, J. (2017), 'Scalable trust-region method for deep reinforcement learning using Kronecker-factored approximation', *CoRR* **abs/1708.05144**.
URL: *https://arxiv.org/abs/1708.05144*

You, Y., Pan, X., Wang, Z. & Lu, C. (2017), 'Virtual to Real Reinforcement Learning for Autonomous Driving', *CoRR* **abs/1704.03952**.
URL: *http://arxiv.org/abs/1704.03952*