# UnityRL: a Deep Reinforcement Learning Framework to train Characters in the Unity 3D Environment

Nihat Isik

Bachelor Thesis
05 September 2018

Prof. Dr. Stelian Coros
Dr. Vittorio Megaro

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

CRL

# Acknowledgements

I would like to thank my supervisor Dr. Vittorio Megaro for his inputs, ideas and support that proved to be really helpful during the development of this thesis. I also want to thank the Computational Robotics Lab (CRL).

# Abstract

Unity 3D is a game development environment which became famous among masses thanks to its easy-to-use interface which allows even non-expert users to quickly create games. Due to the recent success in using Machine Learning (ML) techniques to solve complex tasks (an example being Deep Reinforcement Learning for motion control) the Unity 3D environment now facilitates the communication between its 3D-physics environment and Tensorflow, a python-based Neural Network library. We would like to exploit the connection between the two frameworks to create an easy-to-set-up environment, to quickly apply various DeepRL algorithms on different characters. We will be focusing specifically on cars for stability reasons in Unity, however the framework can easily be adapted to also work with articulated characters.

# Contents

# List of Figures

# List of Tables

# 1

# Introduction

With the continuously growing computing power and resources, such as high performance and accessible GPUs, different computing methods, that had little practical use in the early days of research, started becoming more applicable and widely used. One field of computer science that had a remarkable growth is Machine Learning with its constantly increasing community and frameworks that facilitate the usage of a whole variety of training algorithms. We distinguish mainly two fields in Machine Learning, *supervised* and *unsupervised* learning. The former is used to train a model based on a given data set in order to predict new unlabeled data, with the latter, instead, similarities and differences between data-points are found in order to partition the data meaningfully. In addition to that we have the *reinforcement learning problem*. During the last years a lot of methods to solve this problem have been discovered and successfully applied to some research areas. We could for example mention physics based character animation, a big research field that is constantly having new compelling results based on these methods or slight variations of it.

We could ask ourselves what makes reinforcement learning methods so different from others. It is different from supervised learning, since the latter is basically learning from examples provided by a reliable source, and thus cannot be used alone for learning from interaction where we want our agent (the learner) to learn from its own experience. In general we define an agent to have an explicit goal, we give it the knowledge of some aspects of the environment and the power to interact with it. To be more specific, besides the environment and the actions, our system is formed by a *policy*, a *reward function* and a *value function*. What we try to achieve during the learning process is to adapt the policy (which maps states to actions) in order to maximize the total reward.

The easiest approach that could pop in mind to solve this problem is a greedy approach, thus selecting the highest immediate reward action in every given state. With this solution we won't necessarily get the maximum total reward. We have to consider the fact that one action could also affect the rewards of the subsequent actions. In fact, our first estimation of cumulated

reward can be off. With this in mind another issue arises, the trade-off between *exploitation and exploration*. A variation of the greedy approach that could lead to better results, increasing exploration, is to select a random action with small probability $\epsilon$ ($\epsilon$-*Greedy* approach).

How can we take into consideration so many factors when trying to update our agents policy in order to choose actions that result in the maximum total reward? A popular choice to approximate the optimal policy (the one able to collect the maximum cumulated reward) is to use neural networks. When neural networks with multiple layers are employed to approximate functions, the learning method gets the name of Deep Reinforcement Learning. Putting all this knowledge together brings us to the most used algorithms today, which are *Deep Q-Learning* and *PPO* that are going to be introduced in chapter 3.

Since the interest about deep reinforcement learning has increased, mostly because of successful examples at Google DeepMind and OpenAI, other companies decided to dive into this world. Unity3D, an easy to use game engine, published the first version of their framework *ML-Agents* in 19 September 2017. A PPO implementation lets you train agents placed in environments created directly into the Unity Editor. This framework is what we will be using to experiment different training settings and try to understand how the training is influenced by variations of the reward system and the overall implementation.

# 2

# Related Work

Deep Reinforcement Learning has been used lately for a lot of research areas from visual computing, graphics, natural language processing, music generation and much more. In these field we can see intelligent agents playing a whole variety of games (see DeepMind [Dee17]), or agents learning to walk, run or vault over objects. In this chapter we discuss some of the related work to this thesis.

## 2.1 Deep Reinforcement Learning

The first algorithm that had a major success before PPO was introduced and became the state of the art algorithm for learning agents in environments with continuous state and action was DQN. It was introduced by the team of DeepMind [MKS+13] where they used Neural Networks as a function approximator to estimate the optimal Q value from which the estimate of the optimal policy can be derived. This paper had a deep impact in future research that later on brought to the implementation of PPO. In addition to DQN, other algorithms, such as actor critic methods, managed to achieve really good results and finally helped the formulation of PPO. The introduction of (A3C)[MBM+16] and (A2C) [Ope17b] took advantage of using multiple agents in order to solve a problem using actor critic methods.

During the process of training agents we will try to see the actual difference when learning with more than one agent in chapter 4.

## 2.2 Robotics

Often programming robots requires a lot of time and it is an iterative manual process of adjusting and tuning parameters in order for the robot to behave in the right way. In the same context it is actually also hard for humans to come up with equations that describe a complete behaviour of a robot. This is where Reinforcement Learning can be helpful. It seems like a good trade off to write the high level specification of a robot and let it learn from the world by acquiring experience[SK02].

Reinforcement Learning can also be used in combination with vision, by allowing to take visual inputs from robots. These vision-based RL techniques can be used for example to teach a robot to shoot a ball into a goal [ANTH96]. Recall that also the first use case of DQN used Recurrent Neural Networks in order to transform the raw pixels of the games into something that made sense to the agent [MKS+13]. Before actually trying to train a robot in real world one can simulate it and run a RL algorithm to see how it might perform. Thanks to the framework proposed by Unity, one can easily manage to create 3D objects and train them.

## 2.3 Physics Based Character Animation

Traditionally, the motion trajectories of virtual characters and objects are hand-crafted by skilled animators, this is not only a time-consming process, but also results in animations that are purely kinematic: they have no regard for force or mass. This means that the end result is purely in the hand of the animators, which has to personally take care of those aspects in order to have satisfying animations. Progress has been made in creating motion from control and physics as a means of animations [GP12], but most of the time the research is focused on motion on flat terrains, where there isn't much space for variation.

Deep neural networks in combination with reinforcement learning (DeepRL) has showed to have a lot of success when applied to this kind of problem. Using mixture of actor-critic experts (MACE) architecture to enable accelerated learning to train agents to walk on uneven terrains. Peng et al. [PBvdP16].

# 3

# The Reinforcement Learning Problem

## 3.1 Definitions

Before starting off to explain the approaches that can be taken in order to solve the reinforcement learning problem we want to define the *reinforcement learning system:*

- **Agent**: learner and decision maker.
- **Environment**: the surrounding the agent interacts with, comprising everything outside the agent.
- **Policy**: the agents way of behaving at a given time. More specifically a mapping from percieved states of the environment to actions to be taken when in those states.
- **Reward function**: maps each percieved state of the environment to a scalar, the *reward*, indicating the deisarbility of that state. Defines the goal of a RL problem.
- **Value function**: in contrast from the reward function which specifies which state is good in an immediate sense, the value function specifies which state is good in the long run.

At this point we need to describe how all these concepts interact together to solve the reinforcement learning problem.

The agent interacts with the environment at each of a sequence of discrete time steps, $t = 0, 1, 2, 3, \ldots, T$. The basic process at each time step $t$ is *(see figure* 3.1*)*:

1. the agent observes a *state* $S_t \in \mathcal{S}$ (where $\mathcal{S}$ is the set of possible states) and, if the agent performed an action in step $t-1$, a reward $R_t \in \mathcal{R} \subset \mathbb{R}$ (where $\mathcal{R}$ is the set of possible rewards).
2. the agent selects an *action*, $A_t \in \mathcal{A}(s_t)$, where $\mathcal{A}(s)$ is the set of all possible actions in state $s$.

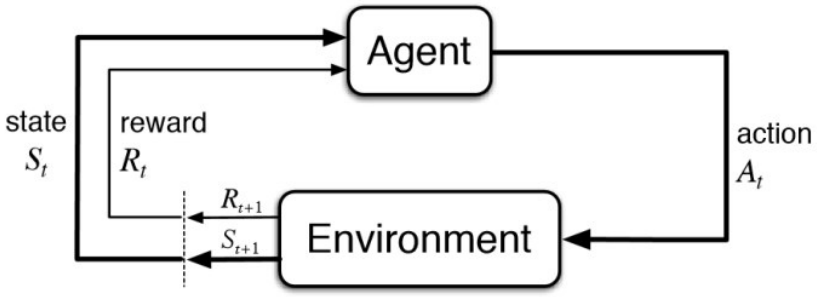


***Figure 3.1:*** *Agent-Environment interaction in RL [SB18]*

This basically creates a sequence that looks like this: $(S_0, A_0, R_1), (S_1, A_1, R_2), (S_2, A_2, R_3), \ldots$

The actions are selected based on the policy that we denote as $\pi$, where $\pi(a|s)$ is the probability that we select action $a$ (so $A_t = a$) when being in state $s$ (when $S_t = s$). Through these iterations the policy gets updated in order to achieve the highest reward in the long run. How the policy gets updated depends on the RL method used.

Before explaining these methods we want to discuss more concretely on how we can differentiate the quality between two given states. The goodness of a state is given by the total amount of reward that we can get starting from that state. This is also called the *expected return* and we compute it using the value function.

## 3.2 MDPs and Value Functions

### 3.2.1 Markov Decision Processes

The problems we will discuss are *Markov Decision Processes*, this means that each state retains all relevant information such that the *transition probabilities* are only based on the current state and action. For all values $s' \in \mathcal{S}$ and $r \in \mathcal{R}$, there is a probability of those values occurring at a time t, given the values of the previous state, $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$

$$p(s', r|s, a) = Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\} \tag{3.1}$$

This function is an ordinary deterministic function of four arguments $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$. State signals with this property are said to have the *Markov property*. The same concept applies for the expected value of the reward at time step $t+1$. We denote this function as $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$

$$r(s, a) = \mathbb{E}[R_t|S_{t-1} = s, A_{t-1} = a] = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s', r|s, a) \tag{3.2}$$

During the next chapters we assume that our environment behaves like a MDP.

### 3.2.2 Value Functions

As we mentioned previously, these functions define how good it is to be in a certain state. An intuitive way of thinking about it is *"what is the total amount of reward I could get starting from this state?"*. In fact we will be using something similar that is called *discounted return*:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{3.3}$$

This equation defines the sum of the discounted future rewards starting from a time step t. The discount value is specified by the $\gamma \in \big[0, 1\big]$. This value is used to weight the rewards such that the immediate rewards have a bigger impact on the final result than the future rewards. Also notice that if $\gamma = 1$ then the sum could be infinite which is something we don't want. So if we're treating an *episodic task* problem, namely a problem with a terminal state for each episode then $\gamma = 1$ is acceptable since the return would be finite. On the other hand, if we're treating a *continuing task* problem, where we have no terminal state, then we need the constraint $\gamma < 1$. It is important to notice that the discounted return can also be defined recursively

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \dots) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{3.4}$$

From the equation 3.3 we define the *state-value function*:

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s] = \mathbb{E}_\pi \left[ \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \Big| S_t = s \right], \forall s \in \mathcal{S} \tag{3.5}$$

which takes as a parameter a state *s* and it computes the *expected discounted return* given a policy $\pi$. In the case there exists a terminal state the value of that state will always be zero. If we also know the action taken in state *s* then we can define the *action-value function*:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a] = \mathbb{E}_\pi \left[ \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \Big| S_t = s, A_t = a \right], \forall s \in \mathcal{S}, a \in \mathcal{A}(s) \tag{3.6}$$

The general idea is trying to estimate these functions and use them to generate an optimal policy for the given problem. In the next section we describe how to estimate these functions.

## 3.3 The Bellman Equation

In order to approximate the value functions we use the *Bellman Equation:*

$$
\begin{aligned}
v_\pi(s) &\overset{3.5}{=} \mathbb{E}_\pi[G_t|S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] \\
&= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \Big[ r + \gamma \mathbb{E}_\pi \left[ G_{t+1}|S_{t+1} = s' \right] \Big] \\
&= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \Big[ r + \gamma v_\pi(s') \Big], \forall s \in \mathcal{S}
\end{aligned} \tag{3.7}
$$

This equation represents a relationship between the value of a state and the value of its successor states [Bel13].

From state *s*, different actions may be chosen (given by $\pi(a|s)$). For each selected action we may land in different states *s'* and get different rewards $r$ that have an occurring probability of $p(s', r|s, a)$. So this equation averages over all the possibilities, weighting each by its probability of occurring. Finally the value of $s$ is equal to the discounted value of the expected next state, plus the reward expected along the way.

An optimal policy $\pi_*$ is defined as the policy with highest expected return compared to other policies (although we might have multiple optimal policies). This is the same as saying that it must have the optimal *value function* $v_*$(s) for all states. The *Bellman Optimality Equation* defines in fact the value function as independent from any policy:

$$
v_*(s) = \max_{a \in \mathcal{A}(s)} q_*(s, a) = \max_a \sum_{s',r} p(s', r|s, a) \left[ r + \gamma v_*(s') \right] \tag{3.8}
$$

$$
q_*(s, a) = \sum_{s',r} p(s', r|s, a) \left[ r + \max_{a'} \gamma q_*(s', a') \right] \tag{3.9}
$$

This equation is fundamental in order to find an optimal policy as we will see in the next chapters.

## 3.4 Value-Based Methods

Generally value-based methods find an optimal policy using iterative methods by adopting the value function.

### 3.4.1 Policy iteration

The first approach we consider is called *policy iteration* and it is composed of mainly two parts:

**Policy evaluation**: also called the *prediction problem* where the state-value function $v_\pi$ gets evaluated using an iterative approach. That means starting from an initial arbitrarily chosen approximation $v_0$ we derive successive approximations using the equation 3.7 as an update rule.

**Policy improvement**: after policy evaluation, we improve the current policy by assigning to each state the action that maximizes the state-action value. With that we mean

$$\begin{aligned}\pi'(s) &= \underset{a}{\text{argmax}}\ q_\pi(s,a) \\ &= \underset{a}{\text{argmax}}\ \mathbb{E}_\pi\{R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a\} \\ &= \underset{a}{\text{argmax}} \sum_{s',r} p(s', r|s, a)\,[r + \gamma v_\pi(s')] \end{aligned} \tag{3.10}$$

this means that the policy learns to choose deterministically an action a when in state s that maximizes the value (see 3.8 for derivation of 3.10).

***1. Initialization***
$V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

***2. Policy Evaluation***
**Repeat**
$\Delta \leftarrow 0$
**For each** $s \in \mathcal{S}$ **do**
$v \leftarrow V(s)$
$V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))\Big[r + \gamma V(s')\Big]$
$\Delta \leftarrow max(\Delta, |v - V(s)|)$

**until** $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

***3. Policy Improvement***
*policy-stable* $\leftarrow$ true
**For each** $s \in \mathcal{S}$ **do**
*old-action* $\leftarrow \pi(s)$
$\pi(s) \leftarrow \underset{a}{\text{argmax}} \sum_{s',r} p(s', r|s, a)\,[r + \gamma V(s')]$
If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ false

**If** *policy-stable*, then stop and **return** $V \approx v_*$ and $\pi \approx \pi_*$; **else** go to 2

**Algorithm 1:** Policy Iteration [SB18]

This algorithm converges often in few iterations [SB18] but a drawback is that for each iteration a complete policy evaluation has to be computed which can be pretty expensive for a big number of states. This is because policy evaluation may run many times before it converges. One way to get rid of this problem is to not rely on our policy estimate and instead greedily improve the value function.

### 3.4.2 Value iteration

This algorithm combines *policy improvement* with a truncated *policy evaluation*:

$$
\begin{aligned}
v_{k+1}(s) &= \max_a \mathbb{E}\left[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t = a\right] \\
&= \max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma v_k(s')\right], \ \forall s \in \mathcal{S}
\end{aligned} \tag{3.11}
$$

We update the value-function (which is initially random) by applying 3.11 to find an estimate of the optimal *value-function*. Once we're satisfied we extract the approximation of the *optimal policy* which will be computed using 3.10 for all states. Note that through this update rule the sequence $\{v_k\}$ still converges to the optimal $v_*$ for an infinite number of iteration [ZZ01]. The complete algorithm looks like this:

Parameters: small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s), \ \forall s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$
**Repeat**
  $\Delta \leftarrow 0$
  **For each** $s \in \mathcal{S}$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$

**until** $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \operatorname{argmax}_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$

**Algorithm 2:** Value Iteration [SB18]

### 3.4.3 Q-Learning

Q-Learning [WD92] is an example of a *model-free temporal difference (TD) method*, meaning that the state transition probability and the expected rewards are not known. The idea is similar to the previous algorithms in the sense that it still *bootstraps*, i.e. it updates the estimates based on other estimates. The difference is that each value gets adjusted based on the immediate next relevant value.

The algorithm works by learning the action-value function $Q$ based on a greedy policy, which directly approximates the optimal $q_*$. Since the updates of the *action-value function* are not based on the policy the agent is following ($\epsilon$-greedy) this method is labeled as an *Off-Policy* approach. The update rule for the action-value function is:

$$
Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right] \tag{3.12}
$$

A pseudo-code for understanding the algorithm is showed below.

Parameters: step size $\alpha \in [0, 1]$, small $\epsilon > 0$
Initialize $Q(s, a),\ \forall s \in \mathcal{S}^+,\ a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$
**For each** episode **do**
    Initialize $S$
    **For each** step of episode **do**
        Choose $A$ from $S$ using policy derived from Q (e.g., $\epsilon$-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** $S$ is terminal

**Algorithm 3:** Policy Iteration [SB18]

Note that the agent is following a policy that isn't greedy. As we briefly mentioned in the introduction Reinforcement Learning suffers from the trade-off between *exploration and exploitation*. Since we're dealing with approximations of policies following these policies greedily may lead the agent to never visit states that may turn out to be much more beneficial. To avoid landing on sub-optimal policies we use $\epsilon$-greedy actions. This means that for a small probability $\epsilon$ a random action $a \in \mathcal{A}(s)$ is chosen instead of the optimal one. This method has been shown to converge to an optimal solution [WD92].

The *On-Policy* variation of this algorithm is known as SARSA [CS15].

## 3.5 Policy Gradient Methods

The methods we discussed previously were all based on somehow estimating the value functions and from there extracting a policy. Policy gradient methods instead try to learn a *parameterized policy* function and optimize it directly instead of having the need of consulting the value function.

$$\pi(a|s; \theta) = P(A_t = a \mid S_t = s; \theta_t = \theta) \tag{3.13}$$

The general idea is to learn the parameters $\theta$ that maximize a policy objective function $J(\theta)$. This is the description of an *optimization* problem and most of the time it is solved with gradient ascent.

The main advantages of these methods are the better convergence properties and the effectiveness in high-dimensional or continuous action spaces. Nevertheless it comes with the disadvantage of typically converging to a local instead of a global optimum [Dav15].

The objective function has to have some information about the quality of our policy since it is being maximized. One simple function could be to use the start value in an episodic task $J_1(\theta) = v_{\pi_\theta}(S_1)$. And for this case the update rule for $\theta$ would be given from the policy gradient $\nabla_\theta J(\theta) = \mathbb{E}[\nabla_\theta log \pi(a|s; \theta) q_{\pi_\theta}(s, a)]$ (policy gradient theorem [SMSM00]) leading to the iterative update:

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta log \pi(a|s;\theta) q_{\pi_\theta}(s,a) \tag{3.14}$$

One of the first policy gradient methods is REINFORCE which uses the return $G_t$ as an unbiased sample of $q_{\pi_\theta}(s,a)$. It wasn't really used much because it performed poorly as a result of its high variance [SMSM00]. One way of improving this is adding a *critic* to estimate the action-value function which brings us to the concept of actor-critic learning that we introduce in chapter 3.6.2.

## 3.6 DeepRL: Deep Reinforcement Learning

Until now we discussed algorithms that worked well in case of problems with small complexity where the state space was small enough that computing the optimal value function in a search tree was possible. For games, such as Go [SHM+16] where the state space extends to $10^{170}$, such methods are infeasible. There is the need to introduce function approximators to generalize the learned behaviour also for unseen states. In DeepRL deep neural networks are used as function approximators.

### 3.6.1 DQN: Deep Q-Network

The first deep learning model that successfully managed to learn control policies from high-dimensional sensory input, raw pixels, without prior knowledge [MKS+13]. This model improves Q-Learning in different ways. First of all Q-Learning estimates the *action-value function* by experience and from this arises a lack of generality, meaning that for new, never encountered, states the agent wouldn't know which action to take (takes random actions). DQN solves the problem by introducing a neural network instead of the table for estimating the Q-value function.

The Q-Network gets as input raw pixels (the states) and using a Convolutional Neural Network it minimizes the mean-squared error between the Q-value approximation and the target

$$y = R_{t+1} + \gamma \max_a q(S_{t+1}, a; \theta_{t-1}) \tag{3.15}$$

where $\theta$ are the parameters being optimized. This yields the loss function:

$$L(\theta_t) = \mathbb{E}_{(S_t, A_t, R_{t+1}, S_{t+1})}[(y - q(S_t, A_t; \theta_t))^2] \tag{3.16}$$

and differentiating $L(\theta_t)$ with respect of $\theta$ we get the update rule for improving our Q-value function:

$$\nabla_{\theta_t} L(\theta_t) = \mathbb{E}_{(S_t, A_t, R_{t+1}, S_{t+1})}\left[\left(R_{t+1} + \gamma \max_a q(S_{t+1}, a; \theta_{t-1}) - q(S_t, A_t; \theta_t)\right) \nabla_{\theta_t} q(S_t, A_t; \theta_t)\right]$$

This algorithm is *model-free* and *off-policy* like the original Q-Learning algorithm [MKS+13].

There is a problem caused by the high correlation between actions and states which causes the weights to vary more drastically. The networks tends to forget previous experiences since it receives sequentially new experience to learn from. If we're dealing with a game that has different levels, this means that by learning to solve new levels (e.g. ground level and water level in super mario) the network might forget to solve the previous ones. To avoid this problem *Experience Replay* is used. The principle is to store previous experience $e_t(s_t, a_t, r_{t+1}, s_{t+1})$ inside the *experience buffer* and sample from this buffer randomly to feed the network in order to keep previous experience fresh. This also reduces the correlation between experiences [MKS+13] [Tho18].

Some variations of DQN has been made to improve it even further, for example using two networks, one to estimate the Q-target value with fixed weights and the other to estimate the Q-value that gets updated. This approach is used in order to stabilize the update process, because of the variance of the Q-target value (it's hard to chase something that is moving). Other improvements are known such as Dueling DQN which handles the problem of overestimation of Q-values and improves the stability [WdFL15].

### 3.6.2 Actor-Critic Methods

Until this point we discussed value based methods that estimated a value function in order to find a good policy and policy gradient methods that directly learned a policy without the need of estimating a value function. Actor-Critic methods are a combination of both methods that improve convergence speed [KT00].

In a nutshell actor-critic methods have two components:

- *Actor*: estimates a policy which the agent will be following and gets influenced by the critic for improvement
- *Critic*: estimates a value function and judges the actions taken by the agent

In other words the role of the critic is to determine what to change in the actor parameters in order to choose actions that would result in higher Q-values. In order to reduce variance of the gradient estimates the *advantage function* can be used [PS08]:

$$\begin{aligned} Adv(S_t, A_t) &= q(S_t, A_t) - v(S_t) \\ &= R_{t+1} + \gamma v(S_{t+1}) - v(S_t) \end{aligned} \tag{3.17}$$

Fortunately we can use TD (temporal difference) error instead of having to estimate also the state-value function. Recall that this requires only one time step to be determined (since in one time step we would get a reward $R_{t+1}$ and a new state $S_{t+1}$).

The equation tells us how much better was choosing action $a$ than expected (what is the extra reward I get). If $Adv(S_t, A_t) > 0$ the gradient is adjusted to a higher probability of selecting $A_t$

when in state $S_t$. Otherwise, if the value of the advantage is negative the probability of selecting $A_t$ will be lower.

A well known algorithm that applies this method is known as A2C where instead of using Experience Replay different instances of the same environment are used to update the same set of weights in a global network (synchronously)[Ope17c].

### 3.6.3 PPO: Proximity Policy Optimization

This algorithm is a major breakthrough in Reinforcement Learning for its easy implementation and very good results in training agents in a continuous state and action space. PPO improves TRPO [SLM+15] by changing it's objective function and achieving a better performance with a simpler implementation [SWD+17] [Ope17a].

The idea is to limit the change in the policy function when updating it to increase stability during training. In fact PPO accomplishes this task using the *Clipped Surrogate Objective* that replaces the original objective function of TRPO.

The objective function of TRPO is [SLM+15]:

$$L^{(TRPO)}(\theta) = \hat{\mathbb{E}}_t \left[ \frac{\pi(A_t|S_t;\theta)}{\pi_{old}(A_t|S_t;\theta)} \widehat{Adv}_t \right] \tag{3.18}$$

with different constraints to guarantee that the update isn't too drastic (which negatively impact the complexity). The $\hat{\mathbb{E}}_t$ stands for the empirical average over a finite batch of samples [SLM+15].

Inside the objective we have:

$$r_t(\theta) = \frac{\pi(A_t|S_t;\theta)}{\pi_{old}(A_t|S_t;\theta)} \tag{3.19}$$

that represents the difference between the new and the old policy function, which means that if the value is greater than $1$ the new policy has a greater appreciation for the action $A_t$ in state $S_t$ over the old policy.

In contrast, PPO simplifies the constraints by applying them directly to the objective function:

$$L^{CLIP} = \hat{\mathbb{E}}_t \left[ \min \left( \underbrace{r_t(\theta)\widehat{Adv}_t}_{3.18}, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\widehat{Adv}_t \right) \right] \tag{3.20}$$

This objective takes the minimum between the previous objective and the same objective with a clipped $r_t(\theta)$ where $\epsilon$ is a small hyperparameter value.

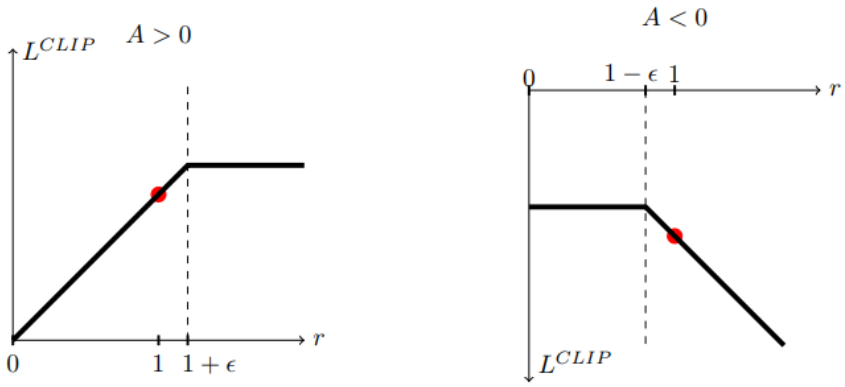


***Figure 3.2:*** *Diagrams showing the objective function with respect to $r(\theta)$ [SWD+17]. (note: A stands for Adv)*

It is important to note two things from these plots:

- When $Adv > 0$ meaning that the action taken was estimated to be good
  - $0 < r < 1$: our new policy has decreased the probability of taking the action which is not good. In fact our objective has a positive gradient (will increase the probability of that action during the next policy update). There is a margin where we still increase the probability although the policy already did which is when $1 \leq r < 1+\epsilon$
  - $r > 1 + \epsilon$: our policy already improved the probability of taking the action so we don't want to overshoot and increase it more (0 gradient)
- When $Adv < 0$ meaning that the action taken was estimated to be bad
  - $0 < r < 1 - \epsilon$: our new policy has already decreased the probability of taking the action so we don't want to overshoot decreasing further.
  - $r > 1$: our new policy has a a greater probability for that action which was estimated to be bad so the objective will have a negative gradient to adjust the probability. See that we're also allowed to lower the probability of the action in case $r > 1 - \epsilon$.

From this diagram the actor-critic structure is also very clear. The actor estimates the policy while the critic tells whether the actions taken were good or bad causing the policy to adjust.

# 4

# DeepRL in Unity3D

As we saw in the previous chapter PPO is the current state of the art reinforcement learning method used for continuous state and action tasks. This makes it a perfect candidate to test and evaluate its performance for different tasks.

Unity3D is a game engine that is having an increasing success for its easy use to create games. The engine is expanding every year introducing new helpful tools and features that aim to improve the workflow even further. Not only game related tools have been released but also features that can be helpful for research. In fact on September 2017 they released the beta of a deep reinforcement learning open-source framework that has been implemented using PPO [Uni17a]. That's exactly what we will be using to test the performance of PPO.

Before showing our results we will do a brief introduction on the components of the framework and on its use. We will be discussing the most recent release of the beta, namely version 0.4b [Uni17c].

Additionally all the results we will be discussing are run on a single computer with a NVIDIA GeForce GTX 1060 3GB

## 4.1 ML-Agents

### 4.1.1 Learning Environment

The main objects that define the learning environment are the *agent*, the *brain* and the *academy*:

- Firstly we have the **agent** object that receives the unique observations (states) and based on those the brain will decide which action to choose in order to maximize the reward. Most of the time our problem will be to define what states we want our agent to receive,

what actions can the agent perform and in what case will the agent get a good or a bad reward.

- Secondly we have the **brain** for which different agents are connected to. The brain is responsible for deciding which actions the agents will take. In this case we will have to decide what type of brain we have (internal or external explained later) and what type of state space and action space we're dealing with (discrete of continuous) along with the size of the actions and states.
- Lastly the **academy** which defines the general settings of the environment such as the speed and rendering quality of the game engine and the global episode length (after that all agents are reset).

When learning, the brain will be set to *external*, which means that the actions and decisions are made using TensorFlow (an open-source library for Machine Learning in python) thanks to a communication over an open-socket to the python API implemented by the Unity team.

Moreover, after the training a bytes file that encapsulates the trained model gets created. Using the brain in *internal* mode we have the chance to use those models and see how good out agents learned.

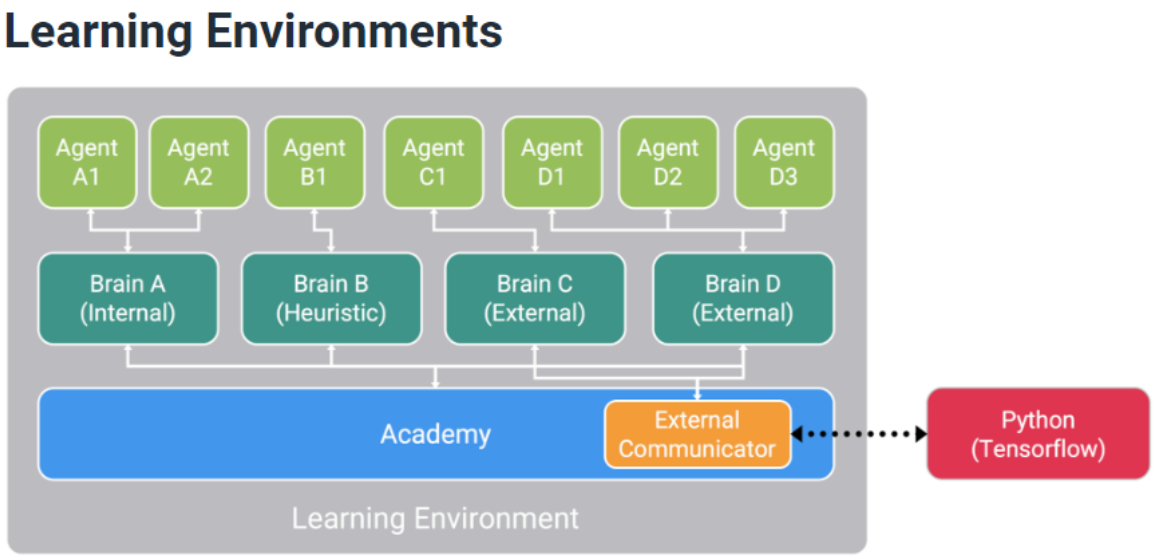


***Figure 4.1:*** *Structure of the Learning Environment [Uni17a]*

### 4.1.2 Hyperparameters

During the next sections we will present different reinforcement learning problems we tried to solve. Before training an agent in an environment we have the choice of selecting the *hyperparameters*. These parameters are meant to tune the settings of our training. When not explicitly noted we will be using these values for our hyperparameters:

- *batch size* **1024**: number of experiences used for one iteration of adam update

- *buffer size* **10240**: the size of collected experience before updating the model based on them (observations, actions, rewards)
- *num epoch* **3**: number of passes through the experience buffer during gradient descent
- *epsilon* **0.2**: this value is the acceptance for the variance between the new and the old policy (see 3.20)
- *gamma* **0.99**: discount factor for the future rewards
- *beta* **0.005**: regularization term for entropy used for randomized action (incentivate exploration)
- *learning rate* **0.0003**: step size of gradient update
- *max steps* **variable**: the maximum amount of steps that are run during training (after that the model file will be created)
- *num layers* **2**: number of hidden layers in our neural network
- *hidden units* **128**: number of units for each fully connected layer

### 4.1.3 Hello World: Training our first agent

In this section we start by showing the first agent we implemented to get a grasp of how the implementation works.

The problem consist of a player and a target and the purpose is for the player to reach the target. For the academy we will be using a generic academy, hence one that is empty and takes care of the different methods that has to be called from the Agent and the Brain.

We will set our brain as having 1 observation and 1 action and both the state and the action space will be set to *continuous*.

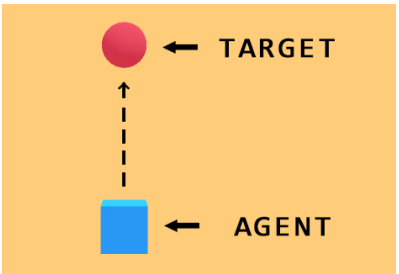


***Figure 4.2:*** *Agent and target.*

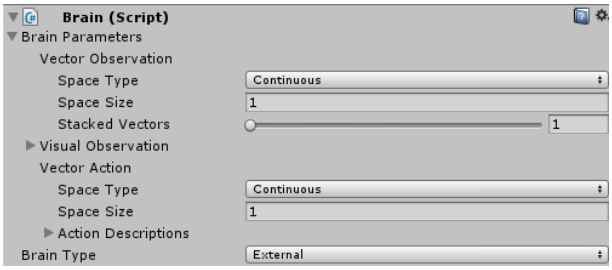


***Figure 4.3:*** *State and action space set to continuous*

We implement the agent in the following way:

```
1     public override void AgentReset()
2     {
3         Agent.pos = new Vector3(0,Random.Range(minY,maxY),0)
4         Target.pos = new Vector3(0,Random.Range(minY,maxY),0)
5         onStartDistance = Mathf.Abs(Agent.pos - Target.pos);
6     }
```

***Listing 4.1:*** *Agent Reset (pseudocode)*

This method will be called each time the agent reaches the max steps or touches the target. What we do here is to just position the agent and the target randomly on the vertical axis between some predefined variables *minY* and *maxY*. We also save of the initial distance between the agent and the target inside the variable *onStartDistance*.

Now our agent has to collect observation and this is done in the following method:

```
1     public override void CollectObservations()
2     {
3         AddVectorObs(Target.pos.y - Agent.pos.y);
4     }
```

***Listing 4.2:*** *Collecting Observations (pseudocode)*

We only get one observation which will be passed to the brain to decide which action to choose. The neat trick here is to use the distance between the two objects instead of using their positions separately leading to two separate observations. As a matter of fact we should always try to keep the number of observations as low as possible while allowing the agent to have enough information to complete the task. Now the only thing that is missing is to define how the action selection will affect the agent and what kind of reward system to use. All this is implemented inside the AgentAction method:

```
1     public override void AgentAction(float[] vectorAction,
2                                      string textAction)
3     {
4         float action_y = vectorAction[0];
5         action_y = Mathf.Clamp(action_y, -1, 1);
6         action_y = action_y * AgentSpeed;
7
8         Agent.pos += new Vector3(0, action_y, 0)
9
10        if (TargetAgentDistance < onDistanceTargetDone)
11        {
12            SetReward(1);
13            Done();
14        }
15    }
```

***Listing 4.3:*** *Agent Action (pseudocode)*

The actions that are decided by the brain are stored inside *vectorAction*. Since we only have one action we can get the value of that action as the first entry of that array. Furthermore note that we clamp the action between -1 and 1. We do this process because we don't want the network to shoot the agent far away. The brain might learn to move the agent upwards by outputting values higher than 1 and downwards with values lower than -1. With this clamping we are sure that the agent will move at most at a step size that equals *AgentSpeed*.

Finally if the distance between the agent and the target is small enough we set the reward to be $+1$ and mark the *done* flag as true which will call $AgentReset()$.

Even though this would seem to be enough there is an additional reward that we generally want to set in order to speed up the learning. That would be to give a penalty for each step so that the agent won't lose steps wandering around or staying on its place. So we add a reward of $-0.1$.

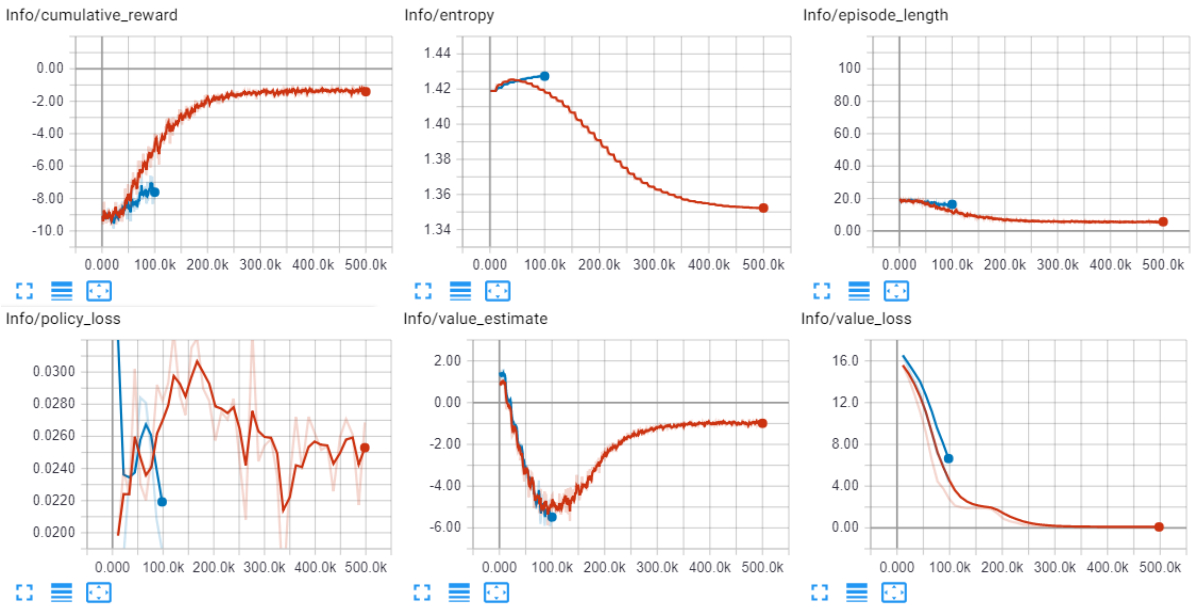


***Figure 4.4:*** *Training plots.*

The above figure represents different useful plots that give us an intuition on how well the training process has been. On the x-axis we have the steps which reaches a maximum of $500'000$. We depict two instances of a training, the blue line has learned with *max step* $= 100'000$ and the red one with *max step* $= 500'000$. As we can see to have a good result only $100'000$ steps wouldn't be enough, in fact from the plots we can see that after $250'000$ steps it starts maintaining the good result. This is observable firstly from our *cumulative reward* that gets stabilized around $-1.5$. The entropy decreases slowly meaning that less random actions are performed. The *policy loss* correlates to how much the policy is changing, in fact it starts to stabilize during the end. The *value loss* converges to $0$ because the value estimate gets more accurate during the training. And finally the *episode length* obviously decreases since the agent reaches the target faster.

Note that it is normal for the cumulative reward to be negative. This is because we set the maximum steps for the agent to be $100$. For each step it gets a penalty of $-0.1$. So if it stayed on its place it would get a cumulative reward of $100 * -0.1 = -10$. On the other hand if it took

the agent 21 steps to reach the target the reward would be $20 * -0.1 + 1 = -2 + 1 = -1$ which is a really good result.

A last improvement that we can make is to use multiple agents instead of only one for solving this task. Recall that with multiple agents we can gather more experience to update our policy and value function. Each agent is still connected to one global brain. The results are shown


***Figure 4.5:*** *Light Blue: multiple agents training. Red: one agent training.*

| | Convergence after step | Convergence in time |
|---|---|---|
| 1 Environment | $\sim 250'000$ | $\sim 15m$ |
| 32 Environments | $\sim 25'000$ | $\sim 1m30s$ |

***Table 4.1:*** *First Environment: Learning time*

To sum up we managed to train a simple agent to catch a ball while allowing it to move up or downwards.

### 4.1.4 Gravity Ball: 2D

In this section we present a similar problem but with different changes that makes it more difficult. Same as before we have an agent and a target but this time the way the agent moves is through forces applied to the rigidbody. In this environment the agent is also affected by gravity.

For this problem can still be solved using one single continuous action. Specifically that action will be used as the angle for the polar coordinates in 2D. Recall that we can convert from polar coordinates to cartesian with the following equations:

$$\begin{aligned} x &= rcos\varphi \\ y &= rsin\varphi \\ &\text{where r is the radius and } \varphi \in [-\pi, \pi] \end{aligned} \tag{4.1}$$

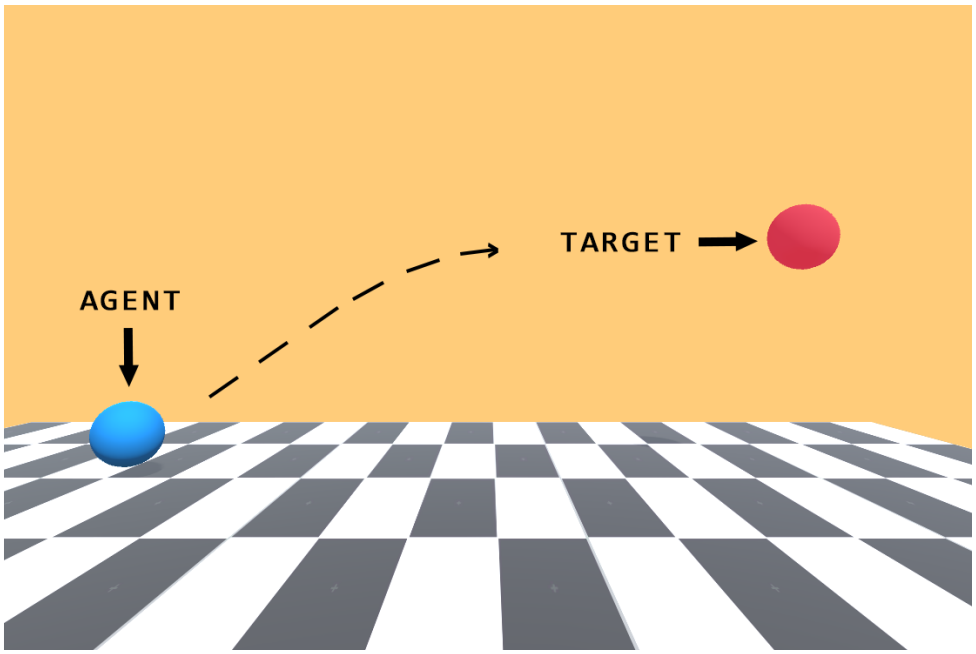


***Figure 4.6:*** *GravityBall agent and target.*

With respect to this problem we need to add more observations.

```
1     public override void CollectObservations()
2     {
3         AddVectorObs(Agent.pos.x - Target.pos.x);
4         AddVectorObs(Agent.pos.y - Target.pos.y);
5
6         AddVectorObs(AgentRB.velocity.x);
7         AddVectorObs(AgentRB.velocity.y);
8     }
```

***Listing 4.4:*** *Collect Observations (pseudocode)*

Where *AgentRB* is the variable referencing the rigidbody of the agent. In addition to the last problem we need to add the distance from the target in the $y$-axis and the velocity of the agent so that the agent can adjust the direction of the force to be applied.

The action method is implemented in the following way:

```
1     public override void AgentAction(float[] vectorAction,
2                                      string textAction)
3     {
4         //(1)
5         float phi = vectorAction[0];
6         AgentRB.AddForce(maxForce * Mathf.Cos(phi),
7                          maxForce * Mathf.Sin(phi), 0);
8
9
```

```
10          float currentAgentTargetDist = (Agent.pos -
11                                          Target.pos).magnitude;
12          //(2)
13          AddReward(-(currentAgentTargetDist / onStartDistance)
14                      / agentParameters.maxStep);
15
16          //(3)
17          if (Agent.localPos.x > rightBoundary) {
18              SetReward(-0.8f);
19              Done();
20          }
21          //(4)
22          if (currentAgentTargetDistance < onDistanceTargetDone)
23          {
24              SetReward(2f - (GetStepCount()/agentParameters.maxStep));
25              Done();
26          }
27      }
```

***Listing 4.5:*** *Agent Action (pseudocode)*

- **(1)**: we take the action decided by the brain and with that we apply a force to the rigidbody (see eq. 4.1). Note that the agent is only responsible for changing the direction of the applied force. The magnitude is stored inside the *maxForce* variable (in our case we used 20).
- **(2)**: differently from the previous case, instead of adding a fixed penalty of $-0.1$ at each step we apply a variable penalty that depends on the distance from the target. This should give a hint to the agent that reducing that distance yields in less penalty.
- **(3)**: this could be optional, but we don't want the agent to shoot too far away from the target and we immediately finish the episode in that case giving a negative reward. The same is done for the left boundary and ceiling boundary. This should the speed of learning.
- **(4)**: when we reach the target we give a reward that has the form $2-(\text{currentStepCount}/\text{maxStep})$. This makes it so that the agent learns to reach the target as fast as possible which yields more reward.

Be **careful** when giving negative rewards, especially with boundaries. If hitting a boundary is very probable and it gives a lower reward than to do nothing during the episode then the agent would learn to stay on its place (local optima).

Moreover note that we decided to increase the max step for the agent (number of step per episode) to $200$ so that it has enough time to reach the target if it is far.

With this implementation we achieved good results by using 32 environments and running for $100'000$ steps stabilized the training, although sometimes we had some spikes in the cumulative reward (see fig. 4.7) given by some bug where the environment freezed for some time during training. But these spikes didn't influence the end result at all.

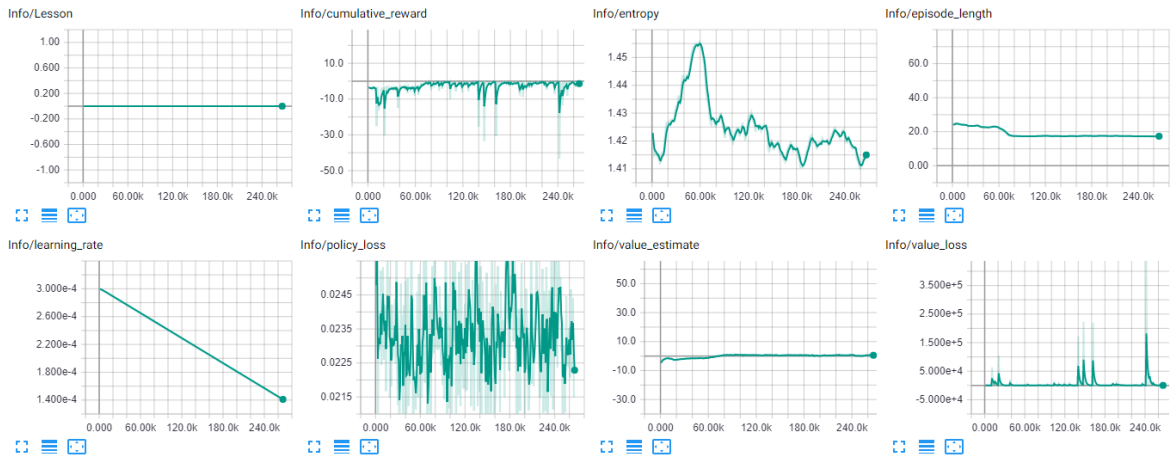


***Figure 4.7:*** *Trained using 32 agents. Agents max step* $= 200$

### 4.1.5 Gravity Ball: 3D

The problem gets harder when we allow the target to be anywhere in the 3D world space. In fact the agent needs to have at least two actions in order to move in this space and also the number of observation is increasing leading to longer training sessions.

For this task we use the spherical coordinates, that we can convert into cartesian coordinates following the equation:

$$
\begin{aligned}
x &= r sin\theta cos\varphi \\
y &= r sin\theta sin\varphi \\
z &= r cos\theta \\
&\text{where r is the radius and } \theta \in [0, \pi],\ \varphi \in [0, 2\pi]
\end{aligned} \tag{4.2}
$$

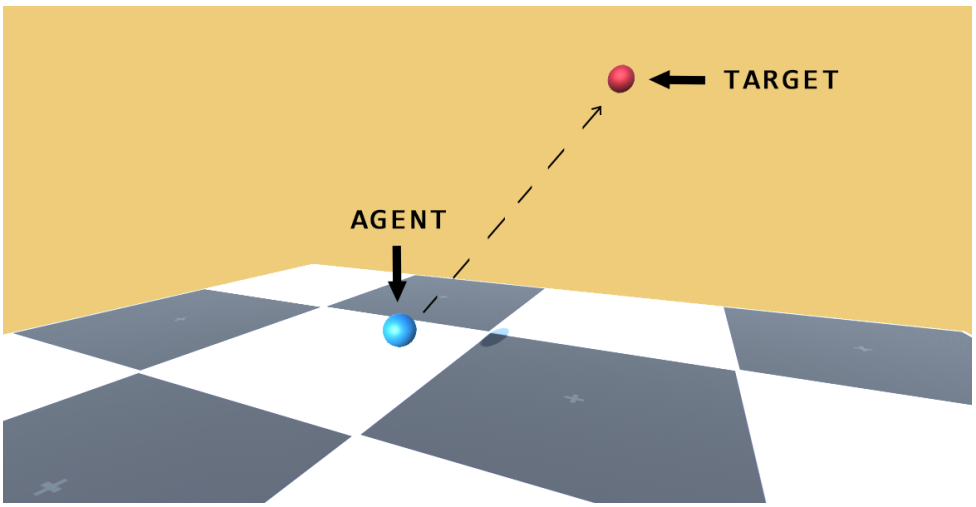


***Figure 4.8:*** *GravityBall 3D*

We will be training our agent with 2 actions, 6 observations and max step $= 250$. The observations we collect are:

```
1         public override void CollectObservations()
2     {
3         AddVectorObs(Agent.pos.x - Target.pos.x);
4         AddVectorObs(Agent.pos.y - Target.pos.y);
5         AddVectorObs(Agent.pos.z - Target.pos.z);
6
7         AddVectorObs(AgentRB.velocity.x);
8         AddVectorObs(AgentRB.velocity.y);
9         AddVectorObs(AgentRB.velocity.z);
10    }
```

***Listing 4.6:*** *Collect Observations (pseudocode)*

In contrast from other example we also need to add the z component of the distance and the velocity of the agent.

For the action method we did the following changes:

```
1     public override void AgentAction(float[] vectorAction,
2                                      string textAction)
3     {    //(1)
4         float theta = action[0];
5         float phi = action[1];
6         AgentRB.AddForce(maxForce * Mathf.Sin(theta) * Mathf.Cos(phi),
7                          maxForce * Mathf.Sin(theta) * Mathf.Sin(phi),
8                          maxForce * Mathf.Cos(theta));
9
10        float currentAgentTargetDist = (Agent.pos -
11                                        Target.pos).magnitude;
12        //(2)
13        AddReward(-(currentAgentTargetDist / onStartDistance)
14                    / agentParameters.maxStep);
15
16        //(3)
17        if (Agent.localPos.x > rightBoundary ||
18            Agent.localPos.y > ceilingBoundary ||
19            Agent.localPos.z > depthBoundary) {
20            SetReward(-0.8f);
21            Done();
22        }
23        //(4)
24        if (currentAgentTargetDistance < onDistanceTargetDone)
25        {
26            SetReward(agentParameters.maxStep - GetStepCount());
27            Done();
28        }
29    }
```

***Listing 4.7:*** *Agent Action (pseudocode)*

- **(1)**: the brain has to choose 2 actions and we use them as described in the equation 4.2.
- **(2)**: remains unchanged.
- **(3)**: as previously mentioned we add some boundaries to stop the episode if it misses the target.
- **(4)**: this is an alternative from the previous reward when touching the target. In fact in this way we still encourage the agent to reach the target as fast as possible. We really emphasize that reaching the target yields a lot of reward.

During training the agent may learn a suboptimal behaviour instead of reaching the target, such as not moving much, but only slowly in the direction of the agent without really getting close. Having bigger values as reward has helped to achieve an optimal behaviour. Another issue is that having two actions for controlling the ball requires more training. The way we used to speed up the control of the ball is to not allow the agent to hit the ground (adding it to our bounds). In this way it would learn faster to levitate and move towards the right direction.

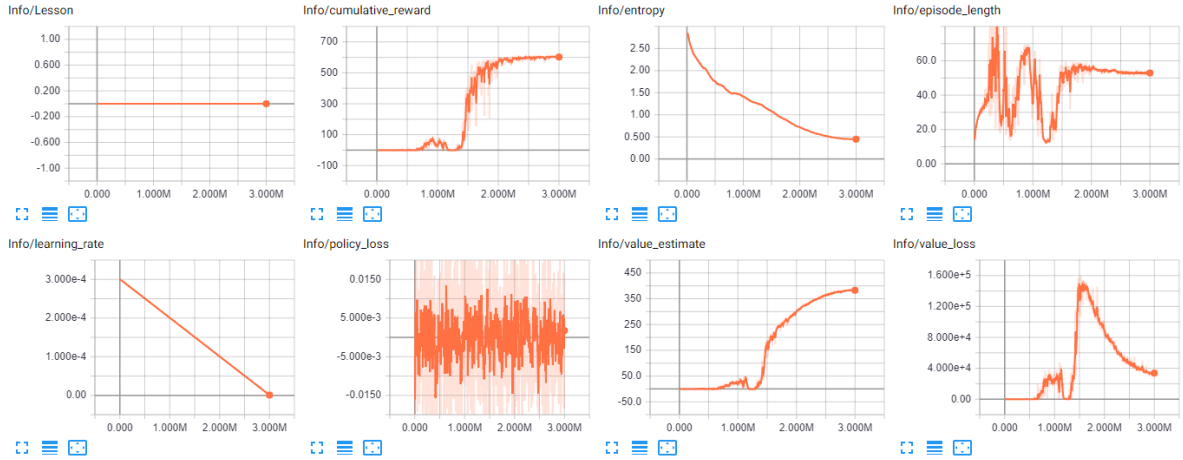


***Figure 4.9:*** *Trained using 32 agents. Agents max step* $= 250$

| | Convergence after step | Convergence in time |
|---|---|---|
| GravityBall 2D | $\sim 100'000$ | $\sim 10m$ |
| GravityBall 3D | $\sim 1'500'000$ | $\sim 1h30m$ |

### 4.1.6 Car Environment

In this section we will be discussing how we managed to teach a brain to drive a car and avoid obstacles. Since this problem is much more complex from the others we used a tool that helped a lot to achieve our final result. We're talking about *curriculum learning* that will be introduced afterwards (section 4.1.8). Initially we wanted to make sure to be able to solve the first task of driving without any obstacles.

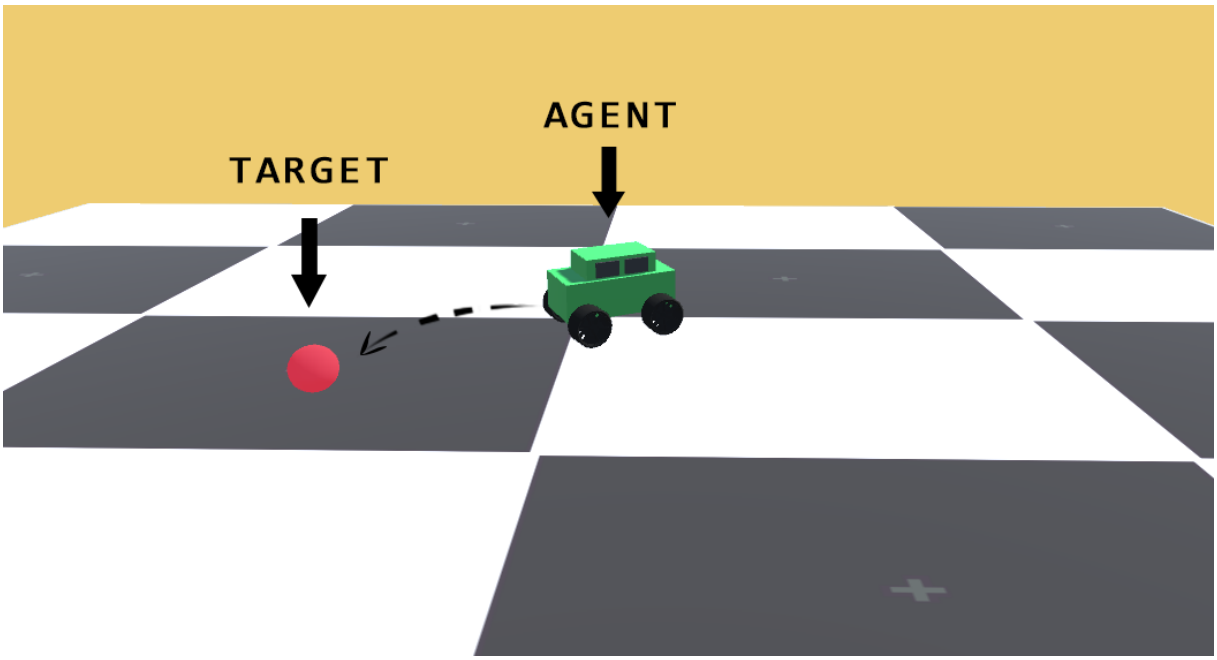


***Figure 4.10:*** *Car Environment*

The rear wheels of our car move the vehicle while the front wheels are used for steering. With respect to this settings our car will be moved using 2 actions. One action for the motor and one action for steering. So besides the velocity and the distance of the car from the target we will add the speed and the current steering angle of the front wheels to the observations.

```
1     public override void CollectObservations()
2     {
3         AddVectorObs(Agent.pos.x - Target.pos.x);
4         AddVectorObs(Agent.pos.z - Target.pos.z);
5
6         AddVectorObs(AgentRB.localVel.x);
7         AddVectorObs(AgentRB.localVel.z);
8
9         AddVectorObs(GetSteeringAngle());
10        AddVectorObs(GetMotorTorque());
11    }
```

***Listing 4.8:*** *Agent Action (pseudocode)*

Note that we are using the local velocity of the car so that we can keep track of the orientation of the car.

The action method is defined as:

```
1     public override void AgentAction(float[] vectorAction,
                                       string textAction)
2     {
```

```
        //(1)
        float motorTorque = Mathf.Clamp(vectorAction[0], -1, 1);
        motorTorque *= 200;

        //(2)
        float steeringAngle = Mathf.Clamp(vectorAction[1], -1, 1);
        steeringAngle *= 60;

        //(3)
        rearAxle.SetAxleMotorTorque(motorModel);
        frontAxle.SetAxleSteering(angleModel);

        float currentAgentTargetDist = (Agent.pos -
                                        Target.pos).magnitude;
        //(4)
        AddReward(-(currentAgentTargetDist / onStartDistance)
                    / agentParameters.maxStep);

        //(5)
        if (currentAgentTargetDistance < onDistanceTargetDone)
        {
            SetReward(10f);
            Done();
        }
    }
```

***Listing 4.9:*** *Collect Observations (pseudocode)*

- **(1)**: fetch action to move the car.
- **(2)**: fetch action to steer the car. Notice that we set the maximum angle is 60 degrees.
- **(3)**: apply the actions that we got from the brain.
- **(4)**: global negative reward in function of the distance (same as previous).
- **(5)**: a fixed reward of 10 when reaching the target.

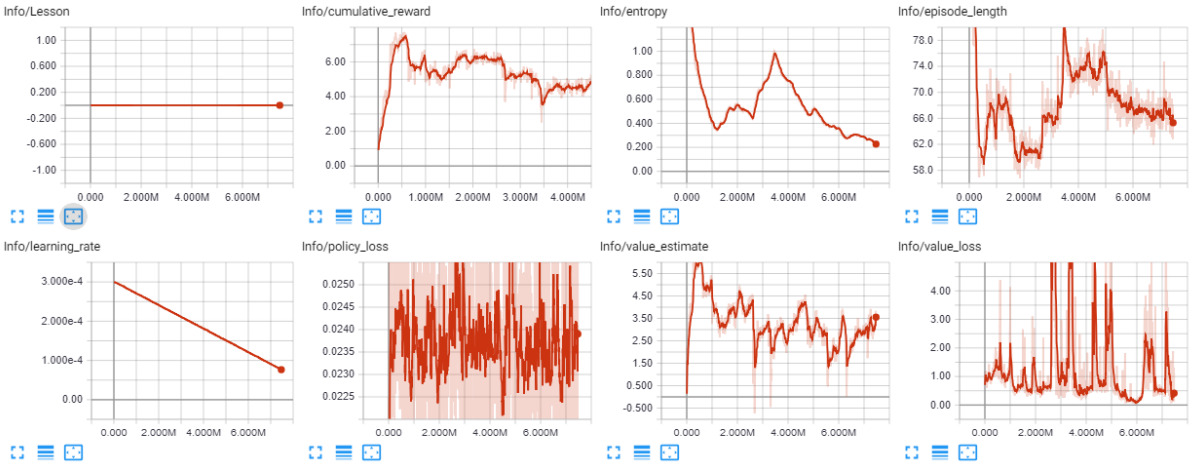


***Figure 4.11:*** *32 Environments with max step* = 1000.

We trained the car with a target that was spawning randomly on the ground (also behind the car). Since there is a lot of variation between each position, learning to move for each of those becomes a hard task. In fact if the target spawns more often in front of the agent the weights of the policy will be more accurate in predicting the actions to take when the ball is in front. From the plot 4.11 the car learns pretty fast but after some steps it does worse. That is because initially more targets have been spawned in front of the agent. After a while the agent became more careful and stabilized its speed to chase also targets that are behind.

### 4.1.7 Car Environment: Vertical Obstacle

Now that we managed to train the car to learn moving, we want to add another layer of complexity by adding obstacles.

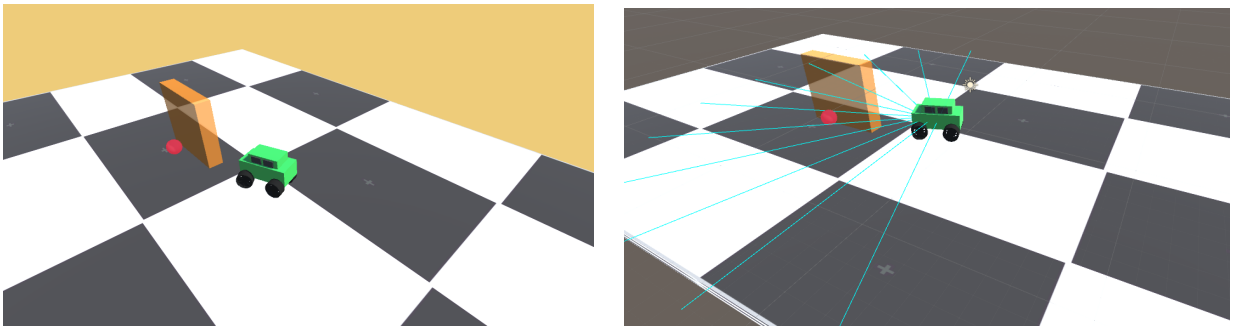

***Figure 4.12:*** *Environment with an obstacle placed vertically with a target randomly spawning on one side or the other.*

The target will be placed randomly on one side or the other of the obstacle. The obstacle itself is in front of the car in a random position in every episode. In this case the number of observations increases significantly because we need some information of what is surrounding the car. We will be using *Physics.SphereCast(ray, radius, length)*, which is a ray that gets cast from the car with a given length. SphereCast is a variant of RayCast, where a radius is given for a sphere placed at the end of the ray. If the sphere collides with an obstacle a hit will be reported. This allows to have less rays that cover more space.

In particular we used 11 rays facing the car (see 4.12) of length 15 units. For the implementation we left the action method unchanged and modified the *CollectObservations()* method. Now our current observations size is 17 (the same 6 of before + 11 rays).

Note that we also give a negative reward whenever the car collides with the wall. We use the same magnitude as for catching the ball, but negative, hence $-10$.

For each ray we encoded the information to be either $-1$ if no hit occured for that ray or a normalized value of the distance of the hit point: *hitDistance / rayLength*.

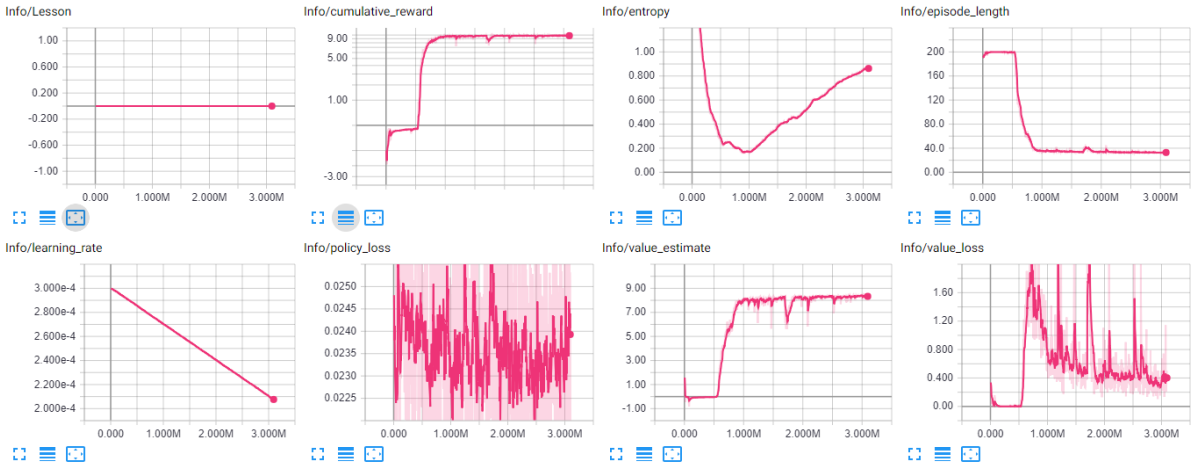


***Figure 4.13:*** *32 Environments with vertical obstacle with max step* $= 1000$.

Initially the agent learned to randomly go on one side of the obstacle in the hope of finding the target. After a lot of iterations 4.2 the agent managed to understand the relations between the distance of the hitting rays and the distance from the target to spot on which side the target is.

### 4.1.8 Car Environment: Horizontal Obstacle and Curriculum Learning

Previously the agent had a good chance to sample enough experience with the target on each side of the obstacle. What if we placed the obstacle horizontally? Well in that case it becomes a problem. That is because whenever the ball is hidden the agent wouldn't know that the ball is on the other side although he has an observation telling him the direction. In this case the agent prefers to just get near the wall and wait (get as near as possible to the ball) without exploring too much because colliding with the wall yields a negative reward. As a result we have a local optima.

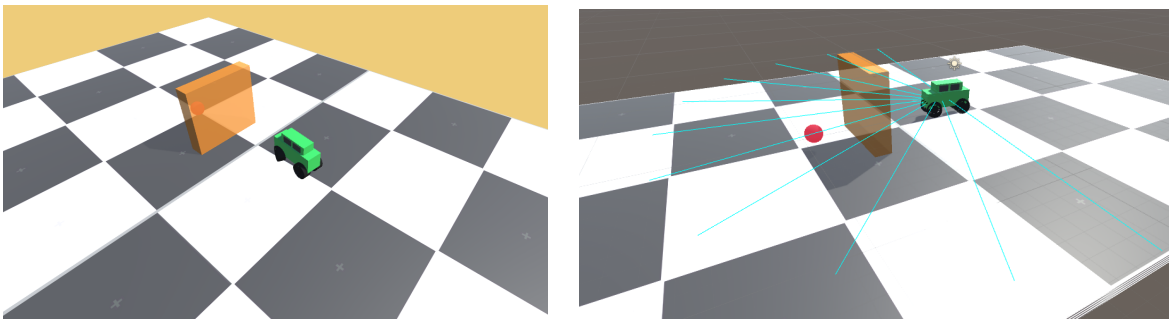

***Figure 4.14:*** *Environment with an obstacle placed horizontally with a target randomly spawning on one side or the other.*

To avoid this and let the agent know that he has to go beyond the wall we can use *curriculum learning*. This is a really powerful tool that can be applied to any other environment we have seen so far to get better results.

The idea of curriculum learning is to divide a complex problem into smaller subsets of the same problem. In our case we would want to have initially the wall as small as possible and gradually have it increase in size every time the agent masters one sub task. These sub tasks are called *lessons* in the ML-Agents framework [Uni17b].

The parameters that have to change during the learning have to be set inside the Academy as *ResetParameters*:

```
1    public override void AcademyReset()
2    {
3        //Assignment of global variables
4        MinTargetDist_X = resetParameters["MinTargetDist_X"];
5        MaxTargetDist_X = resetParameters["MaxTargetDist_X"];
6        MinTargetDist_Z = resetParameters["MinTargetDist_Z"];
7        MaxTargetDist_Z = resetParameters["MaxTargetDist_Z"];
8        MinObstacleLength = resetParameters["MinObstacleLength"];
9        MaxObstacleLength = resetParameters["MaxObstacleLength"];
10
11   }
```

***Listing 4.10:*** *Academy (pseudocode)*

These parameters can be affected during training. A *.json* file should be created where we define for each lesson the values for each variable [Uni17b]. Unity didn't release any tool for creating these files. As a matter of fact in general this is done manually.

The ideal approach would really be to take the hand of our agent and walk it through the process of solving harder tasks gradually without changing it too drastically from one lesson to the other. As a result we wrote our own program that can create huge *.json* files by interpolating the variables between a max and a min and also adding freely a set of fixed values and so on. With this tool we created our curriculum file and we got great results in a short time compared to the other environments (see table 4.2).

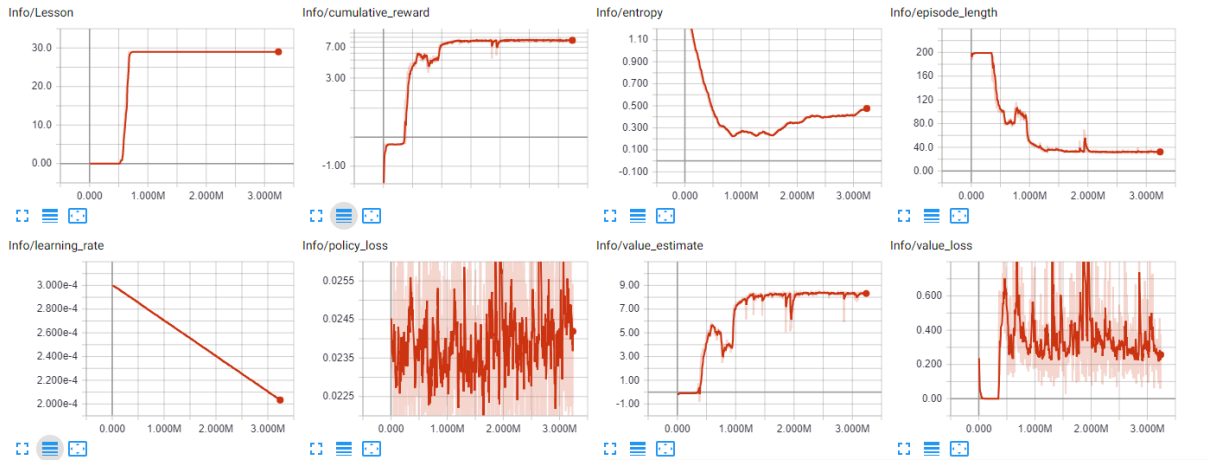


***Figure 4.15:*** *32 Environments with horizontal obstacle with max step* $= 1000$*. Using curriculum with 30 lessons.*

### 4.1.9 Car Environment: Parking

Finally we implemented a parking environment. From all the things we've learned so far we managed to make an agent capable of parking.

We used the same implementations as for the *Horizontal Obstacle* with the same number of observations and rays.

Moreover we created a curriculum file that kept all the cars distant and the target alone near the car. With this lesson the car learned fast to move and catch the target. While the agents makes progress the target moves slowly towards the cars until it stops at the parking spot.

Another neat trick is to increase the number of steps required before a decision is taken, hence before the action method gets called. In this way we reduce jittering effects from the agent.

The agent finally learns to park avoiding the other cars in a pretty accurate way.

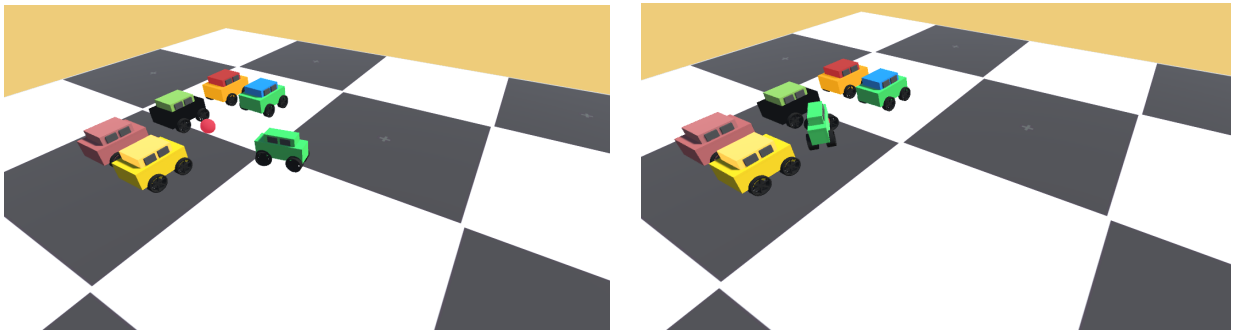

***Figure 4.16:*** *Car Parking*

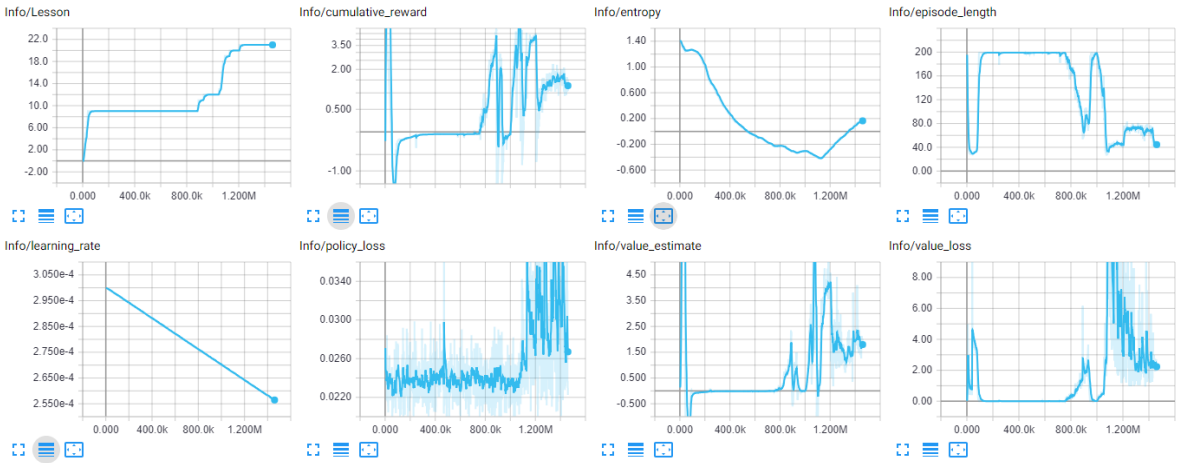


***Figure 4.17:*** *32 Environments with horizontal obstacle with max step* $= 1000$*. Using curriculum with 30 lessons.*

Note that the cumulative reward goes up and down because of the change of the lessons. Since it has to learn with a new environment the reward suffers a drop. That is perfectly normal when using curriculum learning.

| | Convergence after step | Convergence in time |
|---|---|---|
| Car Environment | $\sim 2'000'000$ | $\sim 5h$ |
| Car Vertical Obstacle | $\sim 3'000'000$ | $\sim 8h30m$ |
| Car Horizontal Obstacle | $\sim 1'100'000$ | $\sim 3h$ |
| Car Parking | $\sim 1'500'000$ | $\sim 4h$ |

***Table 4.2:*** *Car Environment*

# 5

# Conclusion and Outlook

In this thesis we did a brief introduction on how Reinforcement Learning was born and has grown in the past years. In the early days we were able to develop methods to learn from discrete spaces (state and action) and later on come up to practical approaches for solving problems with continuous spaces. The process to go from Dynamic Programming approaches to Monte Carlo and finally to Policy Gradient based approaches using function approximators such as Neural Networks has been a long journey that is finally paying off.

A lot of interest was born in *Deep RL* and it is becoming more and more accessible to average users with affordable computing power. Thanks to companies such as Unity3D we were able to get a better understanding of the state of the art Deep RL implementation, PPO [SWD+17] [Ope17a]. We managed to go from an easy task to be learned to increasingly harder and more challenging problems that required us to think and understand how to properly set up different reward systems and how to structure our implementation.

We went from implementing an easy problem to a more challenging one during the end of the thesis. Initially we trained an agent that learns to reach a target with movement limited only in one dimension. Afterwards we added more complexity, such as gravity and movement controlled through application of forces on the agent. This led to our final problem of training a car. The first task was to learn to drive and afterwards to avoid obstacles trying to reach a target. In fact with this last implementation we managed to make a car park on a given spot avoiding collisions.

What we observed in the general context of setting up an agent is:

- to encapsulate all the needed information for the agent into the smallest possible amount of observations.
- when fetching actions from the brain sometimes it is useful to clamp the value between some interval and if necessary to multiply the clamped value by a constant.

- to increase number of steps needed in order to make a decision (stabilizes and reduced jittering effects).
- to take distances between objects instead of their single positions when that's the relation the agent actually needs.
- to normalize values you pass as observations whenever it makes sense to do so.
- to make sure that the agent is able to sample as many experience as possible from all the different scenarios that the agent needs to understand. For instance in our last example we needed to make sure that the car knew that the wall is bad and the target is good.
- to use curriculum learning to divide the training process into multiple increasingly difficult lessons. It takes some time to come up with the variable part of the environment, with the set up and with the creation of the *.json* file, however the time "lost" to do that pays off during the actual training of the agent.

We think that in the future the interest in these fields will increase even further. As a matter of fact ML-Agents, the framework proposed by Unity3D, has already captivated a lot of people from different fields such as Game Development, Researchers or just Hobbyists. Although it engaged lots of people it has still a big room for improvement. The process of training an agent still requires to open the terminal and run some commands. In addition to this in order to set up an environment a lot of manual work in terms of object placement (Brain object has to be manually placed as a child of an Academy object and so on) has to be done. Furthermore the process of constructing a curriculum learning training is very tedious and could be improved by implementing an editor that gives an overview over all the *resetParameters* and the different stages of the curriculum.

# Bibliography


[ANTH96] Minoru Asada, Shoichi Noda, Sukoya Tawaratsumida, and Koh Hosoda. Purposive behavior acquisition for a real robot by vision-based reinforcement learning. *Machine learning*, 23(2-3):279–303, 1996.

[Bel13] Richard Bellman. *Dynamic programming*. Courier Corporation, 2013.

[CS15] Marco Corazza and Andrea Sangalli. Q-learning and sarsa: a comparison between two intelligent stochastic control approaches for financial trading. 2015.

[Dav15] David Silver. Policy gradient methods. `http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/pg.pdf`, 2015. [Online; accessed 31-August-2018].

[Dee17] DeepMind. Deepmind: Alphago. `https://deepmind.com/research/alphago/`, 2017. [Online; accessed 01-September-2018].

[GP12] T. Geijtenbeek and N. Pronost. Interactive character animation using simulated physics: A state-of-the-art review. *Comput. Graph. Forum*, 31(8):2492–2515, December 2012.

[KT00] Vijay R Konda and John N Tsitsiklis. Actor-critic algorithms. In *Advances in neural information processing systems*, pages 1008–1014, 2000.

[MBM+16] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[MKS+13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[Ope17a] OpenAI. https://blog.openai.com/openai-baselines-ppo/. https://blog.openai.com/baselines-acktr-a2c/, 2017. [Online; accessed 01-September-2018].

[Ope17b] OpenAI. Openai: A2c, a3c, acktr. https://blog.openai.com/baselines-acktr-a2c/, 2017. [Online; accessed 01-September-2018].

[Ope17c] OpenAI. Policy gradient methods. https://blog.openai.com/baselines-acktr-a2c/, 2017. [Online; accessed 01-September-2018].

[PBvdP16] Xue Bin Peng, Glen Berseth, and Michiel van de Panne. Terrain-adaptive locomotion skills using deep reinforcement learning. *ACM Trans. Graph.*, 35(4):81:1–81:12, July 2016.

[PS08] Jan Peters and Stefan Schaal. Natural actor-critic. *Neurocomputing*, 71(7-9):1180–1190, 2008.

[SB18] Richard S Sutton and Andrew G Barto. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 2018.

[SHM+16] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

[SK02] William D Smart and L Pack Kaelbling. Effective reinforcement learning for mobile robots. In *Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on*, volume 4, pages 3404–3410. IEEE, 2002.

[SLM+15] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015.

[SMSM00] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.

[SWD+17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[Tho18] Thomas Simonini. Policy gradient methods. https://medium.freecodecamp.org/an-introduction-to-deep-q-learning-lets-play-doom-5 2018. [Online; accessed 31-August-2018].

[Uni17a] Unity3D. Ml-agents. https://github.com/Unity-Technologies/ml-agents, 2017. [Online; accessed 01-September-2018].

[Uni17b] Unity3D. Ml-agents curriculum learning. https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-Curriculum-Learning.md, 2017. [Online; accessed 01-September-2018].

[Uni17c] Unity3D. Ml-agents v0.4b. https://github.com/Unity-Technologies/ml-agents/releases/tag/0.4.0b, 2017.

[Online; accessed 01-September-2018].

[WD92] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[WdFL15] Ziyu Wang, Nando de Freitas, and Marc Lanctot. Dueling network architectures for deep reinforcement learning. *CoRR*, abs/1511.06581, 2015.

[ZZ01] Nevin Lianwen Zhang and Weihong Zhang. Speeding up the convergence of value iteration in partially observable markov decision processes. *Journal of Artificial Intelligence Research*, 14:29–51, 2001.